



## PROJECT DOCUMENTATION

Project Name:	Synopse mORMot Framework
Document Name:	Software Architecture Design
Document Revision:	1.17
Date:	September 9, 2012
Project Manager:	Arnaud Bouchez

### Document License

THE ATTACHED DOCUMENTS DESCRIBE INFORMATION RELEASED BY SYNOPSE INFORMATIQUE UNDER A GPL 3.0 LICENSE.

*Synopse SQLite3/mORMot Framework Documentation.*

Copyright (C) 2008-2012 Arnaud Bouchez.

Synopse Informatique - <http://synopse.info..>

This document is free document; 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.

The *Synopse mORMot Framework Documentation* 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 documentation. If not, see <http://www.gnu.org/licenses..>

### 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:	Title:	Signature:	Date
Arnaud Bouchez	Project Manager		

## 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 1.17.

## Related Documents

Name	Description	Rev.	Date
SDD	Software Design Document	1.17	September 9, 2012
SWRS	Software Requirements Specifications	1.17	September 9, 2012
DI	Design Input Product Specifications	1.17	September 9, 2012

# Table of Contents

---

## Introduction

---

<b>Purpose</b>	28
Document layout	28
First part: global Software architecture	29
Second part: SWRS implications	29
<b>Responsibilities</b>	29
GNU General Public License	29

## 1. Global Architecture

---

<b>1.1. Synapse mORMot Framework Overview</b>	41
1.1.1. Highlights	42
1.1.2. mORMot	42
<b>1.2. Architecture principles</b>	44
1.2.1. Model-View-Controller	44
1.2.2. Multi-tier architecture	45
1.2.3. Service-oriented architecture	45
1.2.4. Object-relational mapping	46
1.2.5. Domain-Driven design	47
<b>1.3. General design</b>	50
1.3.1. SQLite3-powered, not SQLite3-limited	50
1.3.2. Client-Server ORM/SOA architecture	50
<b>1.4. mORMot Framework</b>	53

<b>1.4.1. Object-Relational Mapping</b>	<b>53</b>
1.4.1.1. TSQLRecord fields definition	53
1.4.1.1.1. Text fields	55
1.4.1.1.2. Date and time fields	55
1.4.1.1.3. Enumeration fields	56
1.4.1.1.4. Floating point and Currency fields	56
1.4.1.1.5. Record fields	57
1.4.1.2. Working with Objects	57
1.4.1.3. Queries	58
1.4.1.3.1. Return a list of objects	58
1.4.1.3.2. Query parameters	58
1.4.1.3.3. Introducing TSQLTableJSON	59
1.4.1.3.4. Note about query parameters	60
1.4.1.4. Objects relationship: cardinality	62
1.4.1.4.1. "One to one" or "One to many"	62
1.4.1.4.2. "Has many" and "has many through"	63
1.4.1.4.2.1. Shared nothing architecture (or sharding)	64
1.4.1.4.2.1.1. Arrays, TPersistent, TCollection, TMyClass	65
1.4.1.4.2.1.2. Dynamic arrays fields	66
1.4.1.4.2.1.3. Dynamic arrays from Delphi Code	66
1.4.1.4.2.1.4. Dynamic arrays from SQL code	67
1.4.1.4.2.1.5. TPersistent/TCollection fields	68
1.4.1.4.2.1.6. Custom TObject JSON serialization	71
1.4.1.4.2.2. ORM implementation via pivot table	73
1.4.1.4.2.2.1. Introducing TSQLRecordMany	73
1.4.1.4.2.2.2. Automatic JOIN query	76
1.4.1.5. Calculated fields	78
1.4.1.5.1. Setter for TSQLRecord	78
1.4.1.5.2. TSQLRecord.ComputeFieldsBeforeWrite	79
1.4.1.6. Daily ORM	80



1.4.1.6.1. ORM is not DB	81
1.4.1.6.2. Objects, not tables	81
1.4.1.6.3. Methods, not SQL	81
1.4.1.6.4. Think multi-tier	83
1.4.1.7. ORM Cache	84
1.4.1.8. MVC pattern	84
1.4.1.8.1. Creating a Model	84
1.4.1.8.2. Filtering and Validating	85
1.4.1.8.3. Views	87
1.4.1.8.3.1. RTTI	87
1.4.1.8.3.2. User Interface	89
1.4.1.9. One ORM to rule them all	89
1.4.1.9.1. Rude class definition	89
1.4.1.9.2. Several ORMs at once	90
1.4.1.9.3. The best ORM is the one you need	91
<b>1.4.2. Database layer</b>	92
1.4.2.1. SQLite3-powered, not SQLite3-limited	92
1.4.2.1.1. SQLite3 as core	92
1.4.2.1.2. Extended by SQLite3 virtual tables	93
1.4.2.1.3. Data access benchmark	93
1.4.2.2. SQLite3 implementation	96
1.4.2.2.1. Prepared statement	96
1.4.2.2.2. R-Tree inclusion	98
1.4.2.2.3. FTS3/FTS4	98
1.4.2.2.3.1. Dedicated FTS3/FTS4 record type	99
1.4.2.2.3.2. Stemming	100
1.4.2.2.3.3. FTS searches	101
1.4.2.2.4. NULL handling	102
1.4.2.2.5. ACID and speed	102
1.4.2.3. Virtual Tables magic	103
1.4.2.3.1. Virtual Table module classes	104

1.4.2.3.2. Defining a Virtual Table module	105
1.4.2.3.3. Using a Virtual Table module	107
1.4.2.3.4. Virtual Table, ORM and TSQLRecord	108
1.4.2.3.5. In-Memory "static" process	109
1.4.2.3.5.1. In-Memory tables	109
1.4.2.3.5.2. In-Memory virtual tables	110
1.4.2.3.6. Virtual Tables to access external databases	112
1.4.2.4. External database access	112
1.4.2.4.1. Database agnosticism	112
1.4.2.4.1.1. Direct access to any Database engine	113
1.4.2.4.1.2. Data types	114
1.4.2.4.1.3. SynDB Units	114
1.4.2.4.1.4. Classes and generic use	114
1.4.2.4.1.5. ISQLDBRows interface	116
1.4.2.4.1.6. Late binding	117
1.4.2.4.2. Database access	118
1.4.2.4.2.1. OleDb or ODBC to rule them all	118
1.4.2.4.2.2. Oracle via OCI	119
1.4.2.4.2.3. SQLite3	120
1.4.2.4.3. ORM Integration	120
1.4.2.4.3.1. Transparent use	120
1.4.2.4.3.2. Behind the scene	122
<b>1.4.3. Client-Server</b>	125
1.4.3.1. Involved technologies	125
1.4.3.1.1. JSON	125
1.4.3.1.1.1. Why use JSON?	125
1.4.3.1.1.2. JSON format density	125
1.4.3.1.1.3. JSON format layouts	126
1.4.3.1.1.4. JSON global cache	126
1.4.3.1.2. REST	127
1.4.3.1.2.1. RESTful implementation	127

1.4.3.1.2.2. REST and BLOB fields	128
1.4.3.1.2.3. REST and JSON	128
1.4.3.1.2.4. REST is Stateless	129
1.4.3.1.2.4.1. Server side synchronization	129
1.4.3.1.2.4.2. Client side synchronization	129
1.4.3.1.3. Interfaces	130
1.4.3.1.3.1. Delphi and interfaces	130
1.4.3.1.3.1.1. Declaring an interface	130
1.4.3.1.3.1.2. Implementing an interface with a class	131
1.4.3.1.3.1.3. Using an interface	132
1.4.3.1.3.1.4. There is more than one way to do it	133
1.4.3.1.3.1.5. Here comes the magic	133
1.4.3.1.3.2. SOLID design principles	133
1.4.3.1.3.2.1. Single responsibility principle	134
1.4.3.1.3.2.2. Open/closed principle	135
1.4.3.1.3.2.3. Liskov substitution principle	136
1.4.3.1.3.2.4. Interface segregation principle	137
1.4.3.1.3.2.5. Dependency Inversion Principle	137
1.4.3.1.3.3. Circular reference and (zeroing) weak pointers	137
1.4.3.1.3.3.1. Weak pointers	137
1.4.3.1.3.3.2. Handling weak pointers	139
1.4.3.1.3.3.3. Zeroing weak pointers	139
1.4.3.1.3.3.4. Weak pointers functions implementation details	140
1.4.3.2. Client-Server implementation	141
1.4.3.2.1. Implementation design	141
1.4.3.2.2. Client-Server classes	147
1.4.3.2.3. HTTP client	148
1.4.3.2.4. HTTP server using http.sys	149
1.4.3.2.4.1. Presentation	149

1.4.3.2.4.2. UAC and Vista/Seven support	150
1.4.3.2.5. BATCH sequences for adding/updating/deleting records	151
1.4.3.2.5.1. BATCH process	151
1.4.3.2.5.2. Implementation details	153
1.4.3.2.5.3. Array binding	154
1.4.3.2.6. CRUD level cache	155
1.4.3.2.6.1. Where to cache	156
1.4.3.2.6.2. When to cache	157
1.4.3.2.6.3. What to cache	157
1.4.3.2.6.4. How to cache	157
1.4.3.2.7. Server side process (aka stored procedure)	158
1.4.3.2.7.1. Custom SQL functions	159
1.4.3.2.7.1.1. Implementing a function	159
1.4.3.2.7.1.2. Registering a function	160
1.4.3.2.7.1.3. Direct low-level SQLite3 registration	160
1.4.3.2.7.1.4. Class-driven registration	160
1.4.3.2.7.1.5. Custom class definition	161
1.4.3.2.7.2. Low-level Delphi stored procedure	163
1.4.3.2.7.3. External stored procedure	163
1.4.3.3. Client-Server services	163
1.4.3.3.1. Client-Server services via methods	163
1.4.3.3.1.1. My Server is rich	164
1.4.3.3.1.2. The Client is always right	165
1.4.3.3.2. Interface based services	167
1.4.3.3.2.1. Implemented features	167
1.4.3.3.2.2. How to make services	168
1.4.3.3.2.3. Defining a data contract	168
1.4.3.3.2.3.1. Define an interface	169
1.4.3.3.2.3.2. Available types for methods parameters	170

1.4.3.3.2.3.3. Custom JSON serialization of records	171
1.4.3.3.2.3.4. TInterfacedCollection kind of parameter	172
1.4.3.3.2.4. Server side	173
1.4.3.3.2.4.1. Implementing service contract	173
1.4.3.3.2.4.2. Set up the Server factory	174
1.4.3.3.2.4.3. Instances life time implementation	174
1.4.3.3.2.4.4. Using services on the Server side	177
1.4.3.3.2.5. Client side	178
1.4.3.3.2.5.1. Set up the Client factory	178
1.4.3.3.2.5.2. Using services on the Client side	178
1.4.3.3.2.6. Sample code	179
1.4.3.3.2.6.1. The shared contract	179
1.4.3.3.2.6.2. The server sample application	179
1.4.3.3.2.6.3. The client sample application	181
1.4.3.3.2.7. Implementation details	181
1.4.3.3.2.7.1. Involved classes	181
1.4.3.3.2.7.2. Security	182
1.4.3.3.2.7.3. Transmission content	184
1.4.3.3.2.7.4. Request format	184
1.4.3.3.2.7.5. Response format	185
1.4.3.3.2.7.6. Standard answer as JSON object	185
1.4.3.3.2.7.7. Custom returned content	187
1.4.3.3.2.8. Hosting services	188
1.4.3.3.2.8.1. Shared server	188
1.4.3.3.2.8.2. Two servers	189
1.4.3.3.2.8.3. Two instances on the same server	190
1.4.3.3.2.9. Comparison with WCF	191
<b>1.4.4. Security and Testing</b>	<b>194</b>
1.4.4.1. Security	194
1.4.4.1.1. Per-table access rights	194

1.4.4.1.2. SQL statements safety	194
1.4.4.1.3. Authentication	195
1.4.4.1.3.1. Principles	195
1.4.4.1.3.2. HTTP basic auth over HTTPS	195
1.4.4.1.3.3. Session via Cookies	196
1.4.4.1.3.4. Query Authentication	196
1.4.4.1.4. Framework authentication	196
1.4.4.1.4.1. Per-User authentication	197
1.4.4.1.4.2. Session handling	198
1.4.4.1.4.3. Client interactivity	199
1.4.4.1.4.4. URI signature	199
1.4.4.1.4.5. Authentication using AJAX	200
1.4.4.2. Testing	200
1.4.4.2.1. Thread-safety	200
1.4.4.2.2. Automated testing	201
1.4.4.2.2.1. Involved classes in Unitary testing	201
1.4.4.2.2.2. First steps in testing	202
1.4.4.2.2.3. Implemented tests	204
1.4.4.2.3. Logging	204
<b>1.4.5. Source code</b>	206
1.4.5.1. License	206
1.4.5.2. Availability	206
1.4.5.2.1. Obtaining the Source Code	207
1.4.5.2.2. Expected compilation platform	207
1.4.5.2.3. Note about sqlite3*.obj files	207
1.4.5.2.4. Folder layout	208
1.4.5.3. Installation	210
<b>1.4.6. SynCommons unit</b>	211
1.4.6.1. Unicode and UTF-8	211
1.4.6.2. Currency handling	212
1.4.6.3. Dynamic array wrapper	212

1.4.6.3.1. TList-like properties	213
1.4.6.3.2. Enhanced features	214
1.4.6.3.3. Capacity handling via an external Count	214
1.4.6.3.4. JSON serialization	215
1.4.6.3.4.1. TDynArray JSON features	215
1.4.6.3.4.2. Custom JSON serialization of dynamic arrays	216
1.4.6.3.5. Daily use	217
1.4.6.4. Enhanced logging	219
1.4.6.4.1. Using logging	219
1.4.6.4.2. Including symbol definitions	220
1.4.6.4.3. Exception handling	221
1.4.6.4.3.1. Intercepting exceptions	221
1.4.6.4.3.2. One patch to rule them all	222
1.4.6.4.4. Serialization	224
1.4.6.4.5. Family matters	224
1.4.6.4.6. Automated log archival	225
1.4.6.4.7. Log Viewer	225
1.4.6.4.7.1. Open log files	225
1.4.6.4.7.2. Log browser	226
1.4.6.4.7.3. Customer-side profiler	226
<b>1.4.7. Source code implementation</b>	227
1.4.7.1. mORMot Framework used Units	227
1.4.7.2. SynCommons unit	229
1.4.7.3. SynCrtSock unit	369
1.4.7.4. SynCrypto unit	383
1.4.7.5. SynDB unit	395
1.4.7.6. SynDBODBC unit	428
1.4.7.7. SynDBOracle unit	432
1.4.7.8. SynGdiPlus unit	438
1.4.7.9. SynLZ unit	445

1.4.7.10. SynLZO unit	447
1.4.7.11. SynOleDB unit	447
1.4.7.12. SynPdf unit	459
1.4.7.13. SynSelfTests unit	500
1.4.7.14. SynSQLite3 unit	502
1.4.7.15. SynTaskDialog unit	553
1.4.7.16. SynWinSock unit	559
1.4.7.17. SynZip unit	560
1.4.7.18. SQLite3 unit	567
1.4.7.19. SQLite3Commons unit	575
1.4.7.20. SQLite3HttpClient unit	731
1.4.7.21. SQLite3HttpServer unit	733
1.4.7.22. SQLite3i18n unit	736
1.4.7.23. SQLite3Pages unit	745
1.4.7.24. SQLite3SelfTests unit	760
1.4.7.25. SQLite3Service unit	761
1.4.7.26. SQLite3ToolBar unit	768
1.4.7.27. SQLite3UI unit	781
1.4.7.28. SQLite3UIEdit unit	790
1.4.7.29. SQLite3UILogin unit	793
1.4.7.30. SQLite3UIOptions unit	796
1.4.7.31. SQLite3UIQuery unit	798
<b>1.5. Main SynFile Demo</b>	799
1.5.1. SynFile application	799
1.5.2. General architecture	799
1.5.3. Database design	800
1.5.4. User Interface generation	804
1.5.4.1. Rendering	804
1.5.4.2. Enumeration types	807
1.5.4.3. ORM Registration	807



1.5.4.4. Report generation	808
1.5.4.5. Application i18n and L10n	811
1.5.4.5.1. Creating the reference file	811
1.5.4.5.2. Adding a new language	813
1.5.4.5.3. Language selection	813
1.5.4.5.4. Localization	814
<b>1.5.5. Source code implementation</b>	814
1.5.5.1. Main SynFile Demo used Units	814
1.5.5.2. FileClient unit	816
1.5.5.3. FileEdit unit	818
1.5.5.4. FileMain unit	820
1.5.5.5. FileServer unit	821
1.5.5.6. FileTables unit	822
<b>2. SWRS implications</b>	
<hr/>	
Software Architecture Design Reference Table	827
<b>2.1. Client Server JSON framework</b>	828
2.1.1. SWRS # DI-2.1.1	828
2.1.2. SWRS # DI-2.1.1.1	828
2.1.3. SWRS # DI-2.1.1.2.1	829
2.1.4. SWRS # DI-2.1.1.2.2	829
2.1.5. SWRS # DI-2.1.1.2.3	829
2.1.6. SWRS # DI-2.1.1.2.4	830
2.1.7. SWRS # DI-2.1.2	830
2.1.8. SWRS # DI-2.1.3	831
<b>2.2. SQLite3 engine</b>	832
2.2.1. SWRS # DI-2.2.1	832
2.2.2. SWRS # DI-2.2.2	833
<b>2.3. User interface</b>	833
2.3.1. SWRS # DI-2.3.1.1	833
2.3.2. SWRS # DI-2.3.1.2	834

**2.3.3. SWRS # DI-2.3.1.3**

834

**2.3.4. SWRS # DI-2.3.2**

835

## Pictures Reference Table

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

Pictures	Page
Client-Server implementation - Client side	142
Client-Server implementation - Server side	142
Client-Server implementation - Server side	146
Client-Server implementation - Server side with Virtual Tables	144
Client-Server implementation - Stand-Alone application	143
Client-Server implementation - Server side with "static" Virtual Tables	145
BATCH mode Client-Server latency	152
BATCH mode latency issue on external DB	155
HTTP/1.1 Client architecture	148
HTTP/1.1 Client RESTful classes	148
Client-Server RESTful classes	147
AuditTrail Record Layout	803
AuthGroup Record Layout	197
AuthUser Record Layout	197
Data Record Layout	802
Memo Record Layout	802
SafeData Record Layout	803
SafeMemo Record Layout	802
Design Inputs, FMEA and Risk Specifications	28
FTS3/FTS4 ORM classes	100
External Databases classes hierarchy	123
SynFile TSQLRecord classes hierarchy	801
HTTP Server classes hierarchy	150
Services implementation classes hierarchy	181
TSQLDataBaseSQLFunction classes hierarchy	161

<b>Pictures</b>	<b>Page</b>
TSQldbConnection classes hierarchy	116
TSQldbConnectionProperties classes hierarchy	116
TSQldbStatement classes hierarchy	116
Custom Virtual Tables records classes hierarchy	108
TSQlRestClient classes hierarchy	128
Virtual Tables classes hierarchy	104
Filtering and Validation classes hierarchy	86
Default filters and Validation classes hierarchy	86
TSynTest classes hierarchy	202
<i>FileClient</i> class hierarchy	817
<i>FileEdit</i> class hierarchy	819
<i>FileMain</i> class hierarchy	821
<i>FileServer</i> class hierarchy	822
<i>FileTables</i> class hierarchy	823
<i>SQLite3</i> class hierarchy	568
<i>SQLite3Commons</i> class hierarchy	577
<i>SQLite3HttpClient</i> class hierarchy	731
<i>SQLite3HttpServer</i> class hierarchy	734
<i>SQLite3i18n</i> class hierarchy	737
<i>SQLite3Pages</i> class hierarchy	746
<i>SQLite3Service</i> class hierarchy	762
<i>SQLite3ToolBar</i> class hierarchy	769
<i>SQLite3UI</i> class hierarchy	782
<i>SQLite3UIEdit</i> class hierarchy	791
<i>SQLite3UILogin</i> class hierarchy	794
<i>SQLite3UIOptions</i> class hierarchy	797
<i>SQLite3UIQuery</i> class hierarchy	798
<i>SynCommons</i> class hierarchy	231

<b>Pictures</b>	<b>Page</b>
<i>SynCrtSock</i> class hierarchy	370
<i>SynCrypto</i> class hierarchy	384
<i>SynDB</i> class hierarchy	395
<i>SynDBODBC</i> class hierarchy	429
<i>SynDBOracle</i> class hierarchy	433
<i>SynGdiPlus</i> class hierarchy	438
<i>SynOleDB</i> class hierarchy	448
<i>SynPdf</i> class hierarchy	461
<i>SynSelfTests</i> class hierarchy	501
<i>SynSQLite3</i> class hierarchy	503
<i>SynTaskDialog</i> class hierarchy	554
<i>SynZip</i> class hierarchy	560
CRUD caching in mORMot	156
N-Layered Domain-Oriented Architecture of mORMot	48
Alternate Domain-Oriented Architecture of mORMot	48
General mORMot architecture - Client / Server	50
General mORMot architecture - Stand-alone application	51
General mORMot architecture - Client Server implementation	52
Service Hosting on mORMot - shared server	189
Service Hosting on mORMot - two servers	190
Service Hosting on mORMot - one server, two instances	191
Model View Controller concept	44
Oracle Connectivity with SynDBOracle	119
Unit dependencies in the "Lib\SQLite3" directory	229
Unit dependencies in the "Lib" directory	228
Unit dependencies in the "Lib\SQLite3\Samples\MainDemo" directory	816
Unit dependencies in the "Lib\SQLite3" directory	815
Unit dependencies in the "Lib" directory	814

<b>Pictures</b>	<b>Page</b>
User Interface generated using TMS components	806
User Interface generated using VCL components	806

## 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

Source code File Names	Page
Lib\SQLite3\Samples\MainDemo\FileClient.pas	800, 808
Lib\SQLite3\Samples\MainDemo\FileEdit.pas	800
Lib\SQLite3\Samples\MainDemo\FileMain.pas	130, 812
Lib\SQLite3\Samples\MainDemo\FileServer.pas	800
Lib\SQLite3\Samples\MainDemo\FileTables.pas	800
Lib\SQLite3\SQLite3Commons.pas	104, 204
Lib\SQLite3\SQLite3i18n.pas	807
Lib\SQLite3\SQLite3Pages.pas	808
Lib\SQLite3\SQLite3UIEdit.pas	804
Lib\SynCommons.pas	201, 211
Lib\SynCrypto.pas	800
Lib\SynGdiPlus.pas	800, 810
Lib\SynSQLite3.pas	218

## Keywords Reference Table

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

.	
.msg	811
<b>6</b>	
64 bit	113, 120, 149, 207
<b>A</b>	
ACID	91, <u>94</u> , 102
AJAX	41, 42, 44, 45, 66, 80, 93, 102, 125, 126, 128, 150, 165, 167, 168, 169, 170, 185, 186, 187, 200, 207, 804
ARC	138
Array bind	96, 120, <u>154</u>
Atomic	111, 194
ATTACH DATABASE	109, 109
Authentication	92, 165, 168, 182, 184, 194, 195
<b>B</b>	
Backup	103
BATCH	70, 109, <u>151</u>
Benchmark	<u>93</u> , 155
BinToBase64WithMagic	59
BLOB	55, 65, 66, 67, 92, 98, 125, <u>128</u> , 166, 215, 800, 800
Business rules	42
<b>C</b>	
Cache	41, 49, <u>84</u> , 96, 126, 150, 201



Camel	88, <b>88</b> , 204, 807
Cardinality	<b>62</b> , 64
Client-Server	41, 42, 45, 47, 53, 70, 80, 83, 84, 92, 92, 109, 109, 127, 128, 129, 135, 135, 137, 137, <b>141</b> , 147, 151, 158, 195, 196, 226
Contract	163
CQRS	91
CreateAndFillPrepare	<b>58</b> , 82
CRUD	44, 57, 84, 87, 90, 111, 127, 144, 153, 156, 194, 197, 198
Currency	54, <b>56</b> , 67, 68, 114, 211, 212, 215

## D

DateTimeToSQL	59
DateToSQL	59
DMZ	188
Domain Values	90
Domain-Driven	41, 42, 44, 45, <b>47</b> , 65, 90, 177
DTO	90
Dynamic array	55, 56, 57, 60, 65, 65, 65, 66, 67, 78, 81, 81, 125, 159, 159, 211, 212, 224

## E

Enumerated	54, 88, 136, 220
Event Sourcing	193
Event-Sourcing	90

## F

Factory	137
Filtering	<b>85</b>
FTS	92, <b>99</b> , 99, 100, 101, 104

## G

Gateway	169, 193
General Public License	29, 206
GUID	131

## H

Has many through	<u>64</u> , 73
Has many	<u>64</u>
Hosting	42, 48, 174, <u>188</u>
HTTP	41, 42, 45, 80, 126, <u>127</u> , 128, 129, 129, 141, 147, <u>148</u> , <u>149</u> , 152, 154, 166, 195, 197, 208
HTTPS	195, <u>195</u> , 200

## I

I18n	41, 55, 88, 89, 210, 211, 807, 810, <u>811</u>
IntegerDynArrayContains	66, <u>68</u> , 159
Interface	42, 46, <u>130</u> , 134, 136, 137, 137, 137
ISO 8601	54, 56, 92, 92
Iso8601ToSQL	59

## J

JavaScript	45, 45, 80, 102, 125, 126, 195, <u>200</u> , 207
JOIN	<u>76</u>
JSON	41, 42, 45, 45, 55, 57, 58, 59, 65, 71, 71, 80, 81, 84, 93, 102, 104, 113, 118, 120, <u>125</u> , 125, 126, 126, 128, 150, 152, 163, 170, 171, 184, 204, 211, 213, 215, 224, 800

## L

L10n	<u>811</u>
Lazy Loading	<u>64</u>
Lesser General Public License	206

License	135, <a href="#">206</a>
---------	--------------------------

Log	116, 211, <a href="#">219</a> , 221, 224, 804
-----	---

## M

Map/reduce	155
------------	-----

Master/Detail	<a href="#">62</a> , 65, 66, 78, 81, 81
---------------	---

Mozilla Public License	206
------------------------	-----

MS SQL	41, 45, 50, 112, 113, 115, <a href="#">118</a> , 121, 122, 124
--------	--

MVC	41, <a href="#">44</a> , 84, 85, 88, 804
-----	--

## O

ODBC	41, 42, 50, 93, 96, 112, 114, <a href="#">118</a> , 209
------	---

OleDb	41, 42, 50, 93, 96, 112, 114, <a href="#">118</a> , 209
-------	---

One to many	62, <a href="#">62</a>
-------------	------------------------

One to one	62, <a href="#">62</a>
------------	------------------------

OOP	79, 113, 133, 135, 801
-----	------------------------

Oracle	41, 42, 45, 50, 93, 96, 112, 114, 118, <a href="#">119</a> , 123, 209
--------	---

ORM	41, 42, 45, <a href="#">46</a> , 46, <a href="#">53</a> , 54, 58, 61, 64, 65, 65, 73, 77, 77, 80, 81, 81, 81, 82, 83, 84, 88, 89, 96, 97, 98, 101, 101, 104, 106, 108, 108, 109, 110, 112, 120, 134, 136, 137, 158, 163, 163, 166, 167, 168, 186, 189, 200, 204, 208, 209, 212, 213, 216, 799, 800, 800, 804, 808
-----	---

## P

Pdf	42, 112, 209, 800, 808
-----	------------------------

Prepared	<a href="#">60</a> , 66, 83, 84, 96, 113, 218
----------	---

Published method	158, <a href="#">163</a> , 201
------------------	--------------------------------

Published properties	53, <a href="#">54</a> , 62, 65, 73, 80, 81, 92, 98, 105, 110, 121, 128, 213, 800, 801
----------------------	--

## Q

Query Authentication	194, 195, <a href="#">196</a> , 196, <a href="#">199</a>
----------------------	--

Quotes	60, <b>60</b> , 97
--------	--------------------

## R

RawUTF8	54, <b>55</b> , 60, 68, 70, 76, 83, 97, 101, 114, 170, 211, 215, 218
RDBMS	62, 65
Record	170, 171, 216
Reference-counted	132
RegisterCustomJSONSerializ er	171, <b>215</b> , 216
RegisterCustomSerializer	55, 66, <b>71</b> , 170, 224
Report	42
Resourcestring	810, 811
REST	41, 42, 44, 45, 45, 53, 57, 78, 80, 83, 84, 90, 109, 110, 112, 124, 125, 127, <b>127</b> , 128, 128, 129, 129, 130, 141, 147, 150, 151, 156, 158, 163, 167, 191, 195, 196, 200, 204, 800, 804
RTREE	92, 98, 104
RTTI	45, 53, 57, 57, 60, <b>87</b> , 212, 213, 224, 800, 804, 807, 807, 811

## S

Security	41, 42, 88, 92, 167, 182, 187, 188, <b>194</b> , 204
Serialization	42, 46, 68, <b>71</b> , 125, 170, <b>171</b> , 187, 213, <b>215</b>
Server time stamp	79
Service	44, 45, 45, 68, 78, 83, 127, 147, <b>163</b> , 800
Session	41, 70, 158, 165, 175, 194, 194, 195, <b>196</b> , 196, <b>198</b> , 199, 204
SftBlob	59
Sharding	64, <b>65</b>
Shared nothing architecture	<b>65</b>
SOA	41, 42, 45, <b>45</b> , 90, 137, 137, 138, 167, 168, 191, 195, 216
SOAP	133, 167, 169, 191, <b>193</b>
SOLID	48, <b>133</b> , 169

SQL function	<u>68</u> , 68, 68, 84, 98, 102, 158, <u>159</u>
SQL	41, 42, 45, 46, 53, 53, 58, 60, 61, 67, 69, 80, 80, 80, 81, 83, 84, 88, 88, 92, 92, 92, 96, 98, 101, 102, 103, 126, 126, 126, 204, 226
SQLite3	41, 45, 50, 53, 53, 55, 60, 68, 85, 88, 93, <u>96</u> , 99, 102, 103, 105, 107, 109, 110, 111, 112, 114, 114, 120, 125, 126, 150, 158, 159, 194, 201, 204, 205, 207, 209, 209, 211, 215
Stateless	44, 44, <u>129</u> , 129, 130, 130, 137, 196, 201
Statement	84
Static	104, <u>109</u> , 109, 110, 145
Stored procedure	68, 80, 84, <u>158</u>
Strong type	<u>80</u> , 88, 88, 90, 194
SynDB	55, 93, 96, <u>113</u> , 114, 118, 119, 120, 122, 134, 204, 209
SynDBExplorer	113, 204
SynLZ	104, 150, 154, 208

## T

TCollection	55, <u>65</u> , 65, 66, <u>68</u> , 81, 81, 169, 170, 213, 224
TCreateTime	<u>55</u> , 59, 79, 92
TDateTime	54, <u>56</u> , 59
TDynArrayHashed	113
Test	41, 44, 66, 68, 69, 74, 101, 136, 152, <u>201</u> , 205, 209, 210, 210, 211, 219
Text Search	42
Thread-safe	92, 194, <u>200</u>
Tier	41, 41, 42, 44, <u>45</u>
TInterfacedCollection	169, 170, 170
TModTime	<u>55</u> , 59, 79, 92
TObject	55, 66, 71, 170
TPersistent	55, <u>65</u> , 66, <u>68</u> , 81, 81, 165, 170, 224, 801
Transaction	<u>69</u> , 74, 75, 111, 127, 129, 152, 193, <u>199</u>

TServiceCustomAnswer	169, 170, 187
TSQLModel	44, 53, 81, 81, <b>84</b> , 87, 109, 110, 150, 151, 197, 800, 803
TSQLRecord	44, <b>53</b> , 54, 57, 59, 60, 62, 64, 65, 65, 66, 68, 68, 70, 73, 77, 78, 79, 81, 81, 81, 84, 86, 87, 90, 90, 98, 101, 105, 108, 109, 110, 112, 121, 129, 134, 135, 163, 165, 170, 186, 211, 215, 224, 800, 801, 803, 808
TSQLRecordMany	55, 64, 64, 65, 68, 73, 76, 81
TSQLRecordMappedAutoID	90
TSQLRecordMappedForcedID	90
TSQLRecordVirtual	90
TSQLRest	55, 60, 80, 85, 96, 128, 128, 147, 803
TSQLRestClientDB	110, 142, 147, 194
TSQLRestServerDB	98, 110, 120, 126, 142, 147, 160, 163, 164, 174, 180, 201
TSQLRestServerFullMemory	93, 147, 164, 174, 180
TSQLRestServerRemoteDB	147, 190
TSQLRestServerStaticInMemory	85, 90, 91, 104, 106, <b>109</b> , 109, 110, 111, 201
TSQLTableJSON	58, 59, 60, 129
TSQLVirtualTableBinary	104, 104, 109
TSQLVirtualTableJSON	104, 104, 109
TStrings	55, 65, 66, 81, 81
TTimeLog	<b>55</b> , 59, 92, 810

## U

UTF-8	54, 55, 92, 101, 125, 202, 211, <b>211</b> , 215
-------	--

## V

Validation	<b>85</b>
Virtual Table	42, 50, 98, 99, <b>103</b> , 104, 105, 108, 109, 110, 112, 112, 122, 143
VirtualTableExternalRegistrar	90, <b>112</b> , 122

## W

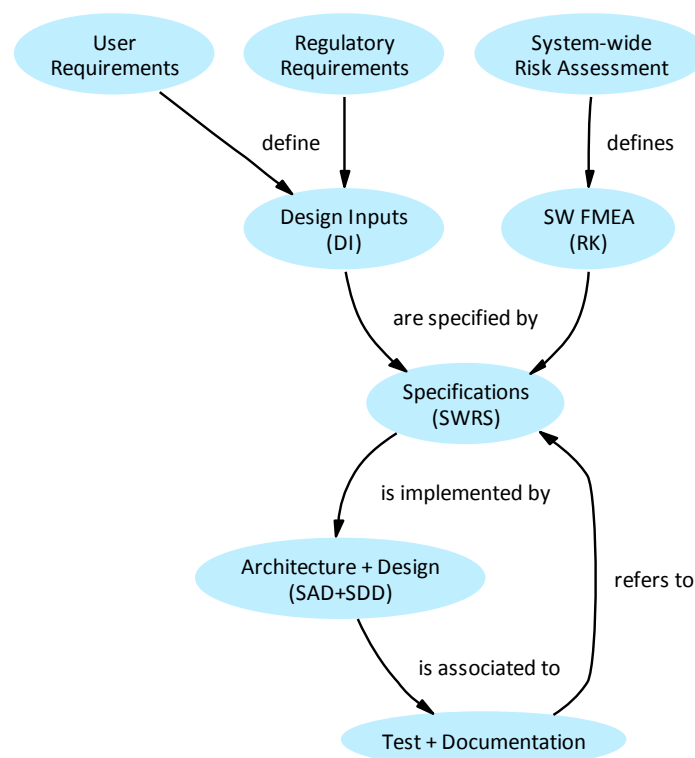
WCF	41, 163, <u><a href="#">191</a></u>
Weak pointers	<u><a href="#">138</a></u>
WideString	170

## Z

Zeroing Weak pointers	<u><a href="#">138</a></u>
-----------------------	----------------------------

# Introduction

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



*Design Inputs, FMEA and Risk Specifications*

## Purpose

This *Software Architecture Design (SAD)* document applies to the 1.17 release of the *Synopse mORMot Framework* library.

It summarizes the implications of every software features detailed in the *Software Design Document (SDD)* document.

## Document layout

In a first part, this document presents the global Software architecture involved in this



implementation, i.e.:

- mORMot Framework (page 53)
- Main SynFile Demo (page 799)

A second part then describes every *Software Requirements Specifications* (SWRS) document item, according to the *Design Input Product Specifications* (DI) document main sections:

- Client Server JSON framework (page 828)
- SQLite3 engine (page 832)
- User interface (page 833)

## First part: global Software architecture

All the source code architecture of the library is deeply analyzed. After a global introduction, 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 main sections of this architecture description are the following:

- Architecture principles - see below (page 44);
- General design - see below (page 50);
- ORM and MVC - see below (page 53);
- Database layer - see below (page 92);
- Client-Server - see below (page 125);
- Security and testing - see below (page 194);
- Source code - see below (page 206);
- The SynCommons unit - see below (page 211);
- Followed by the per-unit description of every defined class or type;
- *SynFile* main demo - see below (page 799).

## Second part: SWRS implications

For each SWRS item, links are made to the units sections of the first part, referring directly to the unit, class or function involved with the *Software Design Document* (SDD) document.

## Responsibilities

- Synapse will try to correct any identified issue;
- The Open Source community will create tickets in a public Tracker web site located at <http://synapse.info/fossil..> ;
- Synapse work on the framework is distributed without any warranty, according to the chosen license terms - see below (page 206);
- 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 feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

### 1. Source Code.

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.

#### 4. Conveying Verbatim Copies.

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.

#### 9. Acceptance Not Required for Having Copies.

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.

#### 12. No Surrender of Others' Freedom.

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. Global Architecture

---

## 1.1. Synopse mORMot Framework Overview

*Synopse mORMot* is a Client-Server ORM and Service Oriented Architecture framework (SOA) for Delphi 6 up to XE2.

It 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 47):

- *Presentation layer* featuring MVC UI generation with i18n and reporting for rich Delphi clients, 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 (including HTTP/1.1);
- *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, or dedicated classes for entities or aggregates;
- *Data persistence infrastructure layer* with ORM persistence over Oracle, MS SQL, OleDb, ODBC with a powerful SQLite3 kernel, and direct SQL access if needed;
- *Cross-Cutting infrastructure layers* for handling data filtering and validation, security, session, cache, logging and testing (framework uses test-driven approach).

If you do not know some of those concepts, don't worry: this document will detail them - see below (page 44).

The main two features of *mORMot* shine at the application layer:

- Client-Server *ORM*: objects persistence and remote access;
- Client-Server *Services*: remote call of high-level data process.

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, ready to implement *Domain-Driven* solutions.

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 or AJAX clients.

No dependency is needed at the client level (no DB driver, nor third-party runtime): it is able to connect via standard HTTP, 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. Speed and scalability has been implemented from the ground up - see below (page 93): a single server is able to handle a lot of clients, and our rich SOA architecture is able to implement both vertical and horizontal scalable hosting.

In short, with *mORMot*, your ROI is maximized.

### 1.1.1. 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 (not via external scripting or such) 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;
- Fastest available HTTP server using *http.sys* kernel-mode server - but may communicate via named pipes, GDI messages or in-process as lighter alternatives;
- Using *SQLite3* as its kernel, but able to connect to any other database (via OleDb / ODBC or direct client library access e.g. for Oracle) - the SynDB classes are self-sufficient, and do not depend on the Delphi DB unit nor any third-party (so even the Delphi Starter edition is enough);
- 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;
- 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 93);
- More than 1000 pages of documentation;
- Delphi and AJAX clients can share the same server;
- Full source code provided - so you can enhance it to fulfill any need;
- Works from Delphi 6 up to XE2, truly Unicode (uses UTF-8 encoding in its kernel, just like JSON), with any version of Delphi (no need to upgrade your IDE).

### 1.1.2. mORMot

Why is this framework named *mORMot*?

- Because its initial identifier was "*Synapse 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 even if they eat greens, they use to fight at Spring;
- Because it may be an acronym for "Manage Object Relational Mapping Over Tables", or whatever you may think of...

## 1.2. Architecture principles

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

- *Model-View Controller* - see below (page 44);
- *Multi-tier architecture* - see below (page 45);
- *Unit testing* - see below (page 200);
- *Object-relational mapping* - see below (page 46);
- *Service-oriented architecture* - see below (page 45);
- *Stateless CRUD/REST* - see below (page 127).

All those points render possible any project implementation, up to complex Domain-Driven design - see below (page 47).

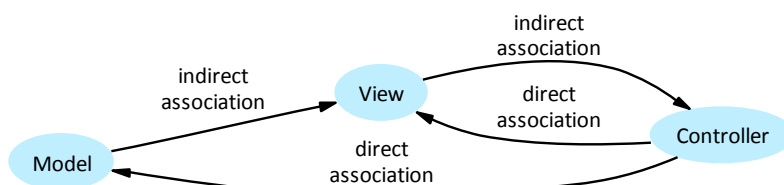
### 1.2.1. Model-View-Controller

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 129).

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.



*Model View Controller concept*

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 *view* is currently only all the User-Interface part of the framework, which is mostly auto-generated from code. It will use the *model* as reference for rendering the data. AJAX clients can also be used -



RESTful and JSON will make it easy.

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 163).

### 1.2.2. 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 middle-ware 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.

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

The *Synapse 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 external databases (as *MS SQL Server* or *Oracle*) - see below (page 112);
- **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 47) - if your project reaches some level of complexity;
- **Presentation Tier** is either a Delphi Client, either 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).

### 1.2.3. Service-oriented architecture

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.

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.

For more details about SOA, see [http://en.wikipedia.org/wiki/Service-oriented\\_architecture..](http://en.wikipedia.org/wiki/Service-oriented_architecture..)

SOA and ORM - see below (page 46) - 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;

- SOA will provide a more advanced way of handling the business logic: with custom parameters and data types, it's 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 45). 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.

#### 1.2.4. Object-relational mapping

Before defining what is an ORM, let's face one drawback of using a database via an object-oriented language like *Delphi* is that you must have your objects interact with the database. Some implementation schemes are possible, in which are some pros and cons. 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.

Scheme	Pros	Cons
Use DB views and tables, with GUI components	<ul style="list-style-type: none"> <li>- SQL is a powerful language</li> <li>- Can use high-level DB tools (UML) and RAD approach</li> </ul>	<ul style="list-style-type: none"> <li>- Business logic can't be elaborated without stored procedures</li> <li>- SQL code and stored procedures will bind you to a DB engine</li> <li>- Poor Client interaction</li> <li>- Reporting must call the DB directly</li> <li>- No Multi-tier architecture</li> </ul>
Map DB tables or views with Delphi classes	<ul style="list-style-type: none"> <li>- Can use elaborated business logic, in Delphi</li> <li>- Separation from UI and data</li> </ul>	<ul style="list-style-type: none"> <li>- SQL code must be coded by hand and synchronized with the classes</li> <li>- Code tends to be duplicated</li> <li>- SQL code could bind you to a DB engine</li> <li>- Reports can be made from code or via DB related tools</li> <li>- Difficult to implement true Multi-tier architecture</li> </ul>
Use a Database ORM	<ul style="list-style-type: none"> <li>- Can use very elaborated business logic, in Delphi</li> <li>- SQL code is generated (in most cases) by the ORM</li> <li>- ORM will adapt the generated SQL to the DB engine</li> </ul>	<ul style="list-style-type: none"> <li>- More abstraction needed at design time (no RAD approach)</li> <li>- In some cases, could lead to retrieve more data from DB than needed</li> <li>- Not yet a true Multi-tier architecture, because ORM is for DB access only and business logic will need to create separated classes</li> </ul>

---

	- Can use very elaborated business logic, in Delphi	
	- SQL code is generated (in most cases) by the ORM	
	- ORM will adapt the generated SQL to the DB engine	
Use a Client-Server ORM	- Services will allow to retrieve or process only needed data	- More abstraction needed at design time (no RAD approach)
	- 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	
	- Complete Multi-tier architecture	

---

Of course, you'll find out that this framework implements a Client-Server ORM.

### 1.2.5. Domain-Driven design

The "official" definition of Domain-driven design is supplied at <http://domaindrivendesign.org..> :

*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.*

It can be implemented in a kind of *Multi-tier architecture* (page 45). In this case, we are talking about *N-Layered Domain-Oriented Architecture*. It involves a common representation splitting the *Logic Tier* into two layers, i.e. *Application layer* and *Domain Model layer*. Of course, this particular layered 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.

It could therefore be presented as in the following model:

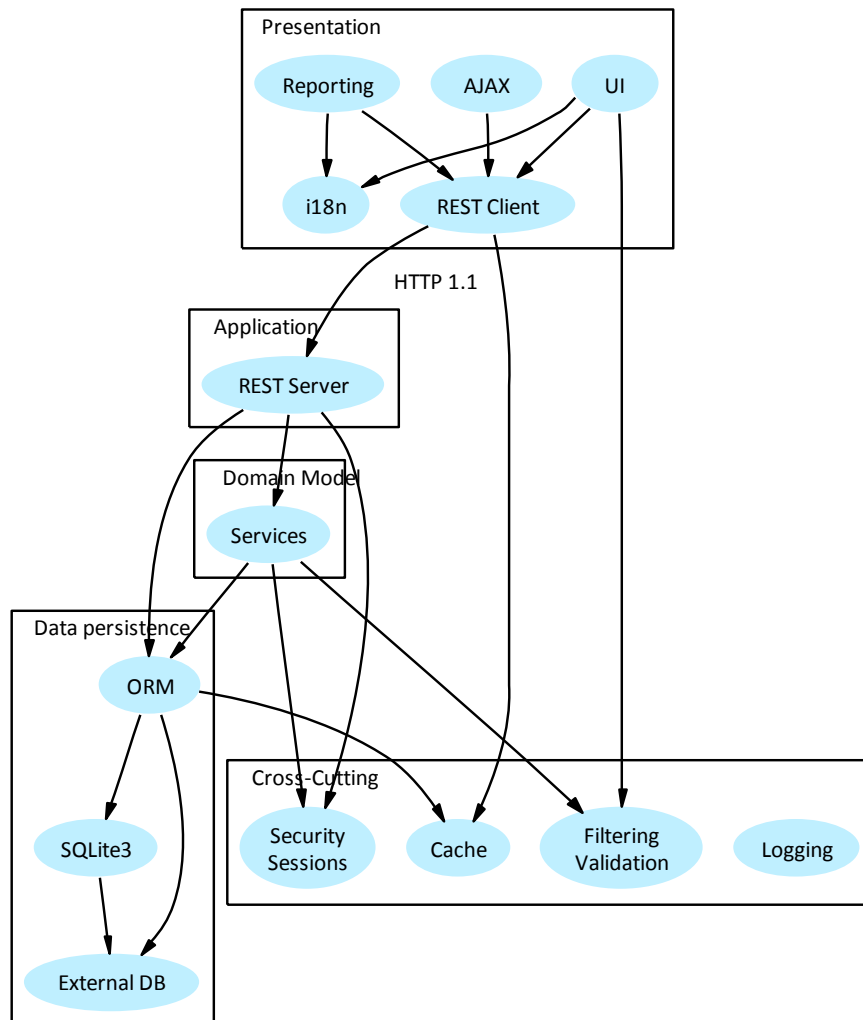
---

Layer	Description
Presentation	MVC UI generation and reporting
Application	Services and high-level adapters
Domain Model	Where business logic remains
Data persistence	ORM and external services
Cross-Cutting	Horizontal aspects shared by other layers

---

In fact, this design matches perfectly the RESTful SOA approach of the *Synapse mORMot framework*.

See the following diagram:

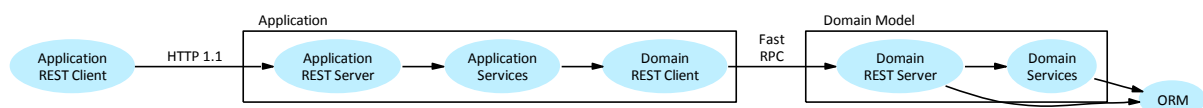


*N-Layered Domain-Oriented Architecture of mORMot*

In fact, since a *Service-oriented architecture* (page 45) 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:



*Alternate Domain-Oriented Architecture of mORMot*

Due to the SOLID design of mORMot - see below (page 133) - 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.

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...

With *mORMot*, your software solution will never be stucked in a dead-end. You'll be able to always adapt to your customers need, and maximize your ROI.

## 1.3. General design

### 1.3.1. SQLite3-powered, not SQLite3-limited

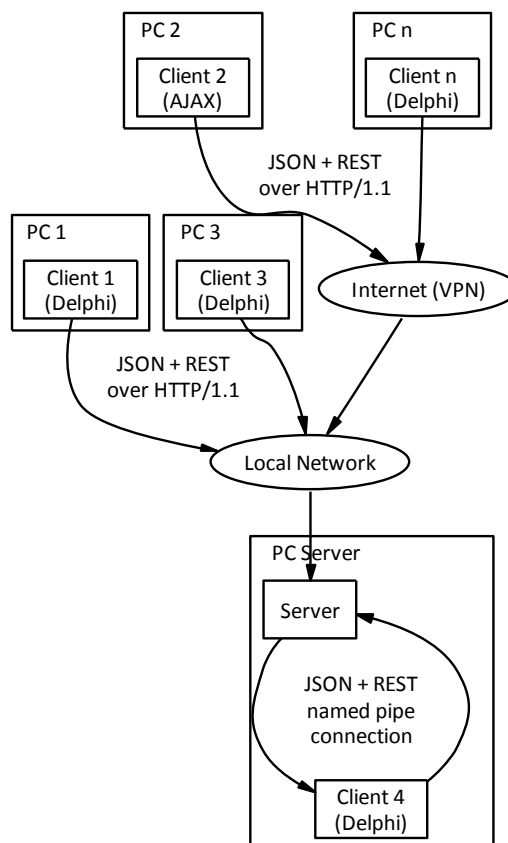
The core database of this framework uses the *SQLite3* library, which is a Free, Secure, Zero-Configuration, Server-less, Single Stable Cross-Platform Database File database engine.

As stated below, you can use any other database access layer, if you wish. A fast in-memory engine is included, and can be used instead or together with the *SQLite3* engine. Since revision 1.15 of the framework you may be able to access any remote database, and use one or more *OleDB*, *ODBC* or *Oracle* connections to store your precious ORM objects. For instance, MS SQL server can be used via *OleDB*. *SQLite3* will be used as the main SQL engine, able to JOIN all those tables, thanks to its Virtual Table unique feature.

### 1.3.2. Client-Server ORM/SOA architecture

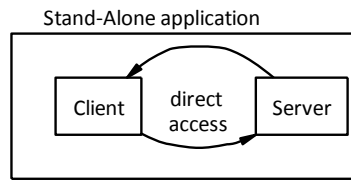
The *Synapse mORMot Framework* implements a Client-Server ORM-based architecture, trying to follow all best-practice patterns just introduced (MVC, n-Tier, SOA).

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



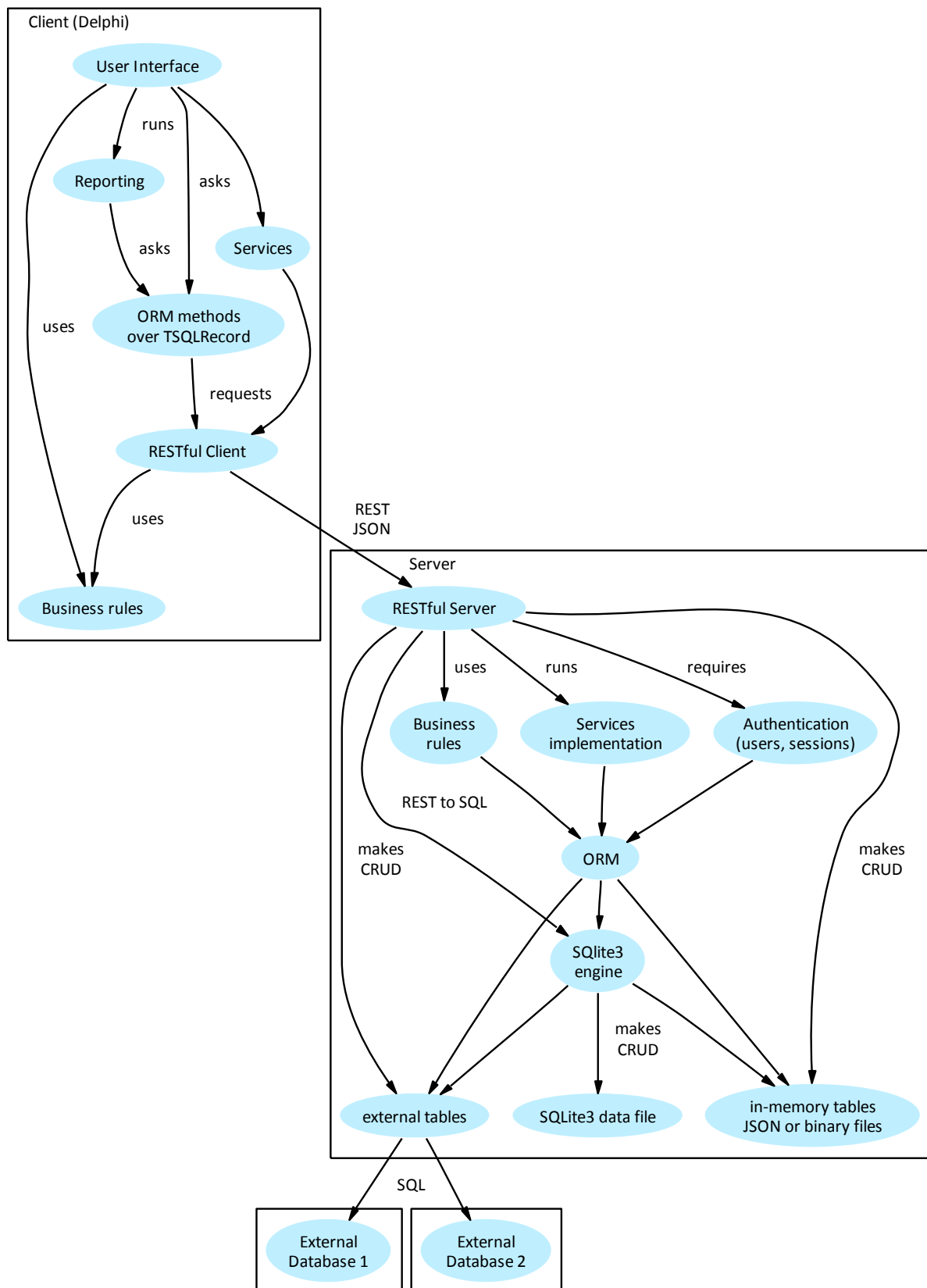
General mORMot architecture - Client / Server

Or the application can be stand-alone:



*General mORMot architecture - Stand-alone application*

A more detailed design may be summarized as in the following diagram:





The following pages will detail and explain how this framework implements this architecture.

## 1.4. mORMot Framework

### 1.4.1. Object-Relational Mapping

In order to implement *Object-relational mapping* (page 46) in our framework, generic access to the data is implemented by defining high-level objects as *Delphi* classes, descendant from the *TSQLRecord* class.

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

- Map the Database tables;
- Define business logic in *Delphi* code;
- Auto-generate the User Interface (grids, edition screens and menus).

This allows to truly implement a Multi-tier architecture - see *Multi-tier architecture* (page 45).

All published properties of the *TSQLRecord* descendant classes are then accessed via RTTI in a Client-Server RESTful architecture. Following the three previous purposes, these properties will be used:

- To store and retrieve data from the *SQLite3* database engine, by the framework - for most common usage, the coder doesn't have to write SQL queries: they are all created on the fly by the *Object-relational mapping* (ORM) framework;
- To have business logic objects accessible for both the Client and Server side, in a RESTful approach;
- 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.

#### 1.4.1.1. TSQLRecord fields definition

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

```
/// 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 *SQLite3* database engine. 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):

Delphi	SQLite3	Remarks
byte	INTEGER	
word	INTEGER	
integer	INTEGER	
cardinal	N/A	You should use <i>Int64</i> instead
Int64	INTEGER	
boolean	INTEGER	0 is false, anything else is true
enumeration	INTEGER	store the ordinal value of the enumerated item(i.e. starting at 0 for the first element)
set	INTEGER	each bit corresponding to an enumerated item (therefore a set of up to 64 elements can be stored in such a field)
single	FLOAT	
double	FLOAT	
extended	FLOAT	stored as <i>double</i> (precision lost)
currency	FLOAT	safely converted to/from currency type with fixed decimals, without rounding error
RawUTF8	TEXT	this is the <b>preferred</b> field type for storing some textual content in the ORM
WinAnsiString	TEXT	<i>WinAnsi</i> char-set (code page 1252) in Delphi
RawUnicode	TEXT	<i>UCS2</i> char-set in Delphi, as <i>AnsiString</i>
TDateTime	TEXT	ISO 8601 encoded date time
TTimeLog	INTEGER	as proprietary fast <i>Int64</i> date time
TModTime	INTEGER	the server date time will be stored when a record is modified (as proprietary fast <i>Int64</i> )
TCreateTime	INTEGER	the server date time will be stored when a record is created (as proprietary fast <i>Int64</i> )
TSQLRecord	INTEGER	RowID pointing to another record (warning: the field value contains <i>pointer(RowID)</i> , not a valid object instance - the record content must be retrieved via its ID using a <i>PtrInt(Field)</i> typecast or the <i>Field.ID</i> method)

TSQLRecordMany	nothing	data is stored in a separate <i>pivot</i> table; this is a particular case of TSQLRecord: it won't contain pointer (RowID), but an instance)
TRecordReference	INTEGER	store both ID and TSQLRecord type in a RecordRef-like value (use e.g. TSQLRest. Retrieve(Reference) to get a record content)
TPersistent	TEXT	JSON object (ObjectToJSON)
TCollection	TEXT	JSON array of objects (ObjectToJSON)
TStrings	TEXT	JSON array of string (ObjectToJSON)
TRawUTF8List	TEXT	JSON array of string (ObjectToJSON)
any TObject	TEXT	See TJSONSerializer.RegisterCustomSerializer
TSQLRawBlob	BLOB	
<i>dynamic arrays</i>	BLOB	in the TDynArray.SaveTo binary format

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

- If the property is marked as stored `false`, it will be created as UNIQUE in the database (i.e. an index will be created and unicity 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 WinAnsiString / RawUTF8 field of an "external" class - i.e. a TEXT field stored in a remote SynDB-based database - see below (page 112), the index number will be used to define the maximum character size of this field, when creating the corresponding column in the database (SQLite3 does not have any such size limit).

#### 1.4.1.1.1. Text fields

Note that WideString, shortstring, UnicodeString (i.e. Delphi 2009/2010/XE/XE2 generic string), and indexed properties are not handled yet (use faster RawUnicodeString instead of WideString or UnicodeString). In fact, the generic string type is handled (as UnicodeString under Delphi 2009/2010/XE/XE2), but you may loose 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. All business process should be made using 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 SQLite3i18n.pas or StringToUTF8 / UTF8ToString which will handle proper char-set conversion according to the current i18n settings.

For additional information about UTF-8 handling in the framework, see below (page 211).

#### 1.4.1.1.2. Date and time fields

For TTimeLog / TModTime / TCreateTime, the proprietary fast Int64 date time format will map the

Iso8601 record type, as defined in SynCommons:

- 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..38 bits will map years.

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

Note that since TTimeLog type is bit-oriented, you can't just use *add* or *subtract* two TTimeLog 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:

```
function TSQLRest.GetServerTimeStamp: TTimeLog;
begin
  PISO8601(@result)^.From(Now+fServerTimeStampOffset);
end;

procedure TSQLRest.SetServerTimeStamp(const Value: TTimeLog);
begin
  fServerTimeStampOffset := PISO8601(@Value)^.ToDateTime-Now;
end;
```

But if you simply want to *compare* TTimeLog kind of date/time, it is safe to directly compare their Int64 underlying value.

Due to compiler limitation in older versions of Delphi, direct typecast of a TTimeLog or Int64 variable into a Iso8601 record (as with Iso8601(aTimeLog).ToDateTime) could create an internal compiler error. In order to circumvent this bug, you would have to use a pointer typecast, e.g. as in PISO8601(@Value).ToDateTime above.

#### 1.4.1.1.3. 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 in the set.

#### 1.4.1.1.4. Floating point and Currency fields

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

In fact, double is the native type handled by most database providers (when it comes to money, a dedicated type is worth the cost in a "rich man's world") - 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. 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 below (page 212) - 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. Rounding issues are a nightmare to track - it sounds safe to have a framework handling natively a currency type from the ground up.

#### 1.4.1.1.5. Record fields

Published properties of *records* are handled in our code, but Delphi doesn't create the corresponding RTTI for such properties so it won't work.

You could use a *dynamic array* with only one element, in order to handle records within your TSQLRecord class definition - see below (page 67).

#### 1.4.1.2. Working with Objects

To access a particular record, the following code can be used to handle CRUD statement (*Add/Update/Delete/Retrieve*), following the RESTful pattern - see below (page 127):

```
var Baby: TSQLBaby;  
    ID: integer;  
begin  
    // create a new record, since Smith, Jr was just born  
    Baby := TSQLBaby.Create;  
    try  
        Baby.Name := 'Smith';  
        Baby.Address := 'New York City';  
        Baby.BirthDate := Now;  
        Baby.Sex := sMale;  
        ID := Client.Add(Baby);  
    finally  
        Baby.Free;  
    end;  
    // update record data  
    Baby := TSQLBaby.Create(Client, 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, just accessing objects via high-level methods. This is the magic of ORM.

### 1.4.1.3. Queries

#### 1.4.1.3.1. 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':

```
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 \*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 it has been called during a FillPrepare / CreateAndFillPrepare process, and only the retrieved filled will be updated on the server side.

#### 1.4.1.3.2. 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..) 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 Iso8601ToSQL() functions, as such:

```
aRec.CreateAndFillPrepare(Client, 'Datum=?', [DateToSQL(EncodeDate(2012,5,4))]);
aRec.CreateAndFillPrepare(Client, 'Datum<=?', [DateTimeToSQL(Now)]);
aRec.CreateAndFillPrepare(Client, 'Datum<=?', [Iso8601ToSQL(Iso8601Now)]);
```

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

Any sftBlob property should better be handled separately, via dedicated RetrieveBlob and UpdateBlob method calls, if the data is expected to be 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 such a query.

Note that there was a *breaking change* about the TSQLRecord.Create / FillPrepare / CreateAndFillPrepare and TSQLRest.OneFieldValue / MultiFieldValues methods: for historical reasons, they expected parameters to be marked as % in the SQL WHERE clause, and inlined via :(...): as stated below (page 60) - 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 112). Both TSQLRestClient.EngineExecuteFmt / ListFmt methods are not affected by this change, since they are just wrappers to the FormatUTF8() function.

#### 1.4.1.3.3. Introducing TSQLTableJSON

As we stated above, \*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:

```
procedure WriteBabiesStartingWith(const Letters: RawUTF8; Sex: TSex);
var aList: TSQLTableJSON;
    Row: integer;
begin
  aList := Client.MultiFieldValues(TSQLBaby, 'ID, BirthDate',
    'Name LIKE ? AND Sex = ?', [Letters+'%', ord(Sex)]);
  if aList=nil then
    raise Exception.Create('Impossible to retrieve data from Server');
  try
    for Row := 1 to aList.RowCount do
      writeln('ID=', aList.GetAsInteger(Row, 0), ' BirthDate=', aList.Get(Row, 1));
    finally
      aList.Free;
    end;
end;
```

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.

See also the following methods of `TSQLRest`: `OneFieldValue`, `OneFieldValues`, `MultiFieldValue`, `MultiFieldValues` which are able to retrieve either a `TSQLTableJSON`, either 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...

#### 1.4.1.3.4. 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:

```
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:

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

or

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

or

```
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:

```
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:

```
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 bound the parameters before execution (in our case, text A and integer 1), reusing any previous matching prepared SQL statement. See below (page 96) for more details about this mechanism.

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

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

or

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

which will produce the same as:

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

So that the following SQL statement will be executed:

```
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':

```
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 would have to be parsed and optimized only once.

The second query method, i.e.

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

is the same as this code:

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

or

```
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 :(:): inlining 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.

#### 1.4.1.4. 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*.

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

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:

```
TSQlMyFileInfo = class(TSQLRecord)
private
  FMyFileDate: TDateTime;
  FMyFileSize: Int64;
published
  property MyFileDate: TDateTime read FMyFileDate write FMyFileDate;
  property MyFileSize: Int64 read FMyFileSize write FMyFileSize;
end;

TSQlMyFile = class(TSQLRecord)
private
  FSecondOne: TSQlMyFileInfo;
  FFirstOne: TSQlMyFileInfo;
  FMyFileName: RawUTF8;
published
  property MyFileName: RawUTF8 read FMyFileName write FMyFileName;
  property FirstOne: TSQlMyFileInfo read FFirstOne write FFirstOne;
  property SecondOne: TSQlMyFileInfo read FSecondOne write FSecondOne;
end;
```

As stated by *TSQLRecord fields definition* (page 53), 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 do not use directly such published properties, like a regular class instance: you'll have an access violation.

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

```
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
  MyDataBase.Add(MyFile);
finally
  MyFile.Free;
  Two.Free;
  One.Free;
end;
end;
```

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

```
var One: TSQLMyFileInfo;
    MyFile: TSQLMyFile;
begin
  MyFile := TSQLMyFile.Create(Client,aMyFileID);
  try
    // here MyFile.FirstOne.MyFileDate will trigger an access violation
    One := TSQLMyFileInfo.Create(Client,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:

```
with TSQLMyFileInfo.Create(Client,MyFile.FirstOne) do
  try
    // here you can access MyFileDate or MyFileSize
  finally
    Free;
  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.

#### 1.4.1.4.2. "Has many" and "has many through"

As [http://en.wikipedia.org/wiki/Many-to-many\\_\(data\\_model\)](http://en.wikipedia.org/wiki/Many-to-many_(data_model)).. wrote:

*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.

#### **1.4.1.4.2.1. Shared nothing architecture (or sharding)**

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 47), 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.

Let us quote what *wikipedia* states at [http://en.wikipedia.org/wiki/Shared\\_nothing\\_architecture..](http://en.wikipedia.org/wiki/Shared_nothing_architecture..)

*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 53), 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.

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 our *BigTable* component are good candidate for light speed data process. Even *SQLite* could scale very well in most cases.

Storing detailed data in BLOB or in TEXT as JSON could first sounds 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 such tries 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 flat scheme.

Finally, this implementation pattern fits much better with a Domain-Driven Design.

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.

#### **1.4.1.4.2.1.1. Arrays, TPersistent, TCollection, TMyClass**

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 content, the serialized using the *TDynArray* wrapper. Therefore, only Delphi clients would be

able to use this field content: you'll lose 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 67)). Those functions are available from AJAX queries.

*TPersistent*, *TStrings* and *TCollection* 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). This format contains only valid JSON arrays or objects: so it could be unserialized 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.

#### 1.4.1.4.2.1.2. Dynamic arrays fields

#### 1.4.1.4.2.1.3. 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:

```
TFV = packed record
  Major, Minor, Release, Build: integer;
  Main, Detailed: string;
end;
TFVs = array of TFV;
TSQLRecordPeopleArray = class(TSQLRecordPeople)
private
  fInts: TIntegerDynArray;
  fCurrency: TCurrencyDynArray;
  fFileVersion: TFVs;
  fUTF8: RawUTF8;
published
  property UTF8: RawUTF8 read fUTF8 write fUTF8;
  property Ints: TIntegerDynArray index 1 read fInts write fInts;
  property Currency: TCurrencyDynArray index 2 read fCurrency write fCurrency;
  property FileVersion: TFVs index 3 read fFileVersion write fFileVersion;
end;
```

This *TSQLRecordPeopleArray* class inherits from *TSQLRecordPeople*, that is it will add UTF8, Ints, Currency and FileVersion fields to this root class.

Some content is added to the *PeopleArray* table, with the following code:

```
var V: TSQLRecordPeople;
    VA: TSQLRecordPeopleArray;
    FV: TFV;
(...)
V2.FillPrepare(Client, 'LastName=(:'Dali')');
n := 0;
while V2.FillOne do
begin
  VA.FillFrom(V2); // fast copy some content from TSQLRecordPeople
```

The *FillPrepare* / *FillOne* method are used to loop through all *People* table rows with a *LastName* column value equal to 'Dali' (with a prepared statement thanks to *:( ):*), then initialize a *TSQLRecordPeopleArray* instance with those values, using a *FillFrom* method call.

```
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.fUTF8);
```

The `n` variable is used to follow the `PeopleArray` number, and will most of the type 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:

```
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:

```
VA.DynArray('Ints').Add(n);
```

Note that the `DynArray` method can be used via two overloaded set of parameters: either the field name ('Ints'), either an index value, as was defined in the class definition. So we could have written:

```
VA.DynArray(1).Add(n);
```

since the `Ints` published property has been defined as such:

```
property Ints: TIntegerDynArray index 1 read fInts write fInts;
```

Similarly, the following line will add a currency value to the `Currency` field:

```
VA.DynArray(2).Add(Curr);
```

And a more complex `TFV` record is added to the `FileVersion` field *dynamic array* with just one line:

```
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:

```
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.

#### 1.4.1.4.2.1.4. 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:

```
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;
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 159) on how to implement it.

#### 1.4.1.4.2.1.5. 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):

```
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's mandatory to initialize then release all internal property instances in the overridden constructor and destructor of the class:

```
constructor TSQLRecordPeopleObject.Create;
begin
  inherited;
  fPersistent := TCollTst.Create;
  fUTF8 := TRawUTF8List.Create;
end;

destructor TSQLRecordPeopleObject.Destroy;
begin
  inherited;
  FreeAndNil(fPersistent);
  FreeAndNil(fUTF8);
end;
```

Here is how the regression tests are performed:

```
var V0: TSQLRecordPeopleObject;
(...)
if Client.TransactionBegin(TSQLRecordPeopleObject) then
try
  V2.FillPrepare(Client, 'LastName=?', ['Morse']);
  n := 0;
  while V2.FillOne do
  begin
    V0.FillFrom(V2); // fast copy some content from TSQLRecordPeople
    inc(n);
    V0.Persistent.One.Color := n+100;
    V0.Persistent.One.Length := n;
    V0.Persistent.One.Name := Int32ToUtf8(n);
    if n and 31=0 then
    begin
      V0.UTF8.Add(V0.Persistent.One.Name);
      with V0.Persistent.Coll.Add do
      begin
        Color := n+1000;
        Length := n*2;
        Name := Int32ToUtf8(n*3);
      end;
    end;
    Check(Client.Add(V0,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):

```
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,V0)` method. Data is retrieved from the database as TEXT values, then unserialized 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:

```
{"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.

#### 1.4.1.4.2.1.6. Custom TObject JSON serialization

Not only `TPersistent`, `TCollection` and `TSQLRecord` types can be serialized by writting all published properties.

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 `JSONToObject()` (aka `TJSONSerializer.WriteObject()` functions).

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 `JSONToObject()` patterns, i.e. calling low-level `GetJSONField()` 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`):

```
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, or for main 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:

```
class procedure TCollTstDynArray.FVClassWriter(const aSerializer: TJSONSerializer;
  aValue: TObject; aHumanReadable, aDontStoreDefault, aFullExpand: Boolean);
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:

```
class function TCollTstDynArray.FVClassReader(const aValue: TObject; aFrom: PUTF8Char;
  var aValid: Boolean): PUTF8Char;
var V: TFileVersion absolute aValue;
    Values: TPUTF8CharDynArray;
begin
  aValid := false;
  aFrom := JSONDecode(aFrom,['Major','Minor','Release','Build','Main','BuildDateTime'],Values);
  if aFrom=nil then
    exit;
  V.Major := GetInteger(Values[0]);
  V.Minor := GetInteger(Values[1]);
  V.Release := GetInteger(Values[2]);
  V.Build := GetInteger(Values[3]);
  V.Main := UTF8DecodeToString(Values[4],StrLen(Values[4]));
  V.BuildDateTime := Iso8601ToDateTimePUTF8Char(Values[5]);
  aValid := true;
  result := aFrom;
end;
```

Here, the `JSONDecode` function will un-serialize the JSON object into an array of `PUTF8Char` values, without any memory allocation (in fact, `Values[]` 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:

```
TJSONSerializer.RegisterCustomSerializer(TFileVersion,  
    TCollTstDynArray.FVClassReader, TCollTstDynArray.FVClassWriter);
```

If you want to disable the custom serialization, you may call the same method as such:

```
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.

#### 1.4.1.4.2.2. 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).

##### 1.4.1.4.2.2.1. 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 SQLite3 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 := Iso8601Now;
    MS.fSignatureTime := MD.fSignatureTime;
    Check(MS.DestList<>nil);
    Check(MS.DestList.InheritsFrom(TSQLRecordMany));
    aClient.TransactionBegin(TSQLSource); // faster process
```

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.

```
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;
```

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]);
```

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
  continue;
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 TSQLDest 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.

#### 1.4.1.4.2.2. 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:

```
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:

```
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:

```
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.

#### 1.4.1.5. 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 `SQLite3` unit:

```

function TSQLRestServerDB.EngineUpdate(Table: TSQLRecordClass; ID: integer;
  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 := EngineExecuteFmt('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;
- Either 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 163).

##### 1.4.1.5.1. 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.

```

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:

```
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, either force the update with a line of code like:

```
Invoice.Details := Invoice.Details; // force Total calculation
```

#### 1.4.1.5.2. 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:

```
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:

```
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.

#### 1.4.1.6. 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's 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's 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 provide 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 125) - 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, GDI 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.

#### 1.4.1.6.1. ORM is not DB

It's 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.

#### 1.4.1.6.2. 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 SQLite3Options 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.

#### 1.4.1.6.3. 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):

```
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:

```
DrivesModel := TSQLModel.Create([TDrives], 'root');
GlobalClient := TMyClient.Create(DrivesModel, nil, 'drives.sqlite', TSQLRestServerDB);
GlobalClient.CreateMissingTables(0);
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's 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):

```
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 in 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's certainly worth reading all the (verbose) interface part of the SQLite3Commons.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 <http://synopse.info..>

#### 1.4.1.6.4. 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 125). 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 163).

Another possibility to access your high-level type, is to use either custom *SQLite3* SQL functions either stored procedures - see below (page 158) - both coded in Delphi.

#### 1.4.1.7. 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 58) and below (page 96) - note that this cache is available not only for the *SQLite3* database engine, but also for any external engine - see below (page 112);
- Global JSON result cache at the database level, which is flushed globally on any INSERT / UPDATE - see below (page 126);
- Tuned record cache at the CRUD/RESTful level for specified tables or records on the *server* side - see below (page 155);
- Tuned record cache at the CRUD/RESTful level for specified tables or records on the *client* side - see below (page 155).

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.

#### 1.4.1.8. MVC pattern

##### 1.4.1.8.1. Creating a Model

According to the *Model-View-Controller* (MVC) pattern - see *Model-View-Controller* (page 44) - 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.

Since this class should be used on both Client and Server sides, it's 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`):

```
function CreateSampleModel: TSQLModel;  
begin  
  result := TSQLModel.Create([TSQLSampleRecord]);  
end;
```

For a more complex model, using for instance some User Interface auto-creation, this function could be written as such - extracted from the main demo application, unit `FileTables.pas`:

```
function CreateFileModel(Owner: TSQLRest): TSQLModel;  
var Classes: array[0..high(FileTabs)] of TSQLRecordClass;  
    i: integer;  
begin  
  for i := 0 to high(FileTabs) do  
    Classes[i] := FileTabs[i].Table;  
    result := TSQLModel.Create(Classes, 'synfile');  
    result.Owner := Owner;  
    result.SetActions(TypeInfo(TFileAction));  
    result.SetEvents(TypeInfo(TFileEvent));  
end;
```

An Owner property is also available, and will allow access to the current running client or server TSQLRest instance associated with this model.

This model will also centralize the available User actions and associated events, for the User Interface and Business Logic handling.

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 TSQLRestServerStaticInMemory instance via the TSQLRestServer.StaticDataCreate method, or being external tables - see below (page 112). So, in order to access all tables properties, you may instead use code like this:

```
var i: integer;  
    Props: TSQLRecordProperties;  
begin  
  ...  
  for i := 0 to high(Model.TableProps) do begin  
    Props := Model.TableProps[i];  
    // now you can access Props.SQLiteTableName or Props.Fields[] ...  
  end;  
end;
```

In fact, the TSQLModel.TableProps[] array maps TSQLModel.Tables[].RecordProps, and allow fast direct access to all the needed ORM properties of every TSQLRecord handled by this model, as retrieved from below (page 87). See TSQLRecordProperties fields and methods to see all the available information.

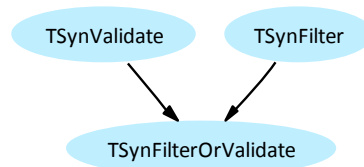
#### 1.4.1.8.2. Filtering and Validating

According to the n-Tier architecture - see *Multi-tier architecture* (page 45) - 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 and validation. They all will be children of any of those both classes:

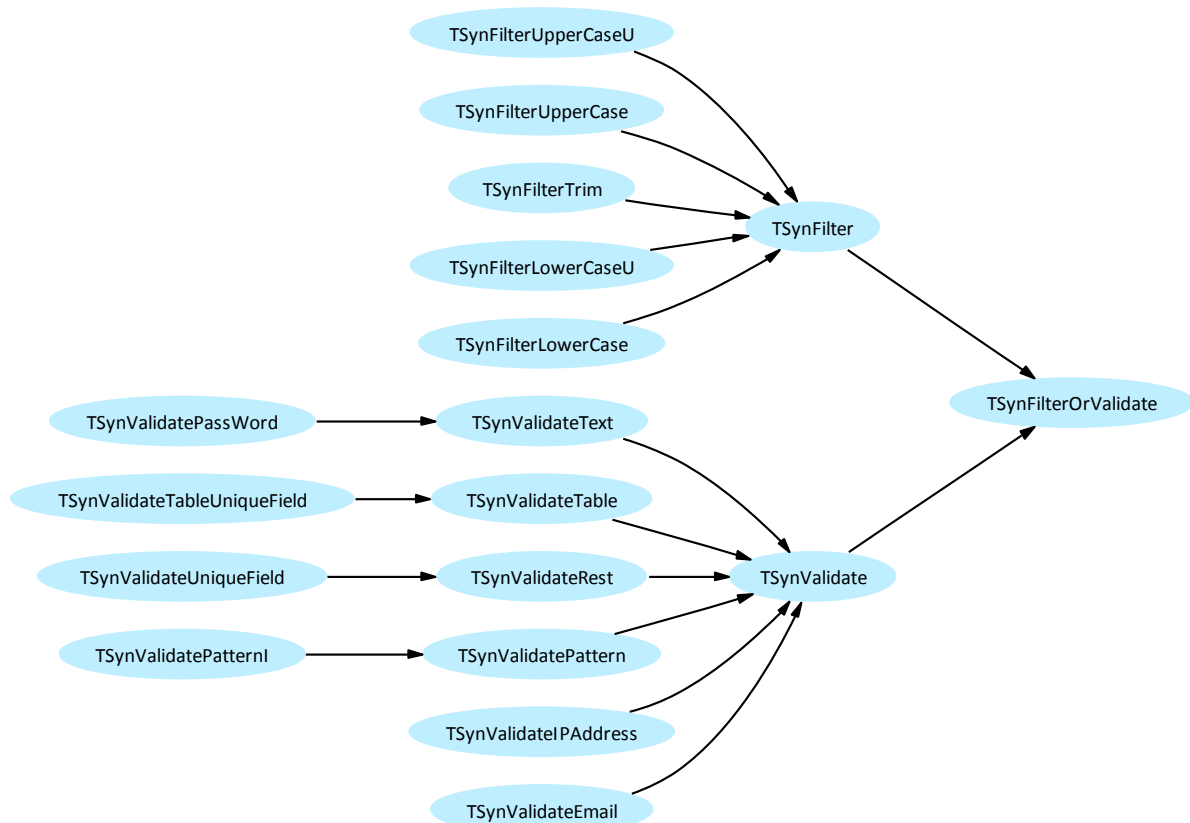


*Filtering and Validation classes hierarchy*

TSQLRecord field content validation is handled in the new TSQLRecord. Validate virtual method, or via some TSynValidate classes.

TSQLRecord field content filtering is handled in the new TSQLRecord. Filter virtual method, or via some TSynFilter classes.

Some "standard" classes are already defined in the SynCommons and SQLite3Commons unit, to be used for most common usage:



*Default filters and Validation classes hierarchy*

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 filtering are existing, like `TSynValidateUniqueField` which inherits

from TSynValidateRest.

Of course, the SQLite3UIEdit unit now handles TSQLRecord automated filtering (using TSQLFilter classes) and validation (using one of the TSQLValidate classes).

The unique field validation is now in TSQLRecord. Validate and not in SQLite3UIEdit itself (to have a better multi-tier architecture).

To initialize it, you can add some filters/validators to your TSQLModel creation function:

```
function CreateFileModel(Owner: TSQLRest): TSQLModel;
var Classes: array[0..high(FileTabs)] of TSQLRecordClass;
    i: integer;
begin
  for i := 0 to high(FileTabs) do
    Classes[i] := FileTabs[i].Table;
  result := TSQLModel.Create(Classes, 'synfile');
  result.Owner := Owner;
  result.SetActions(TypeInfo(TFileAction));
  result.SetEvents(TypeInfo(TFileEvent));
  TSQLFile.AddFilterOrValidate('Name', TSQLFilterLowerCase);
  TSQLUser.AddFilterOrValidate('Email', TSQLValidateEmail);
end;
```

In order to perform the filtering 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 SQLite3UIEdit.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.

#### 1.4.1.8.3. Views

This framework also handles directly the 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.

##### 1.4.1.8.3.1. 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 compiler 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:

```
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..>

Advantages of the RTTI can therefore be sum 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 installation of the software, without additional components or systems.

#### 1.4.1.8.3.2. User Interface

User Interface generation from RTTI and the integrated reporting features will be described below (page 804), during presentation of the Main Demo application design.

#### 1.4.1.9. 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 sound 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.

##### 1.4.1.9.1. 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 false" property, which is used to identify unique mandatory columns.

Using attributes is one of the most common way 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 downside, 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 85) -

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 194). 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. 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.

#### 1.4.1.9.2. Several ORMs at once

To be clear, *mORMot* offers three kind of table definitions:

- Via *TSQLRecord* / *TSQLRecordVirtual* "native ORM" classes: data storage is using either fast in-memory lists via *TSQLRestServerStaticInMemory*, either *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()* function, external DB tables are created and managed by the ORM, via SQL - see below (page 112). These classes will allow creation of tables in any supported external database engine (*SQLite3*, *Oracle*, *MS SQL*, *Jet*, whatever *OleDB* or *ODBC* provider). In this case, we use index for text column length. This is the only needed parameter to be defined for such a basic implementation, in regard to *TSQLRecord* kind of classes.
- 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, either with dedicated classes (or interface).

The concern of not being able to persist any class (it needs to inherit from *TSQLRecord*) does perfectly make sense.

On the other hand, from the implementation point of view, it is very powerful, since you have a lot of methods included within this class definition. It does also make sense to have a common ancestor able to identify all three kind of *mORMot*'s table definitions: the same abstract ancestor is used, and clients won't even need to know that they are implemented in-memory, using a *SQLite3* engine, or even a MS SQL / Oracle database. Another benefit of using a parent class is to enforce code safety using Delphi's *strong type* abilities: you won't be able to pass a non-persistent type to methods which expect one.

From the Domain-Driven / SOA point of view, it is now an established rule to make a distinction between DTO (Data Transfer Objects) and Domain Values (*Entity objects* or *Aggregates*). In most implementations, persistence objects (aka ORM objects) should be either the aggregate roots themselves (you do not store Entity objects and even worse DTOs), either dedicated classes. Do not mix layers, unless you like your software to be a maintenance nightmare!

Some Event-Sourcing architectures even implement *several DB back-end at once*:

- It will store the status on one DB (e.g. high-performance in-memory) for most common requests to be immediate;
- And store the modification events in another ACID DB (e.g. *SQLite3*, *MS SQL* or *Oracle*);
- And even sometimes fill some dedicated consolidation DBs for further analysis.

AFAIK it could be 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.

#### **1.4.1.9.3. The best ORM is the one you need**

Therefore, we may sum up some potential use of ORM, depending of your intent:

- If your understanding of ORM is just to persist some existing objects, *mORMot* won't help you directly (but we identified that some users are using the built-in JSON serialization feature of the framework to create their own dedicated Client-Server ORM-like platform);
- If you want to persist some data objects (not tied to complex business logic), the framework will be a light and fast candidate, via *SQLite3*, *Oracle*, *MS SQL* or even with no SQL engine, using *TSQLRestServerStaticInMemory* class which is able to persist its content with small files - see below (page 109);
- If you need (perhaps not now, but probably in the future) to create some kind of scalable domain-driven architecture, you'll have all needed features at hand with *mORMot*;
- If your expectation is to map an existing complex DB, *mORMot* will handle it soon (it is planned and prepared within the framework architecture).

Therefore, *mORMot* is not just an ORM, nor just a "classic" ORM.



## 1.4.2. Database layer

### 1.4.2.1. SQLite3-powered, not SQLite3-limited

#### 1.4.2.1.1. 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:

- 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 AES-256 level, that is Top-Secret security);
- Database layout is declared once in the Delphi source code (as published properties of classes), avoiding 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);
- *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 *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);
- 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 <http://sqlite.org>. - use the amalgamation C file with a few minor changes (documented in the *SynSQLite3.pas* unit) - the resulting C source code delivered as .obj is also available in the official *Synapse* source code repository.

The overhead of including *SQLite3* in your server application will be worth it: just some KB to the executable, but with so many nice features, even if only external databases are used.

#### 1.4.2.1.2. 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 *SQLite3Commons.pas*. A bridge between the two units is made with *SQLite3.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 103). For instance, any external database (via *OleDB* / *ODBC* providers or direct *Oracle* connection) can be accessed via our *SynDB*-based dedicated units, as stated below (page 112).

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.

#### 1.4.2.1.3. Data access benchmark

On an average desktop computer, depending on the backend database interfaced, *mORMot* excels in speed:

- You can persist up to 150,000 objects per second, or retrieve 240,000 objects per second (for our pure Delphi in-memory engine);
- When data is retrieved from server or client *ORM Cache* (page 84), you can read 450,000 objects per second;
- With a high-performance database like *Oracle* and our direct access classes, you can write 53,000 and read 72,000 objects per second, over a 100 MB network.

Difficult to find a faster ORM, I suspect.

The following tables try to sum up all available possibilities, and give some benchmark (average objects/second for writing or read).

In these tables:

- 'internal' means use of the internal *SQLite3* engine, either via a 'file' back-end or in 'memory';
- 'external' stands for an external access via *SynDB* - see below (page 112);
- 'file off' stands for the file back-end, with *Synchronous := smOff*;
- 'TObjectList' indicates a *TSQLRestServerStaticInMemory* instance - see below (page 109) - 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;

- numbers are expressed in rows/second (or objects/second), 'k' standing for 1000 (e.g. '15k' = 15,000 objects per second);
- 'trans' stands for *Transaction*, i.e. when the write process is nested within BeginTransaction / Commit calls;
- 'batch' mode will be described below (page 151);
- 'read one' states that one object is read per call (ORM generates a `SELECT * FROM table WHERE ID=?`);
- 'read all' is when all 5000 objects are read in a single call (i.e. running `SELECT * FROM table`);
- 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.

Benchmark was run on a good old Core 2 Duo workstation (no SSD), with anti-virus and background applications, over a 100 Mb corporate network, linked to a shared *Oracle* 11g database. So it was a development environment, very similar to production site, not dedicated to give best performance. 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.

You can compile the 15 - External DB performance supplied sample code, and run the very same benchmark on your own configuration.

Database	ACID	Persist	Write one	Write trans	Write batch	Read one	Read all
internal SQLite3 file	Yes	Yes	8	15k	16k	10k	170k
internal SQLite3 file off	Yes	Yes	400	35k	38k	10k	170k
internal SQLite3 mem	No	No	30k	35k	45k	44k	170k
TObjectList static	No	Yes	97k	145k	147k	100k	234k
TObjectList virtual	No	Yes	97k	145k	147k	99k	234k
external SQLite3 file	Yes	Yes	8	13k	15k	45k	160k
external SQLite3 file off	Yes	Yes	400	31k	38k	10k	170k
external SQLite3 mem	No	No	26k	32k	40k	47k	160k
external Oracle	Yes	Yes	460	715	53k	800	72k
external Jet	No	Yes	688	900	900	1000	76k

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 so slow (less than 10 objects per second) 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` 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 up to 400 objects per second, as stated by the "file off" rows of the table - see below (page 102).

Therefore, the typical use may be the following:

Database	Created by	Use
int. SQLite3 file	default	General safe data handling
int. SQLite3 mem	:memory:	Fast data handling with no persistence (e.g. for testing)
TObjectList static	StaticDataCreate	Best possible performance for small amount of data, without ACID nor SQL
TObjectList virtual	VirtualTableRegister	Best possible performance for small amount of data, if ACID is not required nor complex SQL
ext. SQLite3 file	VirtualTableExternalRegister	External back-end, e.g. for disk spanning
ext. SQLite3 mem	VirtualTableExternalRegister	Fast external back-end (e.g. for testing)
ext. Oracle	VirtualTableExternalRegister	Fast, secure and industry standard; can be shared outside <i>mORMot</i>
ext. Jet	VirtualTableExternalRegister	Could be used as a data exchange format (e.g. with Office applications)

For both writing and reading, TObjectList / TSQLRestServerStaticInMemory 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 `false` properties. As a consequence search of not unique values may be slow: the engine has to loop through all rows of data. But for unique values (defined as stored `false`), both insertion and search speed is awesome, due to its optimized O(1) hash algorithm - see the following benchmark, especially the "By name" row, which corresponds to a search of an unique RawUTF8 property value.

	SQLite3 (file full)	SQLite3 (file off)	SQLite3 (mem)	TObjectList (static)	TObjectList (virtual)	SQLite3 (ext file full)	SQLite3 (ext file off)	SQLite3 (ext mem)	Oracle	Jet
<b>By one</b>	10461	10549	44737	103577	103553	43367	44099	45220	901	1074
<b>By name</b>	9694	9651	32350	70534	60153	22785	22240	23055	889	1071
<b>All Virtual</b>	167095	162956	168651	253292	118203	97083	90592	94688	56639	52764
<b>All Direct</b>	167123	144250	168577	254284	256383	170794	165601	168856	88342	75999

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!

In the above list, the *MS SQL Server* is not integrated, but may be used instead of *Oracle* (minus the fact that BULK insert is not implemented yet for it, whereas array binding boosts *Oracle* writing BATCH process performance by 100 times). Any other OleDB or ODBC providers may also be used, with direct access (without *DBExpress* / *BDE* layer nor heavy *TDataSet* instance).

Note that all those tests were performed locally and in-process, via a *TSQLRestClientDB* instance. For both insertion and reading, a Client-Server architecture (e.g. using HTTP/1.1 for *mORMot* clients) will give even better results for BATCH and retrieve all modes. During the tests, internal caching - see below (page 126) and *ORM Cache* (page 84) - 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 450,000 read requests per second, whatever database is used.

#### 1.4.2.2. SQLite3 implementation

Beginning with the revision 1.15 of the framework, the *SQLite3* engine itself has been separated from our *SQLite3.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;
- Either stand-alone with high-level SQL access, using our *SynDB* generic access classes, via *SynDBSQLite3.pas* - so you will be able to change to any other database engine (e.g. MSSQL or Oracle) when needed;
- Either Client-Server based access with all our ORM features - see *SQLite3.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.

##### 1.4.2.2.1. 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:

```
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:

```
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:

```
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):

```
:(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:

```
function TSQLRestServerDB.EngineDelete(Table: TSQLRecordClass; ID: integer): boolean;
begin
  if Assigned(OnUpdateEvent) then
    OnUpdateEvent(self, seDelete, Table, ID); // notify BEFORE deletion
  result := EngineExecuteFmt('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 would rather use the overloaded TSQLRecord.Create, TSQLRecord.FillPrepare, TSQLRecord.CreateAndFillPrepare, TSQLRest.OneFieldValue, TSQLRest.MultiFieldValues, TSQLRestClient.EngineExecuteFmt and TSQLRestClient.ListFmt methods, available since revision 1.15 of the framework, which will accept both '%' and '?' characters in the SQL WHERE format text, inlining '?' 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:

```
SQL.Text := 'SELECT Name FROM Table WHERE ID=:Index';
SQL.ParamByName('Index').AsInteger := aID;
```

At a lowest-level, inlining 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's 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.

#### 1.4.2.2.2. 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>.

A dedicated ORM class, named TSQLRecordRTree, is available to create such tables. It inherits from TSQLRecordVirtual, like the other virtual tables types (e.g. TSQLRecordFTS3).

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: integer 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 a MapData. BlobField filled with [-81,-79.6,35,36.2] the following lines:

```
aClient.RTreeMatch(TSQLRecordMapData,'BlobField',TSQLRecordMapBox,
  aMapData.BlobField,ResultID);
```

will execute the following SQL 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')');
```

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 minX>=...maxY<=... where clause).

#### 1.4.2.2.3. FTS3/FTS4

FTS3/FTS4 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> as reference material about FTS3 usage in *SQLite3*.

Since version 1.5 of the framework, the `sqlite3fts3.obj` file is always available in the distribution file: just define the `INCLUDE_FTS3` conditional globally for your application (it is expected e.g. in `SQLite3.pas` and `SynSQLite3.pas`) to enable *FTS3* in your application.

Leave it undefined if you do not need this feature, and will therefore spare some KB of code.

FTS3 and FTS4 are nearly identical. FTS4 is indeed an enhancement to FTS3, added with *SQLite* version 3.7.4, and included in the release v.1.11 of the framework. They share most of their code in common, and their interfaces are the same. The differences are:

- FTS4 contains query performance optimizations that may significantly improve the performance of full-text queries that contain terms that are very common (present in a large percentage of table rows).
- FTS4 supports some additional options that may be used with the `matchinfo()` function.

Because it stores extra information on disk in two new shadow tables in order to support the performance optimizations and extra `matchinfo()` options, FTS4 tables may consume more disk space than the equivalent table created using FTS3. Usually the overhead is 1-2% or less, but may be as high as 10% if the documents stored in the FTS table are very small. The overhead may be reduced by using a `TSQLRecordFTS3` table type instead of `TSQLRecordFTS4` declaration, but this comes at the expense of sacrificing some of the extra supported `matchinfo()` options.

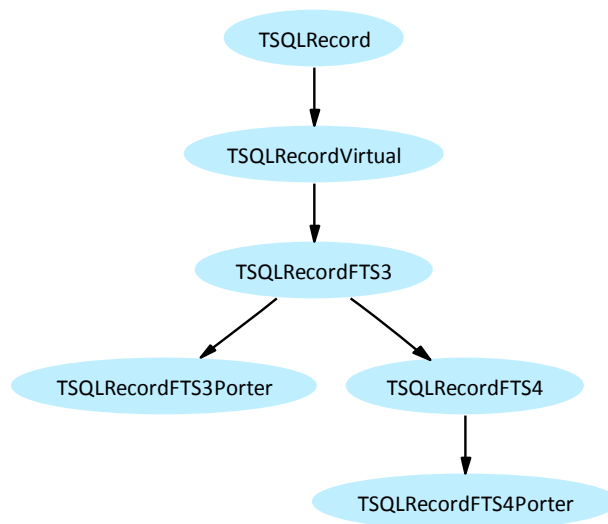
#### 1.4.2.3.1. Dedicated FTS3/FTS4 record type

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);
- `TSQLRecordFTS4` to create a FTS4 table with default "simple" stemming;
- `TSQLRecordFTS4Porter` to create a FTS4 table using the *Porter Stemming* algorithm.

The following graph will detail this class hierarchy:





*FTS3/FTS4 ORM classes*

#### 1.4.2.2.3.2. 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 codepoints 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 codepoints 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 would 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 thei veri frustrat". 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..> page.

#### 1.4.2.2.3.3. FTS searches

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 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:

```
TSQlFTSTest = class(TSQLRecordFTS3)
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 2009/2010/XE/XE2, 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 FTS3 table, just like any regular TSQLRecord content, via the ORM feature of the framework:

```
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));
```

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 TSQLRecordFTS3 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 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's 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 TSQLRecordFTS3 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](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.

#### 1.4.2.2.4. NULL handling

Since you access Delphi properties, NULL doesn't exist as such (it's a SQL concept). So you will have 0 for an integer field, nil for a field referring to another record, and "" for a string field. At the SQL and JSON levels, the NULL value does exist and are converted as expected. At higher level (Delphi code or JavaScript/AJAX code) the NULL value is to be handled explicitly.

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).

There is no direct way of making a difference between NULL and "" for a string field, for example. It can be performed by using a simple SQL statement, which can be added to your database class, as a method common to all your application tables classes. These methods are not yet implemented.

But it's worth noting that NULL handling is not consistent among databases... so we should recommend not using it in any database statements, or only in a 100% compatible way.

#### 1.4.2.2.5. ACID and speed

As stated above in *Data access benchmark* (page 93), 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.

You can overwrite this default 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 93) 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:

```
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 112), you can define the setting for a particular external connection, for instance:

```
Props := TSQLDBSQLite3ConnectionProperties.Create(DBFileName, '', '');  
VirtualTableExternalRegister(Model, TSQLRecordSample, Props, 'SampleRecord');  
Client := TSQLRestClientDB.Create(Model, nil, MainDBFileName, TSQLRestServerDB, false, '');  
TSQLDBSQLite3Connection(Props.MainConnection).Synchronous := smOff;
```

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.

If you can afford losing 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` would help your application scale. Consider using an external and dedicated database (like *Oracle* or *MS SQL*) if your security expectations are very high, and if the default `Synchronous := smFull` safe but slow setting is not enough for you.

In all cases, do not forget to perform backups as often as possible (at least several times a day). You may use *TSQLRestServerDB.Backup* or *TSQLRestServerDB.BackupGZ* methods for a fast backup of a running database. Adding a backup feature on the server side is as simple as running:

```
Client.Server.BackupGZ(MainDBFileName+'.gz');
```

Server will stop working during this phase, so a lower-level backup mechanism could be used instead, if you need 100% of service availability. Using an external database would perhaps keep you main *mORMot* database small in size, so that its backup time will remain unnoticeable on the client side.

#### 1.4.2.3. 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 112). Or the application might compute the content of the virtual table on demand.

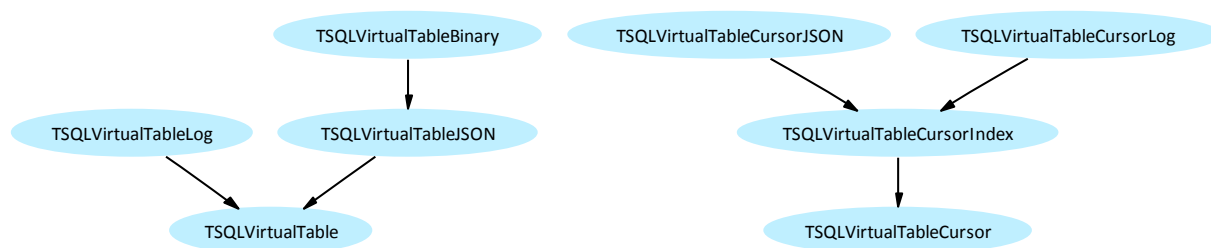
Thanks to the generic implementation of Virtual Table in *SQLite3*, you can use such tables in your SQL statement, and even safely execute a SELECT statement with JOIN or custom functions, mixing normal *SQLite3* tables and any other Virtual Table. From the ORM point of view, virtual tables are just tables, i.e. they inherit from *TSQLRecordVirtual1*, which inherits from the common base *TSQLRecord* class.

#### 1.4.2.3.1. 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 *SQLite3Commons.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 *TSQLRestServerStaticInMemory* 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), either 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 122).

As you probably have already stated, all those Virtual Table mechanism is implemented in SQLite3Commons. 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.

#### 1.4.2.3.2. 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:

```
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:

```
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:

```
TSQLRecordLogFile = class(TSQLRecordVirtualTableAutoID)
protected
  fContent: RawUTF8;
  fDateTime: TDateTime;
  fLevel: TSynLogInfo;
published
  /// the log event time stamp
  property DateTime: TDateTime read fDateTime;
  /// the log event level
  property Level: TSynLogInfo read fLevel;
  /// the textual message associated to the log event
  property Content: RawUTF8 read fContent;
end;
```

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 `TSQLRestServerStaticInMemory` 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:

```
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;
```

Then here is how each 'log' virtual table module instance is created:

```
constructor TSQLVirtualTableLog.Create(aModule: TSQLVirtualTableModule;
  const aTableName: RawUTF8; FieldCount: integer; Fields: PUTF8CharArray);
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:

```
destructor TSQLVirtualTableLog.Destroy;
begin
  FreeAndNil(fLogFile);
  inherited;
end;
```

Then the corresponding cursor is defined as such:

```
TSQLVirtualTableCursorLog = class(TSQLVirtualTableCursorIndex)
public
  function Search(const Prepared: TSQLVirtualTablePrepared): boolean; override;
  function Column(aColumn: integer; var aResult: TVarData): boolean; override;
end;
```

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:



```
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 // found one
        fMax := fCurrent-1; // out of range ID
      end;
    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:

```
function TSQLVirtualTableCursorLog.Column(aColumn: integer;
  var aResult: TVarData): boolean;
var LogFile: TSynLogFile;
begin
  result := false;
  if (self=nil) or (fCurrent>fMax) then
    exit;
  LogFile := TSQLVirtualTableLog(Table).fLogFile;
  if LogFile=nil then
    exit;
  case aColumn of
    -1: SetColumn(aResult,fCurrent+1); // ID = index + 1
    0: SetColumn(aResult,LogFile.EventDateTime(fCurrent));
    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.

#### 1.4.2.3.3. 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);');
```

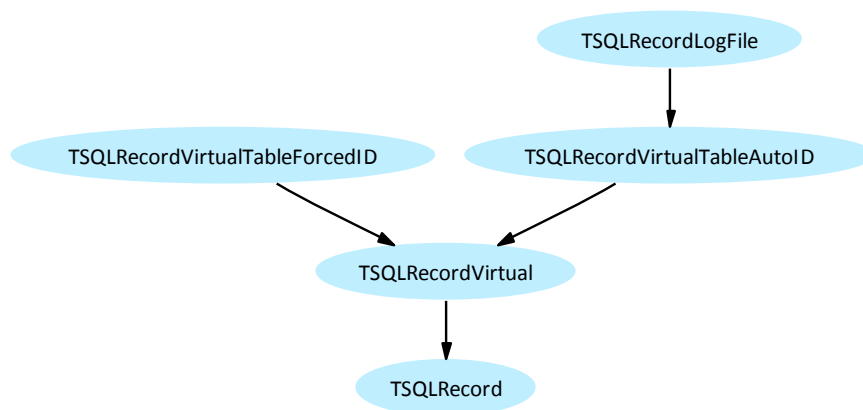
This will create the virtual table. Since all fields are already known by the `TSQLVirtualTableLog` class, it's not necessary to specify the fields at this level. We only specify the log file name, which will be retrieved by `TSQLVirtualTableLog`. Create constructor.

```
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.

#### 1.4.2.3.4. 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 `TSQLRecordFTS3/4`).

`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:

```
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 stand-alone application). Otherwise, an exception is raised at virtual table creation.

#### 1.4.2.3.5. In-Memory "static" process

We have seen that the TSQLVirtualTableJSON, TSQLVirtualTableBinary and TSQLVirtualTableCursorJSON classes implement a Virtual Table module using a TSQLRestServerStaticInMemory 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 TSQLRestServerStaticInMemory 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 TSQLRestServerStaticInMemory allows, and definitively makes sense for an ORM framework;
- On the client or server side, you could create calculated fields easily with TSQLRestServerStaticInMemory dedicated "getter" methods written in Delphi, whereas *SQLite3* in-memory tables would 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 TSQLRestServerStaticInMemory 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.

##### 1.4.2.3.5.1. 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 TSQLRestServerStaticInMemory 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.

#### 1.4.2.3.5.2. 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:

```
TSQLRecordDali1 = 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;
TSQLRecordDali2 = class(TSQLRecordDali1);
```

Both class types are then added to the TSQLModel instance of the application, common to both Client and Server side:

```
ModelC := TSQLModel.Create(
  [TSQLRecordPeople, (...)
  TSQLRecordDali1, TSQLRecordDali2], 'root');
```

Then, on the Server side, the corresponding Virtual Table modules are associated with those both classes:

```
ModelC.VirtualTableRegister(TSQLRecordDali1, TSQLVirtualTableJSON);
ModelC.VirtualTableRegister(TSQLRecordDali2, 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:

```
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:

```
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 TSQLRestServerStaticInMemory-based virtual tables. In fact, the Commit method in the above code will call TSQLRestServerStaticInMemory.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 TSQLRestServerStaticInMemory.UpdateToFile method is not immediate, because it will write all table data each time on disk. It's 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's 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's 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:

```
Server.StaticVirtualTable[TSQLRecordDali1].CommitShouldNotUpdateFile := true;
```

In order to create disk content, you'll then have to explicitly call the corresponding method on purpose:

```
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!

#### 1.4.2.3.6. Virtual Tables to access external databases

As will be stated below (page 112), 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, *Microsoft SQL Server*, *MySQL* 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()` function 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).

#### 1.4.2.4. External database access

##### 1.4.2.4.1. Database agnosticism

Since revision 1.15, our ORM RESTful framework is able to access any available database engine, via a set of generic units and classes.

The framework still relies on *SQLite3* as its SQL core on the server, but a dedicated mechanism allows access to any remote database, and mix those tables content with the native ORM tables of the framework. Thanks to the unique Virtual Tables mechanism of *SQLite3*, those external tables may be accessed as native *SQLite3* tables in our SQL statements. See *General mORMot architecture - Client Server implementation* (page 52).

The current list of available external database classes is:

- Any *OleDB* provider (including *MS SQL Server*, *Jet* or others);
- Any *ODBC* provider (including *FireBird*, *MySQL*, or others);
- *Oracle* direct access (via OCI);
- A *SQLite3* database file.

This list is not closed, and may be completed in the near future. Any help is welcome here: it's 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!

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).

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.

Thanks to the design of our classes, it was very easy (and convenient) to implement *SQLite3* direct access. It's even used for our regression tests, in order to implement stand-alone unitary testing.

#### 1.4.2.4.1.1. Direct access to any Database engine

The SynDB units have the following features:

- Direct fast access to *OleDB*, *ODBC*, *Oracle* (via *OCI*) or *SQLite3* (statically linked) databases;
- Generic abstract OOP layout, able to work with any SQL-based database engine;
- Tested with *MS SQL Server 2008*, *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);
- Ability to be truly Unicode, even with pre-Unicode version of *Delphi* (like *Delphi 7* or *2007*) - use 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;
- Direct 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 the deprecated *BDE* technology;
- Free *SynDBExplorer* tool provided, which is a small but efficient way of running queries in a simple User Interface, about all available engines; it is also a good sample program of a stand-alone usage

of those libraries.

#### 1.4.2.4.1.2. 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:

```
TSQLDBFieldType =  
(ftUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob);
```

Those types will map low-level database-level access types, not high-level Delphi types as TSQLFieldType defined in SQLite3Commons, 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...>

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. It will therefore interface directly with our ORM, which uses the same encoding.

BLOB columns or parameters are accessed as RawByteString variables, which may be mapped to a standard TStream via our TRawByteStringStream.

#### 1.4.2.4.1.3. SynDB Units

Here are the units implementing the external database-agnostic features:

File	Description
SynDB	abstract database direct access classes
SynOleDb	OleDb direct access classes
SynDBODBC	ODBC direct access classes
SynDBOracle	Oracle DB direct access classes (via OCI)
SynDBSQLite3	SQLite3 direct access classes

It's worth noting that those units only depend on SynCommons, therefore are independent of the ORM part of our framework. 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, switching from one database engine to another is just a matter of changing a class type.

#### 1.4.2.4.1.4. Classes and generic use

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.

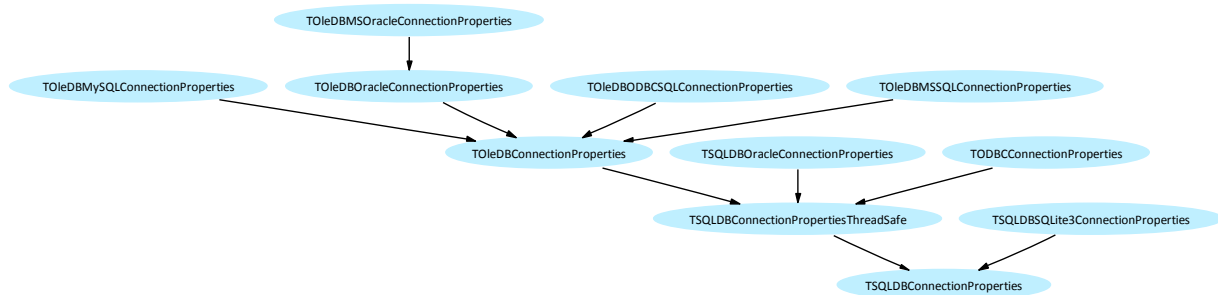
Here is some working sample program, using our `SynOleDB` unit to connect to a local *MS SQL Server 2008 R2 Express edition*, which will write a file with the JSON representation of the `Person.Address` table of the sample database *AdventureWorks2008R2*:

```
program TestOleDB;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Classes,
  SynCommons,
  SynOleDB;

var Props: TOleDBConnectionProperties;
    Conn: TSQLDBConnection;
    Query: TSQLDBStatement;
    F: TFileStream;
begin
  with OleDBSynLogClass.Family do begin
    Level := LOG_VERBOSE;
    AutoFlushTimeOut := 10;
  end;
  Props := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS', 'AdventureWorks2008R2', '', '');
  try
    //Props.ConnectionStringDialogExecute;
    Conn := Props.NewConnection;
    try
      Query := Conn.NewStatement;
      try
        Query.Execute('select * from Person.Address', true, []);
        F := TFileStream.Create(ChangeFileExt(paramstr(0), '.json'), fmCreate);
        try
          Query.FetchAllToJSON(F, false);
        finally
          F.Free;
        end;
      finally
        Query.Free;
      end;
    finally
      Conn.Free;
    end;
  finally
    Props.Free;
  end;
end.
```

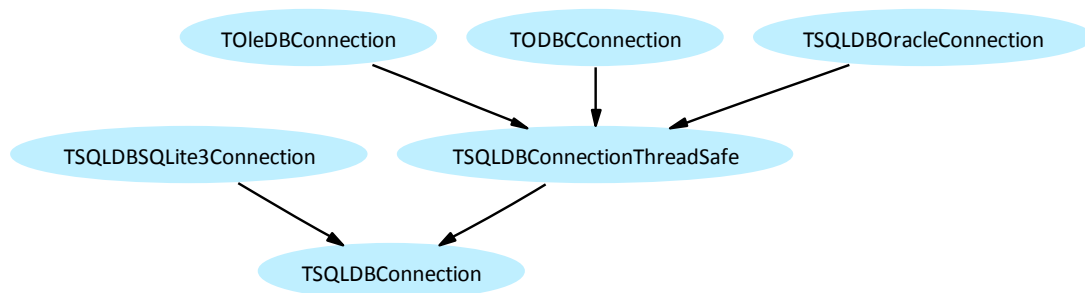
Here are the general class hierarchy, for all available remote *connection properties*:





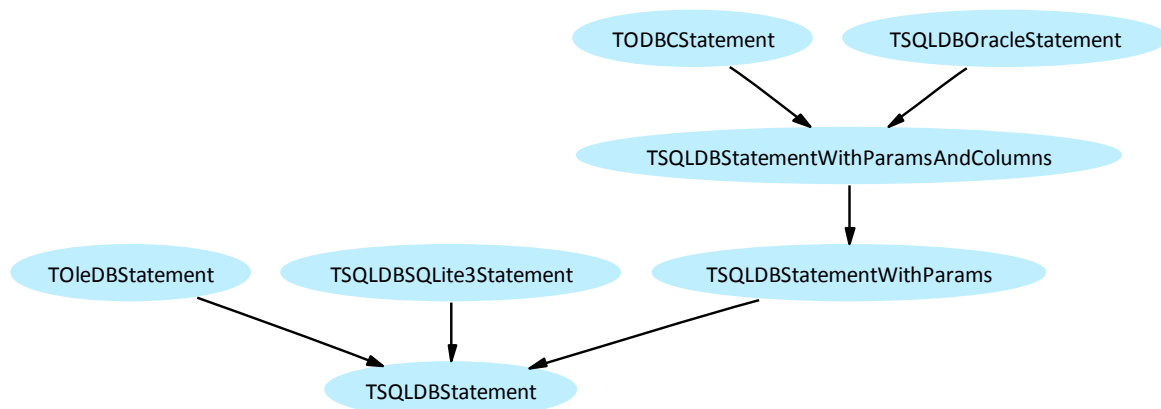
*TSQLDBConnectionProperties classes hierarchy*

Then the following *connection* classes are defined - the usable classes are TOLeDBConnection, TODBConnection, TSQLDBOracleConnection and TSQLDBSQLite3Connection:



*TSQLDBConnection classes hierarchy*

Each connection may create a corresponding *statement* instance:



*TSQLDBStatement classes hierarchy*

You can specify parameters, bound to the request, as such:

```
Query.Execute('select * from Person.Customer where Name like ?',true,['B%']);
```

Or using direct *Bind\*()* methods over a prepared statement. All bound parameters will appear within the SQL statement, when logged using our TSynLog classes - see below (page 219).

#### 1.4.2.4.1.5. ISQLDBRows interface

In order to allow shorter code, an interface type can be used to reference a statement instance.



It allows writing code 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']));
end;
```

In this procedure, no TSQLDBStatement is defined, and there is no need to add a try ... finally Query.Free; end; 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.

#### 1.4.2.4.1.6. 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 where native object properties.

The resulting Delphi code to write is just clear and obvious:

```
Props := TOleDBMSSQLConnectionProperties.Create('.\SQLEXPRESS','AdventureWorks2008R2','','');
procedure TestISQLDBRowsVariant;
var Row: Variant;
begin
  OleDBSynLogClass.Enter;
  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:

```
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's *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 library uses a fast lookup using hashing, the following code will always be faster:

```
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 much slower, after profiling. In fact, the `Row.AccountNumber` expression calls an hidden `DispInvoke` function, which is slow when called multiple times. Our SynCommons 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*.

#### 1.4.2.4.2. Database access

##### 1.4.2.4.2.1. 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, 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...

*ODBC* (*Open DataBase Connectivity*) is a standard C programming language middleware 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 Micro\$oft... one more time!

<http://blogs.msdn.com/b/sqlnativeclient/archive/2011/08/29/microsoft-is-aligning-with-odbc-for-native-relational-data-access.aspx..>

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/AnyDAC/ZDBC* component sets.

Most *OleDB* / *ODBC* providers are free (even maintained by the database owner), others would 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.

#### 1.4.2.4.2.2. Oracle via OCI

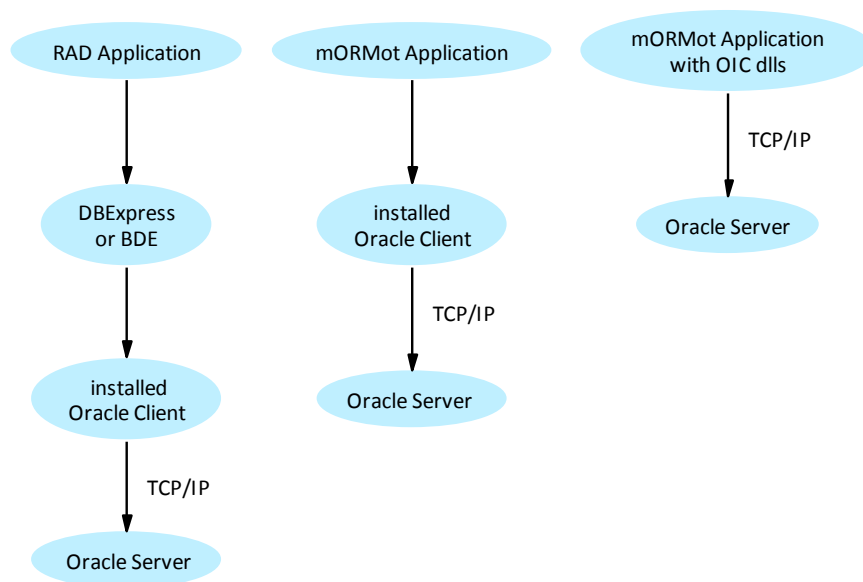
For our framework, and in completion to our *SynOleDb* / *SynDBODBC* units, the *SynDBOracle* unit has been implemented. It allows direct access to any remote Oracle server, using the *Oracle Call Interface*.

*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*.

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. We noted also that our implementation is 10 times faster than the one provided with *ZEOS/ZDBC*, which is far from optimized (it retrieves the rows one by one from OCI).

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..> - which allows to run client applications without installing the standard (huge) Oracle client or having an *ORACLE\_HOME*. 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).



*Oracle Connectivity with SynDBOracle*

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 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 (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);
- Implements *Array Binding* for very fast bulk modifications - insert, update or deletion of a lot of rows at once - see *Data access benchmark* (page 93);
- Handle *Prepared Statements* - but by default, we rely on OCI-side statement cache, if available;
- Native *export to JSON* methods, which will be the main entry point for our ORM framework.

#### 1.4.2.4.2.3. SQLite3

For our ORM framework, we implemented an efficient *SQLite3* wrapper, joining statically (i.e. without any external dll, but within the main exe) the *SQLite3* engine to the executable.

It was an easy task to let the SynSQLite3.pas unit be called from our SynDB database abstract classes. Adding such another Database is just a very thin layer, implemented in the SynDBSQLite3.pas unit.

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.

This 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).

#### 1.4.2.4.3. ORM Integration

##### 1.4.2.4.3.1. Transparent use

An *external* record can be defined as such:

```
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 73) 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 WideChar) to be used when creating the external field for TEXT column. 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 degraded).

Here is an extract of the regression test corresponding to external databases:

```
var RInt: TSQLRecordPeople;
    RExt: TSQLRecordPeopleExt;
    (...)
fConnection := TSQLDBSQLite3ConnectionProperties.Create(':memory:', '', '', '');
VirtualTableExternalRegister(fModel, TSQLRecordPeopleExt, fConnection, 'PeopleExternal');
fClient := TSQLRestClientDB.Create(fModel, nil, 'test.db3', TSQLRestServerDB);
fClient.Server.StaticVirtualTableDirect := StaticVirtualTableDirect;
fClient.Server.CreateMissingTables;
    (...)
while RInt.FillOne do begin
  RExt.Data := RInt.Data;
  (...)
  aID := fClient.Add(RExt, true);
  (...)
  Check(fClient.Retrieve(aID, RExt));
  (...)
end;
Check(fClient.Server.CreateSQLMultiIndex(
  TSQLRecordPeopleExt, ['FirstName', 'LastName'], false));
Check(RInt.FillRewind);
while RInt.FillOne do begin
  RExt.FillPrepare(fClient, 'FirstName=? and LastName=?',
    [RInt.FirstName, RInt.LastName]); // query will use index -> fast :)
  while RExt.FillOne do begin
    Check(RExt.FirstName=RInt.FirstName);
  (...)
  end;
end;
```

```

Now := fClient.ServerTimeStamp;
for i := 1 to aID do
  if i mod 100=0 then begin
    Check(fClient.Retrieve(i,RExt,true),'for update');
    RExt.YearOfBirth := RExt.YearOfDeath;
    Check(fClient.Update(RExt),'Update 1/100 rows');
    Check(fClient.Unlock(RExt));
  end;
for i := 1 to aID do
  if i and 127=0 then
    Check(fClient.Delete(TSQLRecordPeopleExt,i),'Delete 1/128 rows');
for i := 1 to aID do begin
  ok := fClient.Retrieve(i,RExt,false);
  Check(ok=(i and 127<>0),'deletion');
  if ok then begin
    Check(RExt.CreatedAt<=Now);
    if i mod 100=0 then begin
      Check(RExt.YearOfBirth=RExt.YearOfDeath,'Updated');
      Check(RExt.LastChange>=Now);
    end else begin
      Check(RExt.YearOfBirth<>RExt.YearOfDeath,'Not Updated');
      Check(RExt.LastChange<=Now);
    end;
  end;
end;
end;

```

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 methods, i.e. Add / Retrieve / Update / Unlock / Delete, 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, with the appropriate column properties according to the database expectations (e.g. a TEXT for *SQLite3* will be a NVARCHAR2 field for *Oracle*).

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.

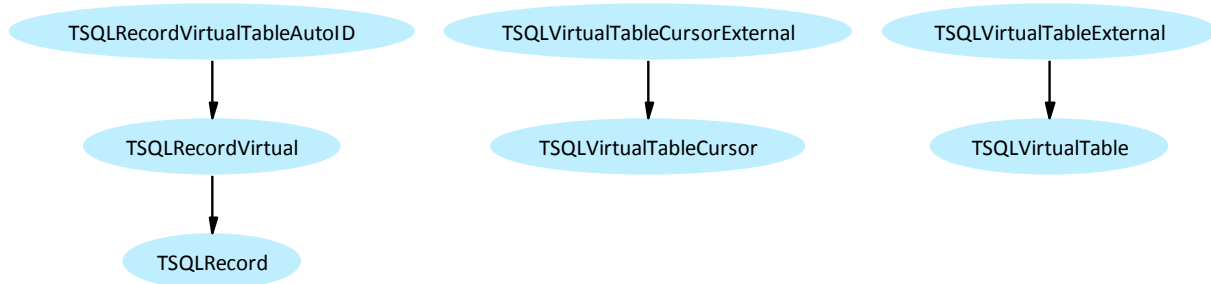
Note that in the above code, the LastChange field was defined as a TModTime: in fact, the current date and time will be stored each time the record is updated, i.e. for each fClient.Add or fClient.Update calls. This is tested by both RExt.LastChange>=Now and RExt.LastChange<=Now checks in the latest 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 Now := fClient.ServerTimeStamp instead of the local Iso8601Now 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<=Now check in the above code.

#### 1.4.2.4.3.2. Behind the scene

The SQLite3DB.pas unit, introduced in revision 1.15, implements Virtual Tables access for any SynDB-based external database for the framework.

In fact, the new `TSQLRestServerStaticExternal`, `TSQLVirtualTableCursorExternal` and `TSQLVirtualTableExternal` classes will implement this feature:



*External Databases classes hierarchy*

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: i.e. it will therefore have an `Integer RowID` published property, auto-incremented at every record insertion (auto-increment will be handled via a `select max(rowid) from tablename`, since not all databases handle such fields - e.g. *Oracle*).

The registration of the class is done by a call to the following new global procedure:

```
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 `TSQLRecordProperties.Kind` property will be overwritten to `rCustomAutoID`);
- 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).

Typical usage may be for instance:

```
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 53).

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 in the SQL statements. Depending on the implementation need, classes could be persistent either via the internal *SQLite3* engine, either via external databases, just via a call to `VirtualTableExternalRegister()` before server initialization.



In fact, `TSQLVirtualTableCursorExternal` will convert any query on the external table into a proper optimized SQL query, according to the indexes existing on the external database. `TSQLVirtualTableExternal` 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 `TSQLRestServerStaticExternal` 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.

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: 198,767 assertions passed  1.39s  
- External via virtual table: 198,767 assertions passed  3.41s
```

The first run is made with `TSQLRestServer`. `StaticVirtualTableDirect` set to `TRUE` (which is the default) - i.e. it will call directly `TSQLRestServerStaticExternal` 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's worth saying that this test is using an in-memory *SQLite3* database as its external DB, so what we test here is mostly the communication 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 93), using a virtual or direct call won't affect the CRUD operation speed: it will by-pass the virtual engine whenever possible.



### 1.4.3. Client-Server

#### 1.4.3.1. Involved technologies

Before describing the Client-Server design of this framework, we may have to detail some standards it is based on:

- JSON as its data transmission format;
- REST as its Client-Server architecture.

##### 1.4.3.1.1. JSON

###### 1.4.3.1.1.1. 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.

Usage of this layout, instead of other like XML or any proprietary format, results in several particularities:

- Like XML, it's 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) application;
- The JSON format is specified in this RFC
- 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 with other fields, see below (page 128) (only exception is *dynamic array* fields).

###### 1.4.3.1.1.2. JSON format density

Most common RESTful JSON used a verbose format for the JSON content: see for example [http://bitworking.org/news/restful\\_json..](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:

```
[{"ID":1}, {"ID":2}, {"ID":3}, {"ID":4}]
```

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 *Synapse mORMot Framework* always returns the JSON content just as a pure response of a SQL query, with an array and field names.

#### 1.4.3.1.1.3. JSON format 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 JavaScript because all browser communicate via HTTP!

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.

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 transferred as shorter:

```
{"fieldCount":1,"values":["ID",1,2,3,4,5,6,7]}
```

#### 1.4.3.1.1.4. JSON global cache

A global cache is used to enhance the framework scaling, and will use JSON 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.

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 independent from the authentication scheme, which pollutes the URI. Associated with the other levels of cache - see *ORM Cache* (page 84) - the framework scaling was found to be very good.

#### 1.4.3.1.2. REST

##### 1.4.3.1.2.1. RESTful implementation

*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.

The *Synapse 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 *SQLite3Http* unit and the *TSQLite3HttpServer* and *TSQLite3HttpClient* classes), in an embedded low resource and fast HTTP server.

The standard RESTful methods are implemented:

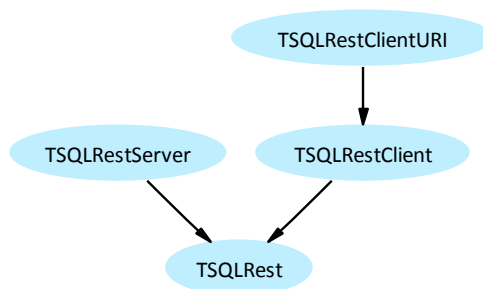
- GET to list the members of the collection;
- PUT to update a member of the collection;
- POST to create a new entry in the collection;
- DELETE to delete a member of the collection.

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..>

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.



*TSQLRestClient classes hierarchy*

This diagram states how the TSQLRest class implements a common ancestor for both Client and Server classes.

#### 1.4.3.1.2.2. REST and BLOB fields

BLOB fields are defined as TSQLRawBlob published properties in the classes definition. 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/ID/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 transfered as full binary.

Some dedicated methods of the generic TSQLRest class handle BLOB fields: RetrieveBlob and UpdateBlob.

#### 1.4.3.1.2.3. REST and JSON

The 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:

- 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:

TSQlite3HttpServer 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 within less than 400 KB.

#### 1.4.3.1.2.4. REST is Stateless

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.

##### 1.4.3.1.2.4.1. Server side synchronization

Even if REST is stateless, 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:

```
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: integer): boolean of object;

  TSQLRestServer = class(TSQLRest)
  (...)
    /// a method can be specified here to trigger events after any table update
    OnUpdateEvent: TNotifySQLEvent;
```

##### 1.4.3.1.2.4.2. 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:

```
/// 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:

```
/// 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):

```
procedure TMainForm.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.

### 1.4.3.1.3. Interfaces

#### 1.4.3.1.3.1. Delphi and interfaces

##### 1.4.3.1.3.1.1. 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:

```
type  
    ICalculator = interface(IInvokable)  
    ['{9A60C8ED-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.

#### 1.4.3.1.3.1.2. 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:

```
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".

#### 1.4.3.1.3.1.3. 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:

```
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  
  ICalculator := 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.

#### 1.4.3.1.3.1.4. 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)  
  protected  
    function Add(n1,n2: integer): integer;  
  end;  
  
function TOtherServiceCalculator.Add(n1, n2: integer): integer;  
begin  
  result := n2+n1;  
end;
```

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:

```
function MyOtherAdd(a,b: integer): integer;  
var Calculator: ICalculator;  
begin  
  ICalculator := TOtherServiceCalculator.Create;  
  result := Calculator.Add(a,b);  
end;
```

#### 1.4.3.1.3.1.5. 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.

#### 1.4.3.1.3.2. 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.

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 would 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.

#### 1.4.3.1.3.2.1. 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 45).

For instance, a `TRectangle` object should not have both `ComputeArea` and `Draw` methods defined at once - they would 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.

When you define an ORM object, do not put 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 won't 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.

Another example is how our database classes are defined in `SynDB.pas` - see *External database access* (page 112):

- 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.

Following this *Single responsibility principle* may sound simple and easy, but in fact, it is one of the

hardest principles to get right. Naturally, we tend to join responsibilities in our class definitions. Our ORM architecture will enforce you, by its Client-Server nature, to follow this principle, but it is always up to the end coder to design properly his/her interfaces.

#### 1.4.3.1.3.2.2. 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*.

When designing our ORM, 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 <http://synopse.info..> web site, and most of the framework enhancements have come from user requests.

The framework Open Source license - see below (page 206) - 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.

Furthermore, this principle will ensure your code to be ready to follow the main framework updates (which are quite regular). When a new version is available, you would be able to retrieve it for free from our web site, replace your files locally, then build a new enhanced version of your application. Even the source code repository is available - at <http://synopse.info/fossil..> - and allows you to follow the current step of evolution of the framework.

In short, abstraction is the key. All your code shall not depend on a particular implementation.

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 below (page 141);
- All object members shall be declared *private* or *protected* - this is a good idea to use *Service-oriented architecture* (page 45) 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.

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.

#### 1.4.3.1.3.2.3. Liskov substitution principle

Even if her name is barely unmemorable, *Barbara Liskov* is a great computer scientist, we should better learn from.

Her "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.).

For our framework, it would signify that `TSQLRestServer` or `TSQLRestClient` instances can be substituted to a `TSQLRest` object. Most ORM methods expect a `TSQLRest` parameter to be supplied.

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.

The main advantages of this coding pattern are the following:

- Thanks to this principle, you will be for instance able to *stub* or *mock* an interface or a class - this principle is therefore mandatory for implementing unitary testing to your project;
- Furthermore, testing would 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;
- If this principle is violated, the open/close principle will be - the parent class would need to be modified whenever a new derivative of the base class is defined;
- 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.

Some patterns which shall not appear in your code:

- Statements like `if aObject is TAClass then begin .... end else if aObject is TAnotherClass then ...` in a parent method;
- Use an enumerated item and a case ... of or nested if ... then to change a method behavior (this will also probably break the single responsibility principle: each enumeration shall be defined as a class);
- Define a method which will stay abstract for some children;
- Need to explicitly add all child classes units to the parent class unit uses clause.

In order to fulfill this principle, you should:

- 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*).

The SOA and ORM concepts as used by our framework are compatible with the Liskov substitution principle.

Furthermore, a more direct *Design by Contract* implementation pattern is also available (involving a more wide usage of interfaces).

#### **1.4.3.1.3.2.4. 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.

Beginning with revision 1.16, our framework allows direct use of interfaces to implement services. This great Client-Server SOA implementation pattern - see below (page 163) - 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.

#### **1.4.3.1.3.2.5. 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.

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's substitution principle.

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.

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 facilitated this decoupling pattern, and allows the use of native Delphi interface to call services from an abstract factory.

#### **1.4.3.1.3.3. Circular reference and (zeroing) weak pointers**

##### **1.4.3.1.3.3.1. 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 130), a SOA framework like *mORMot* has to offer so-called *Weak pointers* and *Zeroing Weak pointers* features.

By default in Delphi, all references are either:

- *weak references* for pointer and class instances;
- explicit copy for low-level value types like integer, `Int64`, currency, double or record (and old deprecated object or shortstring);
- *copy-on-write* with *reference counting* for high-level value types (e.g. string, widestring, variant or a *dynamic array*);
- *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 will not 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:

```
IParent = interface
  procedure SetChild(const Value: IChild);
  function GetChild: IChild;
  function HasChild: boolean;
  property Child: IChild read GetChild write SetChild;
end;

IChild = interface
  procedure SetParent(const Value: IParent);
  function GetParent: IParent;
  property Parent: IParent read GetParent write SetParent;
end;
```

The following implementation will definitively leak memory:

```
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.

#### 1.4.3.1.3.3.2. Handling weak pointers

In order to easily create a weak pointer, the following function was added to `SQLite3Commons.pas`:

```
procedure SetWeak(aInterfaceField: PInterface; const aValue: IInterface);
begin
  PPointer(aInterfaceField)^ := Pointer(aValue);
end;
```

It will assign the `interface` 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:

```
procedure TParent.SetChild(const Value: IChild);
begin
  SetWeak(@FChild, Value);
end;

procedure TChild.SetParent(const Value: IParent);
begin
  SetWeak(@FParent, Value);
end;
```

#### 1.4.3.1.3.3.3. 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 `SQLite3Commons.pas`:

```
procedure SetWeakZero(aObject: TObject; aObjectInterfaceField: PIInterface;  
  const aValue: IInterface);
```

A potential use case could be:

```
procedure TParent.SetChild(const Value: IChild);  
begin  
  SetWeakZero(self,@FChild,Value);  
end;  
  
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:

```
procedure TParent.SetChild(const Value: IChild);  
begin  
  SetWeak0(@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:

```
function TParent.HasChild: boolean;  
begin  
  result := FChild<>nil;  
end;  
  
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.

#### 1.4.3.1.3.3.4. 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..> 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/PostId/410/WeakRefence-in-Delphi-solving-circular-interfac.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..> 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 `SQLite3Commons.pas` for the details.

#### 1.4.3.2. Client-Server implementation

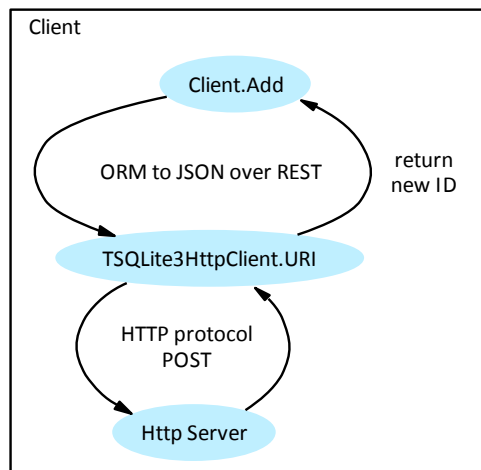
The framework can be used either stand-alone, either in a Client-Server model, via several communication layers:

- Fast in-process access (an executable file using a common library, for instance);
- GDI messages, only locally on the same computer, which is very fast;
- 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.

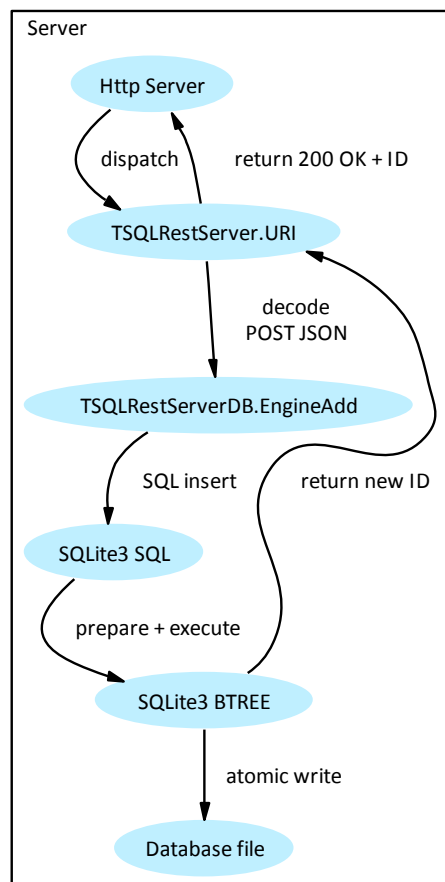
See *General mORMot architecture - Client / Server* (page 50) about this Client-Server architecture.

##### 1.4.3.2.1. Implementation design

A typical Client-Server RESTful POST / Add request over HTTP/1.1 will be implemented as such, on both Client and Server side:



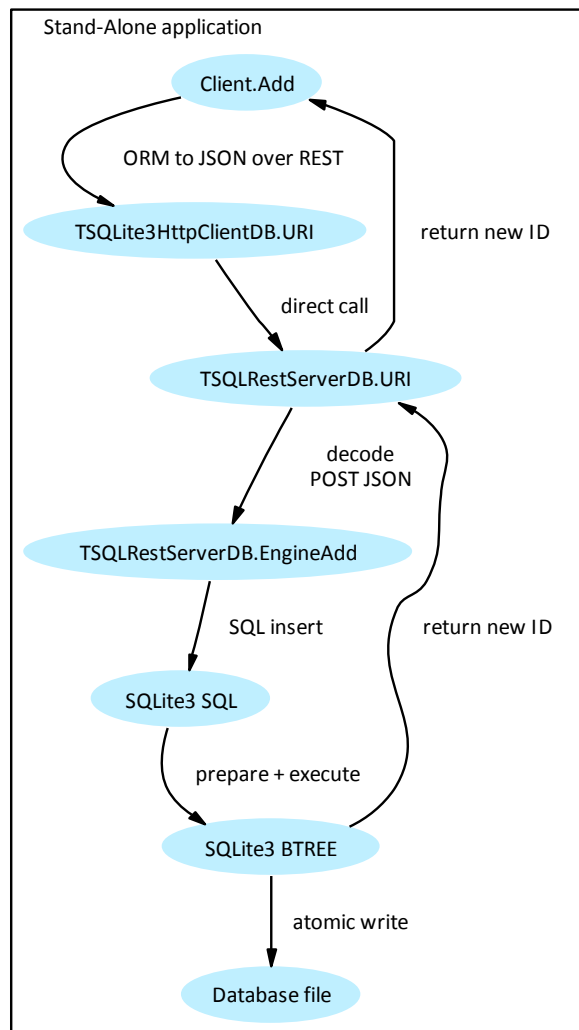
*Client-Server implementation - Client side*



*Client-Server implementation - Server side*

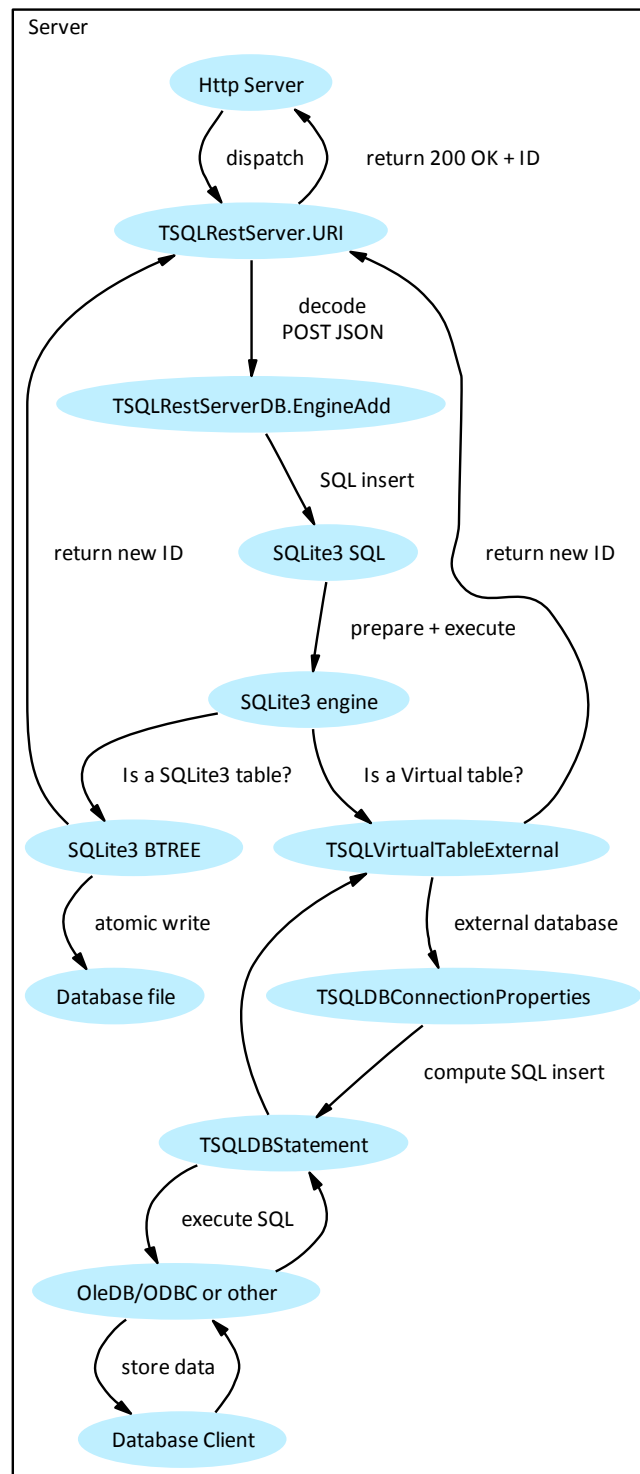
Of course, several clients can access to the same server.

It's possible to by-pass the whole Client-Server architecture, and let the application be stand-alone, by defining a **TSQRestClientDB** class, which will embed a **TSQRestServerDB** instance in the same executable:



*Client-Server implementation - Stand-Alone application*

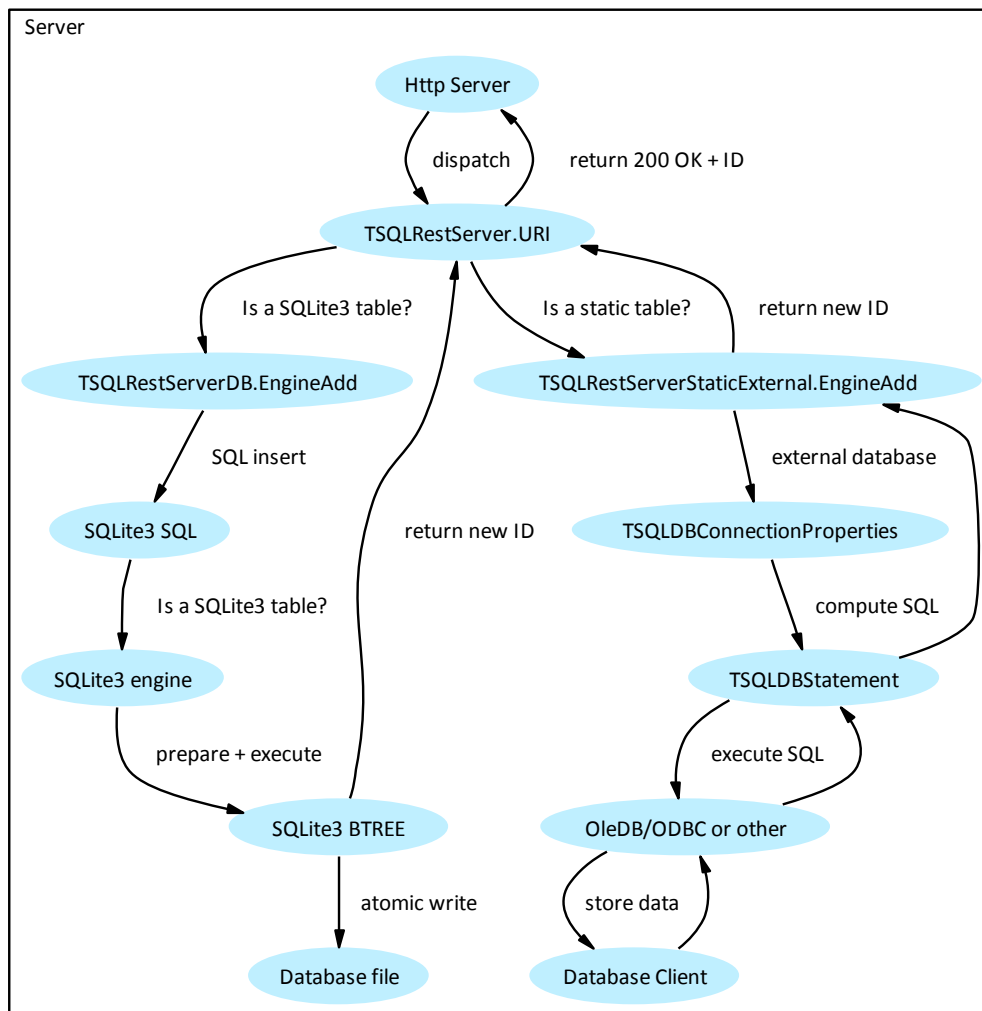
In case of a Virtual Table use (either in-memory or for accessing an external database), the client side remains identical. Only the server side is modified as such:



*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.

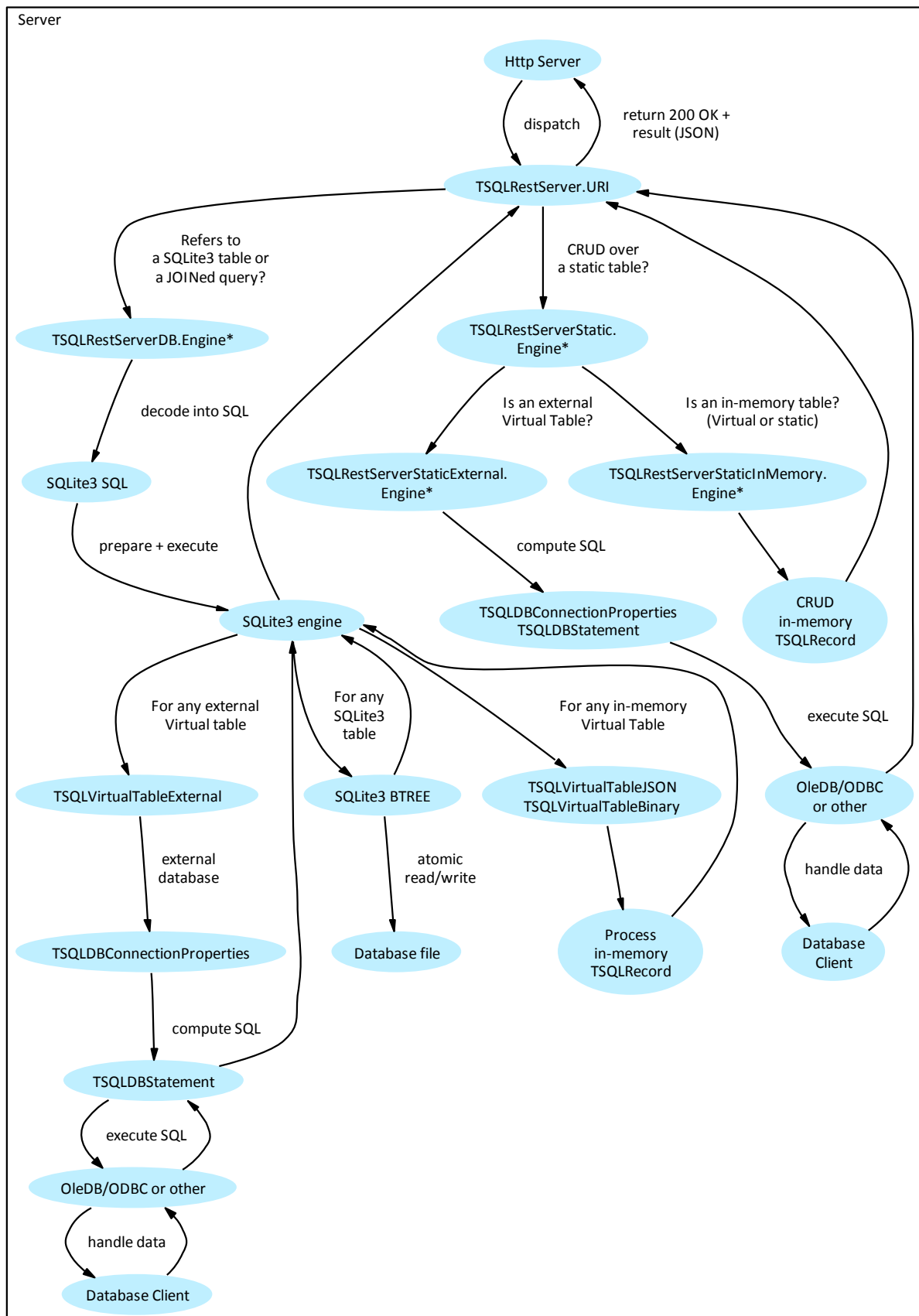
Most of the time, i.e. for RESTful / CRUD commands, the execution is more direct:



*Client-Server implementation - Server side with "static" Virtual Tables*

As stated in *External database access* (page 112), the static `TSQLRestServerStaticExternal` 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 151).

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.

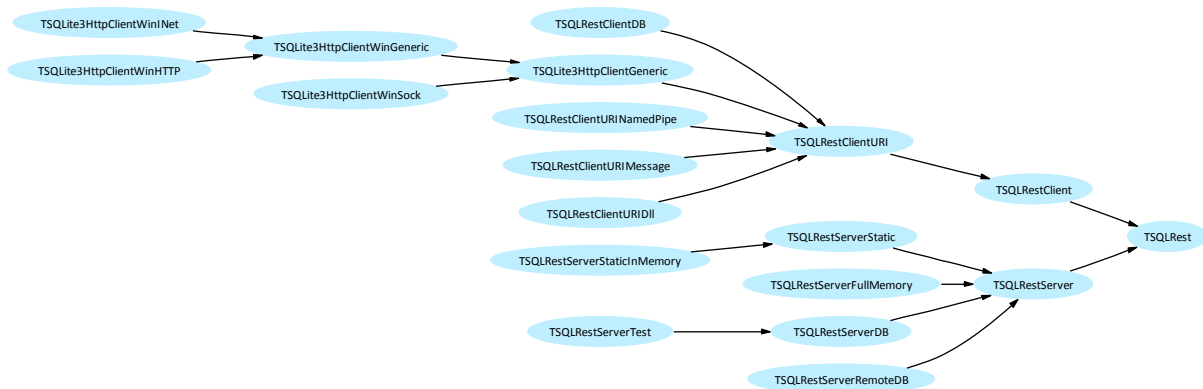


#### Client-Server implementation - Server side

You will find out some speed numbers resulting from this unique architecture in the supplied *Data access benchmark* (page 93).

#### 1.4.3.2.2. Client-Server classes

This architecture is implemented by a hierarchy of classes, implementing the RESTful pattern - see *REST* (page 127) - for either stand-alone, client or server side, all inheriting from a TSQLRest common ancestor:



Client-Server RESTful classes

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 database in the same process, with no speed penalty.

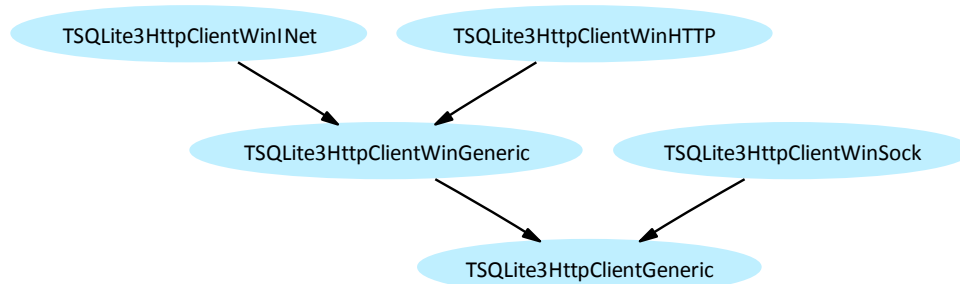
For a Client-Server application, create a TSQLRestServerDB instance, then use the corresponding ExportServer, ExportServerNamedPipe, ExportServerMessage method to instantiate either a in-process, Named-Pipe or GDI message server. For HTTP/1.1 over TCP/IP, creates a TSQLite3HttpServer instance, and associate your running TSQLRestServerDB to it. Then create either a TSQLite3HttpClient, TSQLRestClientURIDll, TSQLRestClientURINamedPipe or TSQLRestClientURIMessage instance to access to your data according to the communication protocol used for the server.

In practice, in order to implement the business logic, you should better create a new class, inheriting from TSQLRestServerDB. 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 TSQLRestServerStaticInMemory 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).

All those classes will implement a RESTful access to a remote database, with associated services and business logic.

#### 1.4.3.2.3. HTTP client

In fact, there are several implementation of a HTTP/1.1 client, according to this class hierarchy:

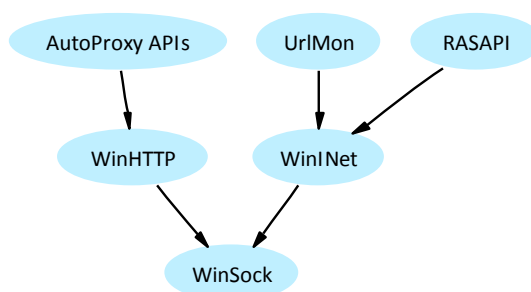


*HTTP/1.1 Client RESTful classes*

So you can select either `TSQlite3HttpClientWinSock`, `TSQlite3HttpClientWinINet` or `TSQlite3HttpClientWinHTTP` for a HTTP/1.1 client.

Each class has its own architecture, and attaches itself to a Windows communication library, all based on *WinSock* API. As stated by their name, `TSQlite3HttpClientWinSock` will call directly the *WinSock* API, `TSQlite3HttpClientWinINet` will call *WinINet* API (as used by IE 6) and `TSQlite3HttpClientWinHTTP` 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 those solutions:

Criteria	WinSock	WinINet	WinHTTP
API Level	Low	High	Medium
Local speed	Fastest	Slow	Fast



Network speed	Slow	Medium	Fast
Minimum OS	Win95/98	Win95/98	Win2000
HTTPS	Not available	Available	Available
Integration with IE	None	Excellent (proxy)	Available (see below)
User interactivity	None	Excellent (authentication, dial-up)	None

As stated above, there is still a potential performance issue to use the direct `TSQLite3HttpClientWinSock` class over a network. It has been reported on our forum, and root cause was not identified yet.

Therefore, the `TSQLite3HttpClient` class maps by default to the `TSQLite3HttpClientWinHTTP` 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, 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.

#### 1.4.3.2.4. HTTP server using `http.sys`

##### 1.4.3.2.4.1. Presentation

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 *Synapse 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.

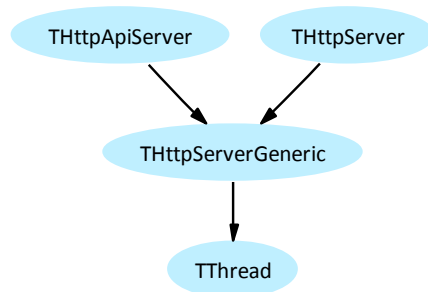
What's good about `https.sys`?

- *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.

All is encapsulated into a single class, named `THttpApiServer`. It 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:

```
TOnHttpRequest = function(  
  const InURL, InMethod, InContent, InContentType: TSocketData;  
  out OutContent, OutContentType, OutCustomHeader: TSocketData): cardinal of object;
```

This event handler prototype is shared by both TThread classes instances able to implement a HTTP/1.1 server:



*HTTP Server classes hierarchy*

When the Two steps are performed:

- The HTTP Server API is first initialized (if needed) during THttpApiServer.Create constructor call. The HttpApi.dll 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.
- We then register some URI matching the RESTful model - REST (page 127) - via the THttpApiServer.AddUrl 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 TSQLModel.Root, if you need it.

If any of the two first point fails (e.g. if http.sys is not available, or if it was not possible to register the URLs), our framework will fall back into using our 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.

Our THttpApiServer class will then receive the request, and pass it to the TSQLRestServer instance matching the incoming URI request, via the THttpApiServer.OnRequest event handler.

All JSON content will be processed, and a response will be retrieved from the internal cache of the framework, or computed using the SQLite3 database engine.

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...

And http.sys will handle all the communication by itself, leaving the server free to process the next request.

#### 1.4.3.2.4.2. UAC and Vista/Seven support

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.

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 for instance part of your Setup program. Then when your server application will be launched (for instance, as a background Windows service), it will be able to register all needed URL.

Nice and easy.

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:

```
program TestSQL3Register;
uses
  SynCrtSock,
  SysUtils;

// force elevation to Administrator under Vista/Seven
{$R VistaAdm.res}

begin
  THttpApiServer.AddUrlAuthorize('root','888',false,'+');
end.
```

#### 1.4.3.2.5. BATCH sequences for adding/updating/deleting records

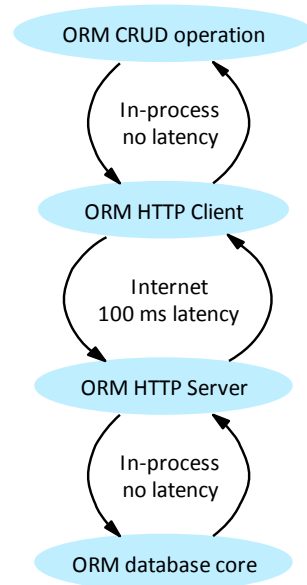
##### 1.4.3.2.5.1. BATCH process

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, then 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  $100 \times 1000 \text{ ms} = 100 \text{ s} = 1:40 \text{ min}$  just because of this network latency!



*BATCH mode Client-Server 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 `TSQRestClientURI` 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 integer.

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. It's up to the caller to use a `TransactionBegin` ... `try`... `Commit` except `RollBack` block.

Here is a typical use (extracted from the regression tests in `SQLite3.pas`):

```
// start the transaction
if ClientDist.TransactionBegin(TSQLRecordPeople) then
try
  // start the BATCH sequence
  Check(ClientDist.BatchStart(TSQLRecordPeople));
  // 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 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 n-1 do
    Check(not ClientDist.Retrieve(IntArray[i],V), 'BatchDelete');
    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;
        // in case of success, apply the Transaction
        ClientDist.Commit;
    except
        // In case of error, rollback the Transaction
        ClientDist.Rollback;
    end;
end;

```

In the above code, all CRUD operations are performed as usual, using Batch\*() methods instead of plain Add / Delete / Update. The ORM will take care of all internal process, including serialization.

#### 1.4.3.2.5.2. Implementation details

As described above, all Batch\*() methods are serialized as JSON on the client side, then sent as once to the server, where it will be processed without any client-server *round-trip* and slow latency.

Here is a typical JSON stream 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€ mÃ£Â g","LastName":"New","YearOfBirth":1000,
    "YearOfDeath":1519},
    "POST":{"FirstName":"@â€|,KAÃ%Ã #Ã¶f","LastName":"New","YearOfBirth":1001,"YearOfDeath":1519},
    (...) all POST = add actions
    "POST":{"FirstName":"+Ã¶tqCXW3Ã,\"","LastName":"New","YearOfBirth":2000,"YearOfDeath":1519}
    ]}

```

Here is a typical JSON stream receiver from the server, on success:

```
[200,200,...]
```

All the JSON generation (client-side) and parsing (server-side) is very optimized and fast. With the new internal *SynLZ* compression (available by default in our HTTP Client-Server classes), used bandwidth is minimal.

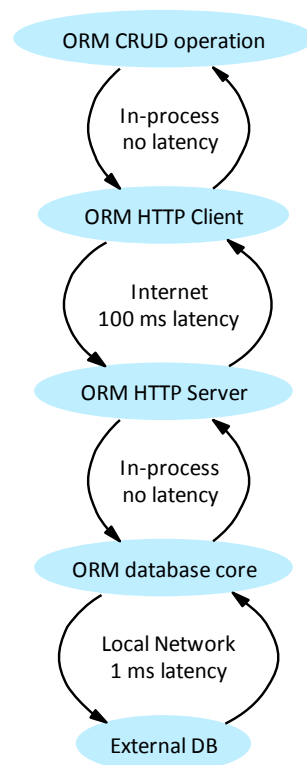
Thanks to these methods, most time is now spent into the database engine itself, and not in the communication layer.

Beginning with revision 1.16 of the framework, 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*). 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:

```
// 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;
  (...)
```

#### 1.4.3.2.5.3. Array binding

When used in conjunction with *External database access* (page 112), BATCH methods can be implemented as *array binding* if the corresponding *TSQLDBConnection* implementation implements the feature (only *SynDBOracle* unit has it by now). In fact, when using a remote database on a physical network, you won't be able to achieve more than 500-600 requests per second when performing INSERT, DELETE or UPDATE statements: the same round-trip occurs, this time between the ORM server side and the external Database engine.



*BATCH mode latency issue on external DB*

Of course, this 1 ms latency due to the external database additional round-trip may sounds negligible, but in *Service-oriented architecture* (page 45) when most process is done on the server side, it may introduce a huge performance difference. Your customers would not understand why using a *SQLite3* engine would be much faster than a dedicated *Oracle* instance they do pay for.

Our SynDB unit has been enhanced to introduce new `TSQLDBStatement.BindArray()` methods, introducing *array binding* for faster database batch modifications. It is working in conjunction with our BATCH methods, so CRUD modification actions are grouped within one *round-trip* over the network.

Thanks to this enhancement, inserting records within *Oracle* comes from 400-500 rows per second to more than 50000 rows per second, in our benchmarks. See *Data access benchmark* (page 93) and for details and performance charts the article at <http://blog.synopse.info/post/2012/07/19/Oracle-Array-Binding-and-BATCH-performance...>

In fact, some modern database engine (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 with statements within a loop.

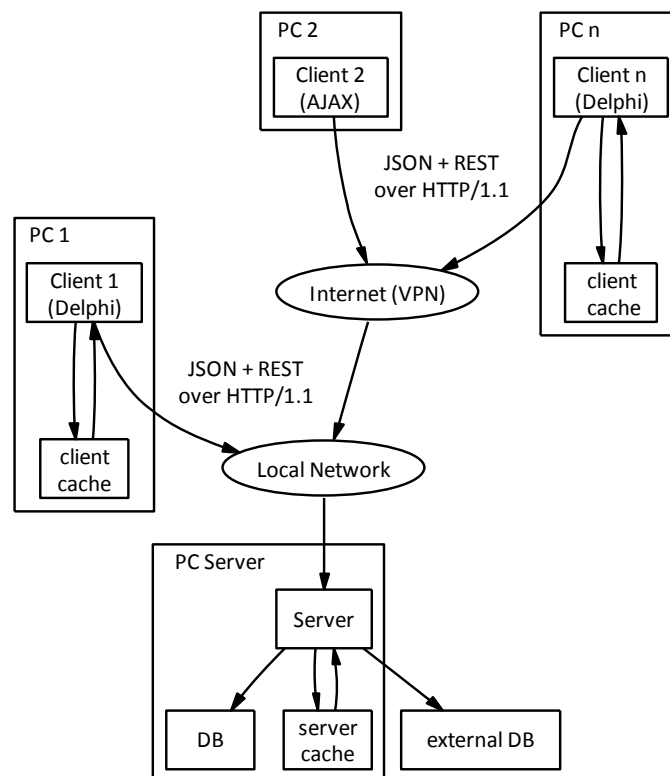
If you want to use a *map/reduce* algorithm in your application, in addition to ORM data access, all those enhancements may speed up a lot your process. Reading and writing huge amount of data has never been so fast and easy: you may even be tempted to replace stored-procedure process by high-level code implemented in your Domain service. N-tier separation would benefit from it.

#### 1.4.3.2.6. CRUD level cache

#### 1.4.3.2.6.1. Where to 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 84) for the other cache patterns available in the framework.

In fact, a unique caching mechanism is shared 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.



CRUD caching in mORMot

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 would 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.



#### 1.4.3.2.6.2. 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 *REST is Stateless* (page 129) 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.

#### 1.4.3.2.6.3. What to cache

A 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 219) - 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.

#### 1.4.3.2.6.4. How to cache

A dedicated `TSQLRestCache` instance can be created, and will maintain such a tuned caching mechanism, for both `TSQLRestClient` and `TSQLRestServer` classes.

A call to `TSQLRest.Cache's SetCache()` and `SetTimeOut()` methods is enough to specify which table(s) or record(s) are to be cached, either at the client or the server level.

For instance, here is how the Client-side caching is tested about one individual record:

```
(...)  
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;
(...)
Database.Cache.SetCache(TSQLRecordPeople); // server-side
(...)
```

Note that 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.

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.

*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.

#### 1.4.3.2.7. Server side process (aka stored procedure)

The framework is able to handle a custom type of "stored procedure" in pure Delphi code, like any powerful Client-Server solution.

In short, a *stored procedure* is a way of moving some data-intensive process on the server 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 will be much faster than a pure client-sided solution.

According to the current state of our framework, there are several ways of handling such a server-side *stored procedure* process:

- Write your own SQL function to be used in `SQLite3` `WHERE` statements;
- Low-level dedicated Delphi stored procedures;
- Define some below (page 163), as `TSQLRestServer` published methods (in fact, such a service can have fast direct access to the database on the Server side, so can be identified as a kind of stored procedure).

The Server-Side service appears to be the more RESTful compatible way of implementing a stored procedure mechanism in our framework. But custom SQL may help the client code to be more generic, and particular queries to be more easily written.

#### 1.4.3.2.7.1. 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 101);
- 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 either 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 67).

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:

```
UPDATE MyTable SET SomeField=0 WHERE IntegerDynArrayContains(IntArrayField,:(10):)
```

##### 1.4.3.2.7.1.1. Implementing a function

Let us implement a `CharIndex()` SQL function, defined as such:

```
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_*(Context,*)` functions.

Here is a 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):

```
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,0) else  
  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[])`.

#### 1.4.3.2.7.1.2. Registering a function

##### 1.4.3.2.7.1.3. Direct low-level SQLite3 registration

Since we have a `InternalSQLFunctionCharIndex()` function defined, we may register it with direct *SQLite3* API calls, as such:

```
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.

##### 1.4.3.2.7.1.4. Class-driven registration

It's 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:

```
Demo.RegisterSQLFunction(InternalSQLFunctionCharIndex,2,'CharIndex');  
Demo.RegisterSQLFunction(InternalSQLFunctionCharIndex,3,'CharIndex');
```

The two lines above will indeed wrap the following code:

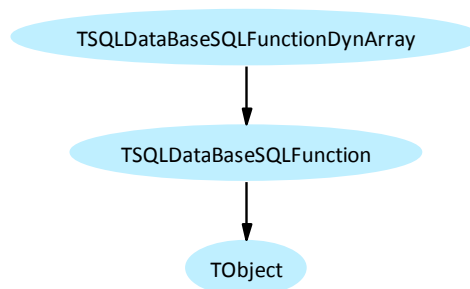
```
Demo.RegisterSQLFunction(TSQLDataBaseSQLFunction.Create(InternalSQLFunctionCharIndex,2,'CharIndex'  
));
```

```
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.

#### 1.4.3.2.7.1.5. 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*

For instance, the following method will register a SQL function able to search into a BLOB-stored custom dynamic array type:

```
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:

```
/// 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);
begin
  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(Ints, :("%"))'),
  [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:

```
aClient.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.



#### 1.4.3.2.7.2. Low-level Delphi stored procedure

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. Code inside this event handler should not use the ORM methods of the framework, but direct low-level *SQLite* access. This event handler should be specified to the corresponding overloaded `TSQLRestServerDB.EngineExecute` method.

This will allow direct content modification during the `SELECT` statement. Be aware that, up to now, *Virtual Tables magic* (page 103) `TSQLVirtualTableCursorJSON` cursors are not safe to be used if the Virtual Table data is modified.

See the description of those event `TOnSQLStoredProc` handler and associated method in the next pages.

#### 1.4.3.2.7.3. External stored procedure

If the application relies on external databases - see *External database access* (page 112) - 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 format (.Net, Java, P/SQL or whatever), then define some below (page 163) to launch those functions. Note that in this case, you'll loose the database-independence of the framework, and switching to another database engine may cost.

#### 1.4.3.3. Client-Server services

In order to implement *Service-oriented architecture* (page 45) in our framework, the business logic can be implemented in two ways in the framework:

- Via some `TSQLRecord` inherited classes, inserted into the database *model*, and accessible via some RESTful URI - this will implement ORM architecture;
- By some RESTful services, implemented in the Server as *published methods*, and consumed in the Client via native Delphi methods;
- Defining some *service contracts* as standard Delphi interface, and then run it seamlessly on both client and client sides.

The first of the two last items can be compared to the *DataSnap* Client-Server features, also JSON-based, introduced in Delphi 2010. See for instance the following example available on the Internet at [http://docwiki.embarcadero.com/RADStudio/en/Developing\\_DataSnap\\_Applications..](http://docwiki.embarcadero.com/RADStudio/en/Developing_DataSnap_Applications..)

The second is purely interface-based, so matches the "designed by contract" principle - see *SOLID design principles* (page 133) - as implemented by Microsoft's WCF technology. *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...> We included most of the nice features made available in WCF in *mORMot*, in a KISS manner.

##### 1.4.3.3.1. 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?

#### 1.4.3.3.1.1. My Server is rich

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:

```
type
  TSQLRestServerTest = class(TSQLRestServerFullMemory)
    (...)
    published
      function Sum(var aParams: TSQLRestServerCallBackParams): Integer;
    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:

```
type
  TSQLRestServerCallBack = function(var aParams: TSQLRestServerCallBackParams): Integer of object;
```

Then we implement this method:

```
function TSQLRestServerTest.Sum(var aParams: TSQLRestServerCallBackParams): Integer;
var a,b: Extended;
begin
  if not UrlDecodeNeedParameters(aParams.Parameters,'A,B') then
    begin
      result := 404; // invalid Request
      aParams.ErrorMessage := 'Missing Parameter';
      exit;
    end;
  while aParameters<>nil do
    begin
      UrlDecodeExtended(aParams.Parameters,'A=',a);
      UrlDecodeExtended(aParams.Parameters,'B=',b,@aParams.Parameters);
    end;
  aParams.Resp := JSONEncodeResult([a+b]);
  // same as : aResp := JSONEncode(['result',a+b],TempMemoryStream);
  result := 200; // success
end;
```

The only not obvious part of this code is the parameters marshaling, i.e. how the values are retrieved from the incoming `aParams.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 `aParams.Parameters`;
- Call `UrlDecodeInteger` / `UrlDecodeInt64` / `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 into `aParams.Resp` variable.

The powerful `UrlDecodeObject` function (defined in `SQLite3Commons.pas`) can be used to unserialize most class instance from its textual JSON representation (`TPersistent`, `TSQLRecord`, `TStringList`...).

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.

The implementation must return the HTTP error code (e.g. 200 on success) as an integer value, and any response in `aParams.Resp` as a serialized JSON object by default (using e.g. `TSQLRestServer.JSONEncodeResult`), since default mime-type is `JSON_CONTENT_TYPE`:

```
{"Result": "OneValue"}
```

or a JSON object containing an array:

```
{"Result": ["One", "two"]}
```

So you can consume these services, implemented Server-Side in fast Delphi code, with any AJAX application on the client side (if you use HTTP as communication protocol).

The `aParams.Head^` parameter may be overridden on the server side to set a custom header which will be provided to the client - it may be useful for instance 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.

In case of an error on the server side (only two valid status codes are 200 and 201), the client will receive a corresponding serialized JSON error object, as such:

```
{  
  "ErrorCode": 404,  
  "ErrorText": "Missing Parameter"  
}
```

The `aParams.ErrorMessage^` parameter can be overridden on the server side 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 `ErrorMessage` is specified, the framework will return the corresponding generic HTTP status text.

The `aParams.Context` parameter may contain at calling time the expected `TSQLRecord` ID (as decoded from *RESTful* URI), and the current session, user and group IDs. If authentication - see below (page 195) - is not used, this parameter is meaningless: in fact, `aParams.Context.Session` will contain either 0 if any session is not yet started, or 1 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).

An *important point* is to remember that the implementation of the callback method **must be thread-safe** - as stated by below (page 200). 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.

#### 1.4.3.3.1.2. The Client is always right

The client-side is implemented by calling some dedicated methods, and providing the service name

('sum') and its associated parameters:

```
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:

```
type
  TMyClient = class(TSQLite3HttpClient) // could be TSQLRestClientURINamedPipe
  (...)
  function Sum(a, b: double): double;
  (...)

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 UriEncode() function. Note that you can specify most class instance into its JSON representation by using some TObject into the method arguments:

```
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, but you can serve any kind of data, binary, HTML, whatever... just by overriding the content type on the server.

The usual protocols of our framework can be used: HTTP/1.1, Named Pipe, Windows GDI messages, direct in-memory/in-process access.

Of course, these services can be related to any table/class of our ORM framework, so you would be able to create easily any RESTful compatible requests on URI like ModelRoot/Table/Name/ID/MethodName. The ID of the corresponding record is decoded from its RESTful scheme into aParams.Context.ID. For example, here we return a BLOB field content as hexadecimal:

```
function TSQLRestServerTest.DataAsHex(var aParams: TSQLRestServerCallBackParams): Integer;
var aData: TSQLRawBlob;
begin
  result := 404; // invalid Request
  if (self=nil) or (aParams.Table=nil) or
    not aParams.Table.InheritsFrom(TSQLRecord) or
    (aParams.Context.ID<0) then
    exit; // we need a valid record and its ID
  if not RetrieveBlob(TSQLRecordPeople, aParams.Context.ID, 'Data', aData) then
    exit; // impossible to retrieve the Data BLOB field
  aParams.Resp := JSONEncodeResult([SynCommons.BinToHex(aData)]);
  // idem: aResp := JSONEncode(['result', BinToHex(aRecord.fData)], TempMemoryStream);
  result := 200; // success
```

end;

#### 1.4.3.3.2. Interface based services

The *Client-Server services via methods* (page 163) implementation gives full access to the lowest-level of the *mORMot*'s 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.

But this implementation pattern has some drawbacks:

- Most content marshaling 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 marshaling twice, for both client and server;
- 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);
- Security is handled globally for the user, or should be checked by hand in the implementation method (using the *aParams.Context* values).

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, you can consult this classic article: <http://www.ibm.com/developerworks/webservices/library/ws-soa-design1..>

According to this document, all expected SOA features are now available in the current implementation of the *mORMot* framework (including service catalog aka "broker").

##### 1.4.3.3.2.1. Implemented features

Here are the key features of the current implementation of services using interfaces in the *Synapse mORMot* framework:

Feature	Remarks
Service Orientation	Allow loosely-coupled relationship
Design by contract	Data Contracts are defined in Delphi code as standard interface custom types
Factory driven	Get an implementation instance from a given interface
Server factory	You can get an implementation on the server side
Client factory	You can get a "fake" implementation on the client side, remotely calling the server to execute the process

Auto marshaling	The contract is transparently implemented: no additional code is needed e.g. on the client side, and will handle simple types (strings, numbers, dates, sets and enumerations) and high-level types (objects, collections, records, dynamic arrays) from Delphi 6 up to XE2
Flexible	Methods accept per-value or per-reference parameters
Instance lifetime	An implementation class can be: <ul style="list-style-type: none"> <li>- Created on every call,</li> <li>- Shared among all calls,</li> <li>- Shared for a particular user or group,</li> <li>- Stay alive as long as the client-side interface is not released,</li> <li>- or as long as an authentication session exists</li> </ul>
Stateless	Following a standard request/reply pattern
Signed	The contract is checked to be consistent before any remote execution
Secure	Every service and/or methods can be enabled or disabled on need
Safe	Using extended RESTful authentication - see below (page 195)
Multi-hosted (with DMZ)	Services are hosted by default within the main ORM server, but can have their own process, with a dedicated connection to the ORM core
Broker ready	Service meta-data can be optionally revealed by the server
Multiple transports	All Client-Server protocols of mORMot are available, i.e. direct in-process connection, GDI 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), either in a JSON-RPC model (the AJAX way)
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

#### 1.4.3.3.2.2. 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.

#### 1.4.3.3.2.3. Defining a data 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* (page 45) 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 would be easily re-coded with external code, using e.g. a SOAP/WSDL gateway).

#### 1.4.3.3.2.3.1. Define an interface

The data contract is to be defined as a plain Delphi interface type. In fact, the sample type as stated above - see *Interfaces* (page 130) - can be used directly:

```
type
  ICalculator = interface(IInvokable)
    ['{9A60C8ED-CEB2-4E09-87D4-4A16F496E5FE}']
    /// 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).

#### 1.4.3.3.2.3.2. Available types for methods parameters

Handled types of parameters are:

Delphi type	Remarks
boolean	Transmitted as JSON true/false
integer cardinal Int64 double currency TDateTime	Transmitted as JSON numbers
enumerations	Transmitted as JSON number
set	Transmitted as JSON number - one bit per element (up to 32 elements)
RawUTF8 WideString	Transmitted as JSON text (UTF-8 encoded)
string	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
TPersistent	Published properties will be transmitted as JSON object
TSQLRecord	All fields (including ID) will be transmitted as JSON object
any TObject	Via TJSONSerializer.RegisterCustomSerializer
TCollection	Not allowed: use TInterfacedCollection instead
TInterfacedCollection	Transmitted as a JSON array of JSON objects - see below (page 172)
dynamic arrays	Transmitted as JSON arrays - see below (page 212)
record	Transmitted as binary with Base-64 encoding or custom JSON serialization, needed to have RTTI (so a string or dynamic array field within), just like with regular Delphi interface expectations
TServiceCustomAnswer	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)

You can therefore define complex interface types, as such:

```
type
  ICalculator = interface(IInvokable)
    ['{9A60C8ED-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;  
end;
```

Note how `SpecialCall` and `ComplexCall` methods have quite complex parameters definitions, including dynamic arrays, sets and records. The framework will handle `const` and `var` parameters as expected, i.e. as input/output parameters, also on the client side. And 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.

#### 1.4.3.3.2.3.3. Custom JSON serialization of records

By default, any record parameter or function result will be serialized with our proprietary binary (and optimized layout) - i.e. `RecordLoad` and `RecordSave` functions - then encoded in Base-64, to be stored as plain text within the JSON stream.

But custom record JSON serialization can be defined, as with any class - see *Custom TObject JSON serialization* (page 71) - or *dynamic array* - see below (page 216).

In fact, there are two ways of specifying a custom JSON serialization for record:

- When setting a custom dynamic array JSON serializer, 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.

For instance, you can serialize the following record definition:

```
TSQLRestCacheEntryValue = record  
  ID: integer;  
  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 a `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 would be like:

```
class function TTestServiceOrientedArchitecture.CustomReader(P: PUTF8Char;  
  var aValue; out aValid: Boolean): PUTF8Char;  
var V: TSQLRestCacheEntryValue absolute aValue;  
  Values: TPUTF8CharDynArray;  
begin  
  result := JSONDecode(P,['ID','TimeStamp','JSON'],Values);  
  if result=nil then  
    aValid := false else begin  
      V.ID := GetInteger(Values[0]);  
      V.TimeStamp := GetCardinal(Values[1]);  
      V.JSON := Values[2];  
      aValid := true;  
    end;  
end;
```

Note that the callback signature used for *records* matches the one used for *dynamic arrays* serializations - see below (page 216) - as it will be shared between the two of them.

Even if older versions of *Delphi* are not able to generate the needed RTTI information for such serialization, the *mORMot* framework offers a common way of implementing any custom serialization of records.

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.

#### 1.4.3.2.3.4. InterfacedCollection kind of parameter

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:

```
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.

So a dedicated `TInterfacedCollection` abstract type has been defined:

```
TInterfacedCollection = class(TCollection)  
protected  
  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:

```
TCollTests = class(TInterfacedCollection)  
protected  
  function GetClass: TCollectionItemClass; override;  
end;
```



```
function TCollTests.GetClass: TCollectionItemClass;  
begin  
  result := TCollTest;  
end;
```

Or, if you want a more complete / convenient implementation:

```
TCollTests = class(TInterfacedCollection)  
private  
  function GetCollItem(Index: Integer): TCollTest;  
protected  
  function GetClass: TCollectionItemClass; override;  
public  
  function Add: TCollTest;  
  property Item[Index: Integer]: TCollTest read GetCollItem; default;  
end;
```

Then you will be able to define a contract as such:

```
procedure Collections(Item: TCollTest; var List: TCollTests; out Copy: TCollTests);
```

A typical implementation of this contract may be:

```
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.

#### 1.4.3.3.2.4. Server side

##### 1.4.3.3.2.4.1. Implementing 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 130) - 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.

#### 1.4.3.3.2.4.2. 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);
```

The Server instance can be any `TSQLRestServer` inherited class, implementing any of the supported protocol of *mORMot's Client-Server implementation* (page 141), 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 the `sicShared` 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).

#### 1.4.3.3.2.4.3. Instances life time implementation

The available instance management options are the following:

Lifetime	Description
<code>sicSingle</code>	One class 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 <code>TSQLRestServer.ServiceRegister</code> method.
<code>sicShared</code>	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
<code>sicClientDriven</code>	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 <code>nil</code> ), it will be released on the server side - a numerical identifier will be transmitted with all JSON requests

<code>sicPerSession</code>	One object instance will be maintained during the whole running session
<code>sicPerUser</code>	One object instance will be maintained and associated with the running user
<code>sicPerGroup</code>	One object instance will be maintained and associated with the running user's authorization group

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:

Lifetime	Use case
<code>sicSingle</code>	An asynchronous process (may be resource consuming)
<code>sicShared</code>	Either a very simple process, or requiring some global data
<code>sicClientDriven</code>	The best candidate to implement a Business Logic workflow
<code>sicPerSession</code>	To maintain some data specific to the client application
<code>sicPerUser</code>	Access to some data specific to one user
<code>sicPerGroup</code>	Access to some data shared by a user category (e.g. administrator, or guests)

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 (e.g. in `sicShared`, `sicPerUser` or `sicPerGroup` 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 53).

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):

```
type
  IComplexNumber = interface(IInvokable)
  ['{29D753B2-E7EF-41B3-B7C3-827FEB082DC1}']
  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:

```

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;
  fImaginary := fImaginary+aImaginary;
end;

procedure TServiceComplexNumber.Assign(aReal, aImaginary: double);
begin
  fReal := aReal;
  fImaginary := aImaginary;
end;

function TServiceComplexNumber.GetImaginary: double;
begin
  result := fImaginary;
end;

function TServiceComplexNumber.GetReal: double;
begin
  result := fReal;
end;

procedure TServiceComplexNumber.SetImaginary(const Value: double);
begin
  fImaginary := Value;
end;

procedure TServiceComplexNumber.SetReal(const Value: double);
begin
  fReal := Value;
end;

```

This interface is registered on the server side as such:

```
Server.ServiceRegister(TServiceComplexNumber,[TypeInfo(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.

When any 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
  Factory: TServiceFactoryServer;
  /// the currently running session identifier which launched the method
  // - make available the current session or authentication parameters
  // (including e.g. user details via Factory.RestServer.SessionGetUser)
  Session: ^TSQLRestServerSessionContext;
end;
```

When used, a local copy or a `PServiceRunningContext` pointer should better be created, since accessing a threadvar has a non negligible performance cost.

#### 1.4.3.3.2.4.4. 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* (page 45), 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 47)): 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 133), you'd better rely on abstraction in your code, i.e. not call the service implementation, 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, so is the right access point to all services):

```
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:

```
var I: ICalculator;
begin
  if Server.Services['Calculator'].Get(I) then
    result := I.Add(10,20);
end;
```

or, for a more complex service:

```
var CN: IComplexNumber;
begin
  if not Server.Services.Info(TypeInfo(IComplexNumber)).Get(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
```

You can of course cache your TServiceFactory instance within a local field, if you wish.

#### 1.4.3.3.2.5. 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*), nor client-side methods to write - as with our *Client-Server services via methods* (page 163).

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.

##### 1.4.3.3.2.5.1. Set up the Client factory

On the client side, you have to register the corresponding interface, as such:

```
Client.ServiceRegister([TypeInfo(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:

```
Client.ServiceRegister([TypeInfo(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.

##### 1.4.3.3.2.5.2. 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:

```
var I: ICalculator;  
begin  
  if Client.Services['Calculator'].Get(I) then  
    result := I.Add(10,20);  
end;
```

or, for a more complex service, initialized in *sicClientDriven*:

```
var CN: IComplexNumber;  
begin  
  if not Client.Services.Info(TypeInfo(IComplexNumber)).Get(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
```

You can of course cache your TServiceFactory instance within a local field, if you wish.

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.

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 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.

#### 1.4.3.3.2.6. 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.

##### 1.4.3.3.2.6.1. The shared contract

First, you'll find a common unit, shared by both client and server applications:

```
unit Project14Interface;  
  
interface  
  
type  
  ICalculator = interface(IInvokable)  
    ['{9A60C8ED-CEB2-4E09-87D4-4A16F496E5FE}']  
    function Add(n1,n2: integer): integer;  
  end;  
  
const  
  ROOT_NAME = 'service';  
  APPLICATION_NAME = 'RestService';  
  
implementation  
  
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.

##### 1.4.3.3.2.6.2. The server sample application

The server is implemented as such:

```
program Project14Server;  
  
{$APPTYPE CONSOLE}
```

```

uses
  SysUtils,
  SQLite3Commons,
  SQLite3,
  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
        ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared);
        if ExportServerNamedPipe(APPLICATION_NAME) then
          writeln('Background server is running.#10) else
          writeln('Error launching the server'#10);
        write('Press [Enter] to close the server. ');
        readln;
      finally
        Free;
      end;
    finally
      aModel.Free;
    end;
  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 ServiceRegister() 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
    ServiceRegister(TServiceCalculator,[TypeInfo(ICalculator)],sicShared);
    if ExportServerNamedPipe(APPLICATION_NAME) then
      (...)

```

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.



#### 1.4.3.3.2.6.3. 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);
  if err<>0 then begin
    edtA.SetFocus;
    exit;
  end;
  val(edtB.Text,b,err);
  if err<>0 then begin
    edtB.SetFocus;
    exit;
  end;
  if Client=nil then begin
    if Model=nil then
      Model := TSQLModel.Create([],ROOT_NAME);
      Client := TSQLRestClientURINamedPipe.Create(Model,APPLICATION_NAME);
      Client.SetUser('User','synopse');
      Client.ServiceRegister([TypeInfo(ICalculator)],sicShared);
    end;
    if Client.Services['Calculator'].Get(I) then
      lblResult.Caption := IntToStr(I.Add(a,b));
  end; // here local I will be released
```

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).

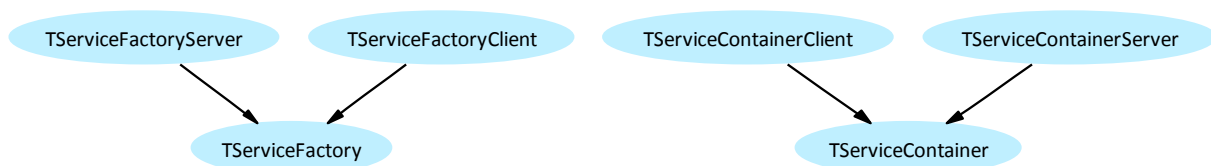
Once the client is up and ready, the local I: ICalculator variable instance is retrieved, and the remote service is called directly via a simple I.Add(a,b) statement.

You can imagine how easy and safe it will be to implement a *Service-oriented architecture* (page 45) for your future applications, using *mORMot*.

#### 1.4.3.3.2.7. Implementation details

##### 1.4.3.3.2.7.1. Involved classes

You will find out in SQLite3Commons.pas all classes implementing this interface communication.



*Services implementation classes hierarchy*

There are two levels of implementation:

- A *services catalog*, available in `TSQLRest.Services` property, declared as `TServiceContainer` (with two inherited versions, one for each side);
- A *service factory* for each interface, declared as `TServiceFactory` (also with two inherited versions, one for each side).

In fact, `TServiceFactory.Create` constructor will retrieve all needed RTTI information of the given interface, i.e. GUID, name and all methods (with their arguments). It will compute the low-level stack memory layout needed at execution to emulate a call of a native Delphi interface. And the corresponding "contract" will be computed from the signature of all interfaces and methods, to validate that both client and server expect the exact same content.

On the server side, `TServiceFactoryServer.ExecuteMethod` method (and then a nested `TServiceMethod.InternalExecute` call) is used to prepare a valid call to the implementation class code from a remote JSON request.

On the client side, a `TInterfacedObjectFake` class will be created, and will emulate a regular Delphi interface call using some on-the-fly asm code generated in the `TServiceFactoryClient.Create` constructor. For technical information about how interfaces are called in *Delphi*, see <http://sergworks.wordpress.com/2010/07/06/delphi-interfaces-on-binary-level..> and the `FakeCall` method implementation.

Here is the core of this client-side implementation of the "call stubs":

```
for i := 0 to fMethodsCount-1 do begin
  fFakeVTable[i+RESERVED_VTABLE_SLOTS] := P;
  P^ := $68ec8b55; inc(P);           // push ebp; mov ebp,esp
  P^ := i; inc(P);                   // push {MethodIndex}
  P^ := $e2895251; inc(P);           // push ecx; push edx; mov edx,esp
  PByte(P)^ := $e8; inc(PByte(P));   // call FakeCall
  P^ := PtrUInt(@TInterfacedObjectFake.FakeCall)-PtrUInt(P)-4; inc(P);
  P^ := $c25dec89; inc(P);           // mov esp,ebp; pop ebp
  P^ := fMethods[i].ArgsSizeInStack or $900000; // ret {StackSize}; nop
  inc(PByte(P),3);
end;
```

Just for fun... I could not resist posting this code here; if you are curious, take a look at the "official" `RTTI.pas` or `RIO.pas` units as provided by Embarcadero, and you will probably find out that the *mORMot* implementation is much easier to follow, and also faster (it does not recreate all the stubs or virtual tables at each call, for instance). :)

#### 1.4.3.3.2.7.2. Security

As stated above, in the features grid, a complete security pattern is available when using client-server services. In a *Service-oriented architecture* (page 45), securing messages between clients and services is essential to protecting data.

Security is implemented at several levels:

- For communication stream - e.g. when using HTTPS protocol at the *Client-Server implementation* (page 141), or a custom cypher within HTTP content-encoding;
- At RESTful / URI authentication level - see below (page 195); introducing *Group* and *User* notions;
- At interface or method (service/operation) level - we'll discuss this part now.

By default, all services and operations (i.e. all interfaces and methods) are allowed to execution.

Then, on the server side (it's an implementation detail), the `TServiceFactoryServer` instance (available from `TSQLRestServer.Services` property) 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 195), *mORMot* provides a versatile and inter-operable security pattern.

Here is some extract of the supplied regression tests:

```
(...)
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');
(...)
```

The Test() procedure is used to validate the corresponding methods of ICalculator (1=Add, 2=Multiply, 3=Subtract, 4=ToText...).

In this above code, the GroupID value is retrieved as such:

```
GroupID := fClient.MainFieldID(TSQLAuthGroup, 'User');
```

And the current authenticated user is member of the 'User' group:

```
fClient.SetUser('User', 'synapse'); // default user for Security tests
```

Since TSQLRestServer.ServiceRegister method returns 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.ServiceRegister(TServiceCalculator, [TypeInfo(ICalculator)], sicShared).
  DenyAll.AllowAllByName(['Supervisor']);
```

This will allow access to the ICalculator methods only for the *Supervisor* group of users.

#### 1.4.3.3.2.7.3. 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.

#### 1.4.3.3.2.7.4. Request format

As stated above, there are two mode of routing, defined by `TServiceRoutingMode`. The routing to be used is defined globally in the `TSQLRest.Routing` property.

	rmREST	rmJSON_RPC
Description	URI-based layout	JSON-RPC mode
Default	Yes	No
URI scheme	/Model/Interface.Method[/ClientDrivenID]	/Model/Interface
Body content	JSON array of parameters	{ "method": "MethodName", "params": [...] [, "id": ClientDrivenID] }
Security	RESTful authentication for each method	RESTful authentication for the whole service (interface)
Speed	10% faster	10% slower

In the default rmREST 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 a typical request for `ICalculator.Add`:

```
POST /root/Calculator.Add
(...)
[1,2]
```

Here we use a POST verb, but the framework will also allow GET, if needed (e.g. from a ). The pure Delphi client implementation will use only POST.

For a `sicClientDriven` mode service, the needed instance ID is appended to the URI:

```
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 using URI is that it will be more secure in our RESTful authentication scheme - see below (page 195): each method (and even any client driven session ID) will be signed properly.

In this rmREST 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
```

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.

If rmJSON\_RPC mode is used, the URI will define the interface, and then the method name will be inlined with parameters, e.g.

```
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:

```
POST /root/ComplexNumber
(...)
{"method":"Add","params":[20,30],"id":1234}
```

This mode will be a little bit slower, but will probably be more AJAX ready.

It's up to you to select the right routing scheme to be used.

#### 1.4.3.3.2.7.5. Response format

#### 1.4.3.3.2.7.6. Standard answer as JSON object

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],"id":0}
```

The *result* 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 rmREST mode:

```
POST root/Calculator.ComplexCall
(...)
[[288722014,1231886296], ["one","two","three"], ["ABC","DEF","GHIJK"],
"ifzBgAAAAAAAAAAAAAAAAACNE01xEZXZcbGliXFNRTG10ZTNCZXh1XFRlc3RTUUwzLmV4ZQ==",
"ifzXow1EdkXbUkDYWJj"]
```

will be answered as such:

```
'{"result": [ ["ABC","DEF","GHIJK","one,two,three"],
"ifzX4w1EdgXbUkUMjg4NzIyMDE0LDEyMzE4ODYyOTY=",
"ifzXow1EdkXbUkDYWJjRDpcRGV2XGxpY1xTUUxpdGUzXGV4ZVxUZXR0U1FMMY5leGU="], "id":0}'
```

It matches the var / const / out parameters declaration of the method:

```
function ComplexCall(const Ints: TIntegerDynArray; Strs1: TRawUTF8DynArray;
var Str2: TWideStringDynArray; const Rec1: TVirtualTableModuleProperties;
var 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 Base-64 encoded.

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 would 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)
    ['{8D0F3839-056B-4488-A616-986CF8D4DEB7}']
    // purpose of this unique method is to substract two complex numbers
    // - using class instances as parameters
    procedure Substract(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:

```

POST root/ComplexCalculator.Substract
(...)
[{"Real":2,"Imaginary":3},{ "Real":20,"Imaginary":30}]

```

will be answered as such:

```

{"result":[{"Real":-18,"Imaginary":-27}], "id":0}

```

Those content have perfectly standard JSON declarations, so can be generated and consumed directly in any AJAX client.

In case of an error, the standard message object will be returned:

```

{
  "ErrorCode":400,
  "ErrorText":"Error description"
}

```

The following error descriptions may be returned by the service implementation from the server side:

ErrorText	Description
Method name required	rmJSON_RPC call without "method": field
Unknown method	rmJSON_RPC call with invalid method name (in rmRest mode, there is no specific message, since it may be a valid request)
Parameters required	The server expect at least a void JSON array (aka []) as parameters
Unauthorized method	This method is not allowed with the current authenticated user group - see <i>Security</i> above
... 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

#### 1.4.3.3.2.7.7. 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:

```
TServiceCustomAnswer = record
  Header: RawUTF8;
  Content: RawByteString;
end;
```

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 either (since there is no JSON result array any more).

In order to implement such method, you may define such an interface:

```
IComplexCalculator = interface(ICalculator)
  ['{8D0F3839-056B-4488-A616-986CF8D4DEB7}']
  function TestBlob(n: TComplexNumber): TServiceCustomAnswer;
end;
```

Which may be implemented for instance as such:

```
function TServiceComplexCalculator.TestBlob(n: TComplexNumber): TServiceCustomAnswer;
begin
  Result.Header := TEXT_CONTENT_TYPE_HEADER;
  Result.Content := FormatUTF8('%,%',[n.Real,n.Imaginary]);
end;
```

This will return not a JSON object, but a plain TEXT content.

Regression tests will make the following process:

```
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, in conjunction with rmREST mode and URI-encoded parameters.

#### 1.4.3.3.2.8. Hosting services

About hosting, 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 50) about this generic Client-Server architecture.

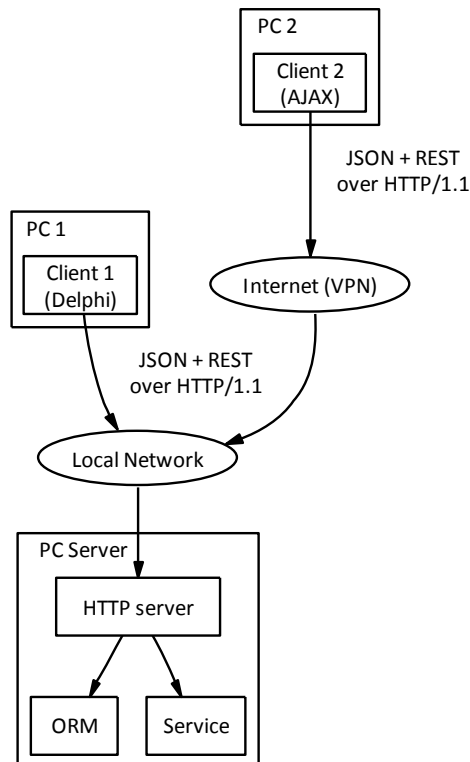
But you may find out some (good?) reasons which main induce another design:

- 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 would like to have a DMZ hosting only services, and a separate safe database and logic instance;
- Services are not the main part of your business, and you would like to enable or disable on need the exposed services, on demand;
- To implement an efficient solution for the most complex kind of application, as provided by *Domain-Driven design* (page 47).
- Whatever your IT or managers want *mORMot* to.

##### 1.4.3.3.2.8.1. 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.





*Service Hosting on mORMot - shared server*

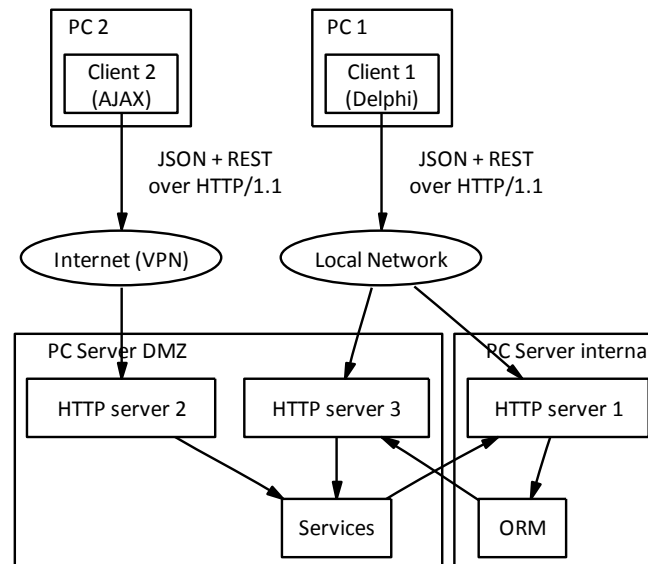
You can tune this solution, as such:

- Setting the group user rights properly - see below (page 195) - 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.

#### 1.4.3.3.2.8.2. 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*

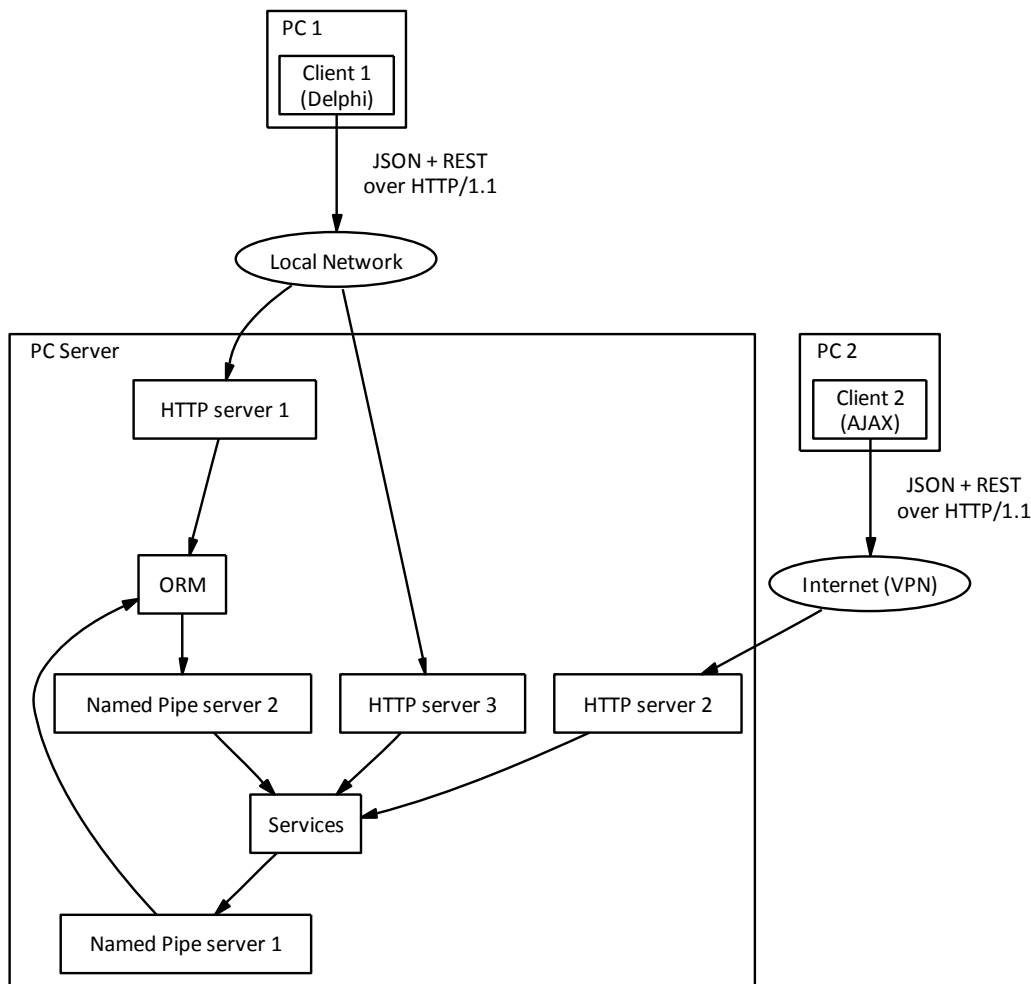
Of course, the database will be located on "PC Server internal", i.e. the one hosting the ORM, and the Services will be one regular client: so we may use *CRUD level cache* (page 155) on purpose to enhance performance. In order to access the remote ORM features, and provide a communication endpoint to the embedded services, a `TSQLRestServerRemoteDB` kind of server class can be used.

#### 1.4.3.3.2.8.3. 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 GDI 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.

1.4.3.3.2.9. Comparison with WCF

Here is a short reference table of WCF / mORMot SOA features and implementation patterns.

Feature	WCF	mORMot
Internal design	SOAP	RESTful
Hosting	exe/service/ISS/WAS	in-process/exe/service
Scalability/balancing	up to WAS	by dedicated hosting

MetaData	WSDL	none (only code)
Data contract	class	class/record
ORM integration	separated	integrated in the model
Service contract	interface + attributes	interface + shared Model
Versioning	XML name-space	interface signature
Message protocol	SOAP/custom	RESTful
Messaging	single/duplex	stateless (REST)
Sequence	attributes on methods	interface life time
Transactional	fully transactional	on implementation side
Instance life time	per call/per session/single	per call/per session/per user/ per group/single
Configuration	.config file or code	code
Operation	synchronous/asynchronous	synchronous (REST)
Session	available (optional)	available (optional)
Encryption	at Service level	at communication level
Compression	at Service level	at communication level
Serialization	XML/binary/JSON	JSON (customizable)
Communication protocol	HTTP/HTTPS/TCP/pipe/MSMQ	HTTP/HTTPS/TCP/pipe/GDI/in-process
HTTP/HTTPS server	http.sys	http.sys/native (winsock)
Security	Attribute driven	User group driven
Weight	Middle	Low
Speed	Good	High
Extensibility	verbose but complete	customizable
Standard	De facto	KISS design (e.g. JSON, HTTP)
Source code	Closed	Published
License	Proprietary	Open
Price	Depends	Free
Support	Official + community	Community
Runtime required	.Net framework (+ISS/WAS)	None (blank OS)

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* or *per group* instance life time, or GDI local communication) are even unique to *mORMot*.

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 your version of Delphi does not include SOAP process, you may take a look at [http://wiki.freepascal.org/Web\\_Service\\_Toolkit..](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.

The only missing feature of *mORMot* SOA is transactional process, which must be handled on server side, within the service implementation (e.g. with explicit commit or rollback).

An *Event Sourcing* design has been added to the *mORMot* road map, in order to handle transactions on the SOA side, relying on ORM for its data persistence, but not depending on database transactional abilities. In fact, transactions should better be implemented at SOA level, as we do want transactions to be database agnostic (*SQLite3* has a limited per-connection transactional scheme, and we do not want to rely on the DB layer for this feature). *Event Sourcing* sounds to be a nice pattern to implement a strong and efficient transactional process in our framework - see <http://blikl.abdullin.com/event-sourcing/why..>

## 1.4.4. Security and Testing

### 1.4.4.1. Security

The framework tries to implement security at several levels:

- Atomicity of the *SQLite3* database - see *ACID and speed* (page 102);
- *REST is Stateless* (page 129) architecture to avoid most synchronization issues;
- *Object-relational mapping* (page 46) associated to the Object pascal strong type syntax;
- *Per-table access right* functionalities built-in at lowest level of the framework;
- Build-in optional authentication mechanism, implementing both *per-user sessions* and individual *REST Query Authentication* - used e.g. for service tuned execution policy, using the authentication groups.

#### 1.4.4.1.1. Per-table access rights

A pointer to a *TSQLAccessRights* record, and its GET / POST / PUT / DELETE fields, is sent as parameter to the unique access point of the server class:

```
function TSQLRestServer.URI(const url, method, SentData: RawUTF8;  
    out Resp, Head: RawUTF8; RestAccessRights: PSQLAccessRights): Int64Rec;
```

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:

```
if MethodUp='POST' then begin  
    if Table=nil then begin  
        (...)  
    end else  
        // here, Table<>nil and TableIndex in [0..MAX_SQLFIELDS-1]  
        if not (TableIndex in RestAccessRights^.POST) then // check User  
            result.Lo := 401 else // HTTP Unauthorized  
            (...)
```

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.

#### 1.4.4.1.2. SQL statements safety

In our RESTful implementation, the POST command with no table associated in the URI allows to execute any SQL statement directly.

This special command should be carefully tested before execution, since SQL misuses could lead into major security issues. A *AllowRemoteExecute: boolean* field has therefore been made available in the *TSQLAccessRights* record to avoid such execution on any remote connection, if the SQL statement is not a SELECT, i.e. if it may affect the data content. By default, this field value is left to false in *SUPERVISOR\_ACCESS\_RIGHTS* constant for security reasons.

In the current implementation, the *SUPERVISOR\_ACCESS\_RIGHTS* constant is transmitted for all handled communication protocols (direct access, GDI messages, named pipe or HTTP).

Only direct access via *TSQLRestClientDB* will use *FULL\_ACCESS\_RIGHTS*, i.e. will have *AllowRemoteExecute* parameter set to true.

By default, when no explicit authentication mechanism is enabled in the framework, most commands will be executed, following the `SUPERVISOR_ACCESS_RIGHTS` constant. But no remote call will be allowed of SQL statements with no `SELECT` inside with a generic `POST` command.

#### 1.4.4.1.3. Authentication

##### 1.4.4.1.3.1. 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:

Criteria	HTTPS <i>basic auth</i>	Cookies+Session	Query Auth.
Browser integration	Native	Native	Via JavaScript
User Interaction	Rude	Custom	Custom
Web Service use (rough estimation)	95%	4%	1%
Session handling	Yes	Yes	No
Session managed by	Client	Server	N/A
Password on Server	Yes	Yes/No	N/A
Truly Stateless	Yes	No	Yes
Truly RESTful	No	No	Yes
HTTP-free	No	No	Yes

##### 1.4.4.1.3.2. 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 `TSQLite3HttpClientWinHTTP` and `TSQLite3HttpClientWinINet` clients classes are able to connect using HTTPS, and the `THttpApiServer` server class can send compatible content.

#### 1.4.4.1.3.3. 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.

#### 1.4.4.1.3.4. 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> 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.

#### 1.4.4.1.4. 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.

According to RESTful principle, handling per-session data is not to be implemented in such an 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 both `AuthGroup` and `AuthUser` are not available on the Server `TSQLModel` (i.e. if the `aHandleUserAuthentication` parameter was set to `false` for the `TSQLRestServer`. Create constructor), no authentication is performed. All tables will be accessible by any client, as stated in *SQL statements safety* (page 194). 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.

On the Server side, a dedicated service, accessible via the `ModelRoot/Auth` URI is to be called to register an User, and create a session.

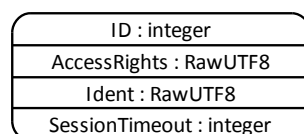
If authentication is enabled for the Client-Server process (i.e. if both `AuthGroup` and `AuthUser` are available in the Server `TSQLModel`, and the `aHandleUserAuthentication` parameter was set to `true` at the `TSQLRestServer` instance construction), the following security features will be added:

- Client *should* open a session to access to the Server, and provide a valid `UserName` / `Password` pair (see next paragraph);
- 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 `AllowRemoteExecute=true`;
- 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.

#### 1.4.4.1.4.1. Per-User authentication

On the Server side, two tables, defined by the `TSQLAuthGroup` and `TSQLAuthUser` classes will handle respectively per-group access rights, and user authentication.

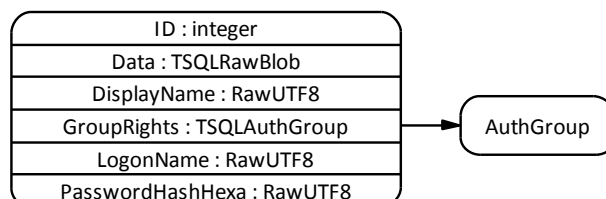
Here is the layout of the `AuthGroup` table, as defined by the `TSQLAuthGroup` class type:



*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:



*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 SHA-256 hash of its registered password. 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:

AuthGroup	POST SQL	Auth Read	Auth Write	Tables R	Tables W
Admin	Yes	Yes	Yes	Yes	Yes
Supervisor	No	Yes	No	Yes	Yes
User	No	No	No	Yes	Yes
Guest	No	No	No	Yes	No

Then the corresponding 'Admin', 'Supervisor' and 'User' AuthUser accounts are created, with the default 'synapse' password.

**You MUST override those default 'synapse' passwords for each AuthUser record to a custom genuine value.**

'Admin' will be the only group able to execute remote not SELECT SQL statements for POST commands (i.e. to have TSQLOAccessRights. AllowRemoteExecute = true) and modify the Auth\* tables (i.e. AuthUser and AuthGroup) content.

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 TSQLOAccessRights record layout.

This will implement both *Query Authentication* together with a group-defined *per-user right* management.

#### 1.4.4.1.4.2. Session handling

A dedicated RESTful service, available from the ModelRoot/Auth URI, is to be used for user authentication, handling so called sessions.

Here are the typical steps to be followed in order to create a new user session:

- Client sends a GET ModelRoot/auth?UserName=... request to the remote server;
- Server answers with an 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 failed;
- On success, Server will create a new in-memory session (sessions are not stored in the database, for lighter and safer process) and returns 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;
- When the Client is about to close (typically in TSQLORestClientURI. Destroy), the 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);

- 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.

Note that 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 implemented in the TSQLRest.AcquireWrite / ReleaseWrite protected methods, and used on the Server-Side by TSQLRestServer.URI;
- If the server do not handle Session/Authentication, transactions can be unsafe, in a multi-client concurrent architecture.

Therefore, for performance reasons in a multi-client environment, it's mandatory to release a transaction (via commit or rollback) as soon as possible.

#### 1.4.4.1.4.3. 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 *Client-Server* (page 125), then call the SetUser method as such:

```
Check(Client.SetUser('User','synapse')); // 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;
```

#### 1.4.4.1.4.4. URI signature

*Query Authentication* is handled at the Client side in TSQLRestClientURI. SessionSign method, by computing the session\_signature parameter for a given URL.

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 a 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, checked by the following Delphi expression:

```
(crc32(crc32(fPrivateSaltHash, PTimeStamp, 8), pointer(aURL), aURLlength)=aSignature);
```

A RESTful GET of the TSQLRecordPeople table with RowID=6 will have the following URI:

```
root/People/6?session_signature=0000004C000F6DD02E24541C
```

For better Server-side performance, the URI signature will use fast *crc32* hashing method, and not the more secure (but much slower) SHA-256. 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 SHA-256, 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 either from a AJAX / JavaScript client.

#### 1.4.4.1.4.5. Authentication using AJAX

Some 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.

See <http://synopse.info/forum/viewtopic.php?pid=2995#p2995..>

The current revision of the framework contains the code as expected by this JavaScript code - especially the results encoded as *JSON* (page 125) objects.

In the future, some "official" code will be available for such AJAX clients. It will probably rely on pure-pascal implementation using such an *Object-Pascal-to-JavaScript* compiler - it does definitely make sense to have Delphi-like code on the client side, not to break the ORM design. For instance, the Open Source DWS (*DelphiWebScript*) compiler matches our needs - see <http://delphitools.info/tag/javascript..>

#### 1.4.4.2. Testing

##### 1.4.4.2.1. Thread-safety

On the Server side, our Framework was designed to be thread-safe.

In fact, the TSQLRestServer.URI method is expected to be thread-safe, e.g. from the TSQLite3HttpServer. Request method. Thanks to the RESTful approach of our framework, this method is the only one which is expected to be thread-safe.

In order to achieve this thread-safety without sacrificing performance, the following rules were applied in TSQLRestServer.URI:

- Most of this methods's logic is to process the incoming parameters, so is thread-safe by design (e.g.

- Model and RecordProps access do not change during process);
- The *SQLite3* engine access is protected at SQL/JSON cache level, via `DB.LockJSON()` calls in `TSQLRestServerDB` methods;
  - `TSQLRestServerStatic` main methods (`EngineList`, `EngineRetrieve`, `EngineAdd`, `EngineUpdate`, `EngineDelete`, `EngineRetrieveBlob`, `EngineUpdateBlob`) are thread-safe: e.g. `TSQLRestServerStaticInMemory` uses a per-Table Critical Section;
  - `TSQLRestServerCallBack` methods (i.e. published methods of the inherited `TSQLRestServer` class) must be implemented to be thread-safe;
  - A protected `fSessionCriticalSection` is used to protect shared `fSession[]` access between clients;
  - Remote external tables - see *External database access* (page 112) - 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 `TSQLRestServerStaticInMemory`).

There is some kind of "giant lock" at the *SQLite3* engine level, so all requests process will be queued. This was not found to be a major issue, since the internal SQL/JSON cache implementation need such a global lock, and since most of the *SQLite3* resource use will consist in hard disk access, which gain to be queued.

From the Client-side, the REST core of the framework is expected to be Client-safe by design, therefore perfectly thread-safe: it's the benefit of the stateless architecture.

#### 1.4.4.2.2. 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.

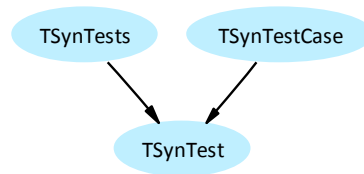
The framework has been implemented using this approach, and provide all the tools to write tests.

##### 1.4.4.2.2.1. Involved classes in Unitary testing

The `SynCommons.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 XE2, no external dependency).

The following diagram defines this class hierarchy:



*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.

#### 1.4.4.2.2. First steps in testing

Here are the functions we want to test:

```
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:

```
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):

```
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;  
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: 1000 assertions passed  
- Test double add: 1000 assertions passed  
Total failed: 0 / 2000 - Numbers adding PASSED  
  
1.2. Numbers multiplying:  
- Test integer multiply: 1000 assertions passed  
- Test double multiply: 1000 assertions passed  
Total failed: 0 / 2000 - Numbers multiplying PASSED
```

Generated with: Delphi 7 compiler

```
Time elapsed for all tests: 1.96ms
Tests performed at 23/07/2010 15:24:30

Total assertions failed for all test suits: 0 / 4000

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 classes defined.

This test has been uploaded in the `SQLite3\Sample\07 - SynTest` folder of the Source Code Repository.

#### 1.4.4.2.3. Implemented tests

The *SAD # DI-2.2.2* (page 833) 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 8,000,000 individual checks are performed for revision 1.17), 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 6;
- Delphi 7, with and without our Enhanced Run Time Library;
- Delphi 2007;
- Delphi 2010 (and in some cases, Delphi 2009 - but we assume that if it works with Delphi 2010, it will work with Delphi 2009);
- Delphi XE2.

Then all sample source code (including the *Main Demo* and *SynDBExplorer* sophisticated tools) are compiled, and user-level testing is performed.

#### 1.4.4.2.3. Logging

The framework makes an extensive use of the logging features introduced with the *SynCommons* unit - see below (page 219).

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 `s11Client` 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` and `s11Info` levels.

Those levels are available via the `TSQLLog` class, inheriting from `TSynLog`, as defined in `SQLite3Commons.pas`.

Three main `TSynLogClass` global variables are defined in order to use the same `TSynLog` class for all logging available in the framework units. Since all layers are not common, several variables have been defined, as such:

- `SynDBLog` for all *SynDB\** units, i.e. all generic database code;
- `SQLite3Log` for all *SQLite3\** units, i.e. all ORM related code;



- SynSQLite3Log for the SynSQLite3 unit, which implements the *SQLite3* engine itself.

For instance, if you execute the following statement at the beginning of `TestSQL3.dpr`, most regression tests will produce some logging, and will create about 270 MB of log file content, if executed:

```
with TSQLLog.Family do begin
  Level := LOG_VERBOSE;
  HighResolutionTimeStamp := true;
  TSynLogTestLog := TSQLLog;
  SynDBLog := TSQLLog;
  {$ifdef WITHLOG}
  SQLite3Log := TSQLLog;
  SynSQLite3Log := TSQLLog;
  {$endif}
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.

## 1.4.5. Source code

### 1.4.5.1. License

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).

This allows the use of the framework code in a wide variety of software projects, while still maintaining copy-left on code Synopse wrote.

In short:

- For GPL projects, use the GPL license - see <http://www.gnu.org/licenses/gpl-2.0.html>.
- For LGPL license, use the LGPL license - see <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> - which is the most permissive.

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 <http://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> for additional information;
- You have to publish any modified unit (e.g. `SynTaskDialog.pas`) in a public web site (e.g. <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 (C) 2012 Arnaud Bouchez provided by Synopse - <http://synopse.info> - under Mozilla Public License 1.1; modified source code is available at <http://SoftwareCompany.com/MPL>.*

Note that this documentation is under GPL license only, as stated in this document front page.

### 1.4.5.2. 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 <http://synopse.info/fossil>.

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.

#### 1.4.5.2.1. Obtaining the Source Code

Each official release of the framework is available in a dedicated SynopseSQLite3.zip archive from the official <http://synopse.info> web site, but you may want to use the latest version available.

You can 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 <http://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.

#### 1.4.5.2.2. Expected compilation platform

The framework source code tree will compile and is tested for the following platform:

- Delphi 6 up to Delphi XE2 compiler and IDE;
- For Windows 32 bit platform;
- GUI may be compiled optionally with third-party non Open-Source TMS Components, instead of default VCL components.

Some part of the library (e.g. SynCommons.pas or the *External database access* (page 112) units) are also compatible with Delphi 5.

Note that the framework is expected to create only 32 bit Windows applications yet (which will run without any issue on a 64 bit Windows operating system). But 64 bit compilation and even cross-platform is on its way: 64 bit support will need to adapt some low-level WinAPI changes, and cross-platform will probably use the Delphi XE2 FireMonkey library for User Interface generation, or other tools more neutral, using JavaScript and AJAX - or both. But the framework source code implementation and design tried to be as cross-platform as possible, since the beginning. See <http://blog.synopse.info/post/2011/08/08/Our-mORMot-won-t-hibernate-this-winter%2C-thanks-to-FireMonkey..>

#### 1.4.5.2.3. Note about sqlite3\*.obj files

In order to maintain the source code repository in a decent size, we excluded the sqlite3\*.obj storage in it, but provide the full source code of the *SQLite3* engine in the corresponding sqlite3.c file, ready to be compiled with all conditional defined as expected by SynSQLite3.pas.

Therefore, sqlite3.obj and sqlite3fts.obj files are available as a separated download, from <http://synopse.info/files/sqlite3obj.7z..> Please download the latest compiled version of these .obj

files from this link. You can also use the supplied `c.bat` file to compile from the original `sqlite3.c` file available in the repository, if you have the `bcc32` C command-line compiler installed.

The free version works and was used to create both `.obj` files, i.e. *C++Builder Compiler (bcc compiler) free download* - as available from *Embarcadero* web site. The upcoming 64 bit version will use the Microsoft Visual ++ compiler, since the *C++Builder* version is not existing yet.

#### 1.4.5.2.4. Folder layout

As retrieved from our source code repository, you'll find the following file layout.

Directory	Description
/	Root folder, containing common files
Htm1View/	A fork of the freeware THtm1View component, used as a demo of the SynPdf unit - not finished, and not truly Unicode ready
LVCL/	<i>Light VCL</i> replacement files for standard VCL (for Delphi 6-7 only)
RTL7/	Enhanced RTL .dcl for Delphi 7 (not mandatory at all), and <i>FastMM4</i> memory manager to be used before Delphi 2006
SQLite3/	Contains all ORM related files of the framework
SynProject/	Source code of the <i>SynProject</i> tool, used to create e.g. this documentation

In the *Root folder*, some common files are defined:

File	Description
CPort.*	A fork of the freeware <i>ComPort</i> Library ver. 2.63
PasZip.pas	ZIP/LZ77 Deflate/Inflate Compression in pure pascal
SynBigTable.pas	class used to store huge amount of data with fast retrieval
SynBz.pas bunzipasm.inc	fast BZ2 compression/decompression
SynBzPas.pas	pascal implementation of BZ2 decompression
SynCommons.pas	common functions used by most Synopse projects
SynCrtSock.pas	classes implementing HTTP/1.1 client and server protocol
SynCrypto.pas	fast cryptographic routines (hashing and cypher)
SynDprUses.inc	generic header included in the beginning of the uses clause of a .dpr source code
SynGdiPlus.pas	GDI+ library API access with anti-aliasing drawing
SynLZ.pas	SynLZ compression decompression unit - used by SynCommons.pas
SynLZO.pas	LZO compression decompression unit

SynMemoEx.pas	Synopse extended TMemo visual component (used e.g. in <i>SynProject</i> )
SynPdf.pas	PDF file generation unit
SynScaleMM.pas	multi-thread friendly memory manager unit - not finished yet
SynSelfTests.pas	automated tests for common units of the Synopse Framework
SynSQLite3.pas	SQLite3 embedded Database engine
SynTaskDialog.*	implement TaskDialog window (native on Vista/Seven, emulated on XP)
SynWinSock.pas	low level access to network Sockets for the Win32 platform
SynZip.pas deflate.obj trees.obj	low-level access to ZLib compression, 1.2.5
SynZipFiles.pas	high-level access to .zip archive file compression
Synopse.inc	generic header to be included in all units to set some global conditional definitions
vista.*	A resource file enabling theming under XP
vistaAdm.*	A resource file enabling theming under XP and Administrator rights under Vista

In the same *Root folder*, the external database-agnostic units are located:

File	Description
SynDB	abstract database direct access classes
SynOleDB	fast OleDB direct access classes
SynDBODBC	fast ODBC direct access classes
SynDBOracle	Oracle DB direct access classes (via OCI)
SynDBSQLite3	SQLite3 direct access classes

In the *SQLite3/* folder, the files defining the *Synopse ORM framework* (using mostly SynCommons, SynLZ, SynSQLite3, SynGdiPlus, SynCrtSock, SynPdf, SynTaskDialog and SynZip from the *Root folder*):

File	Description
Documentation/	Sub folder containing the source of the Synopse documentation
Samples/	Sub folders containing some sample code
SQLite3Commons.pas	Main unit of the ORM framework
SQLite3.pas	SQLite3 kernel bridge between SQLite3Commons.pas and SynSQLite3.pas

*.bmp *.rc	Resource files, compiled into *.res files
SQLite3FastCgiServer.pas	FastCGI server - not fully tested
SQLite3HttpClient.pas	HTTP/1.1 Client
SQLite3HttpServer.pas	HTTP/1.1 Server
SQLite3Pages.pas	Integrated Reporting engine
SQLite3Service.pas	Stand-alone Service
SQLite3ToolBar.pas	ORM ToolBar User Interface generation
SQLite3UI.*	Grid to display Database content
SQLite3UIEdit.*	Record edition dialog, used to edit record content on the screen
SQLite3UILogin.*	some common User Interface functions and dialogs
SQLite3UIOptions.*	General Options setting dialog, generated from code
SQLite3UIQuery.*	Form handling queries to a User Interface Grid, using our ORM RTTI to define search parameters and algorithms
SQLite3i18n.pas	internationalization (i18n) routines and classes
TestSQL3.dpr	Main unit testing program of the Synopse <i>mORMot</i> framework
TestSQL3Register.dpr	Run as administrator for <i>TestSQL3</i> to use <i>http.sys</i> on Vista/Seven
c.bat sqlite3.c	Source code of the <i>SQLite3</i> embedded Database engine

### 1.4.5.3. Installation

Just unzip the .zip archive, including all sub-folders, into a local directory of your computer (for instance, D:).

Then add the following paths to your Delphi IDE (in *Tools/Environment/Library* menu):

- *Library path*: (...existing path...);D:\Dev\Lib;D:\Dev\Lib\SQLite3
- *Search path*: (...existing path...);D:\Dev\Lib;D:\Dev\Lib\SQLite3

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.

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.

### 1.4.6. SynCommons unit

In the following next paragraphs, we'll comment some main features of the lowest-level of the framework, mainly located in `SynCommons.pas`:

- Unicode and UTF-8;
- currency type;
- *dynamic array* wrappers (`TDynArray` and `TDynArrayHashed`);
- Enhanced logging.

#### 1.4.6.1. Unicode and UTF-8

Our *mORMot* Framework has 100% UNICODE compatibility, that is compilation under Delphi 2009/2010/XE/XE2. 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 version, i.e. from Delphi 6 up to Delphi 2007.

Since our framework is natively UTF-8 (this is the better character encoding for fast text - JSON - streaming/parsing and it is natively supported by the *SQLite3* engine), we had to establish a secure way our framework used 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 Free Pascal Compiler.

Some string types have been defined, and used in the code for best cross-compiler efficiency (avoiding most conversion between formats):

- `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 `SQLite3i18n`), i.e. text ready to be used within the VCL, as either `AnsiString` (for Delphi 2 to 2007) or `UnicodeString` (for Delphi 2009/2010/XE/XE2);
- `RawUnicode` in some technical places (e.g. direct `Win32 *W()` API call in Delphi 7) - note: this type is NOT compatible with Delphi 2009/2010/XE/XE2 `UnicodeString`;
- `RawByteString` for byte storage (e.g. for `FileFromStream()` 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 2009/2010/XE/XE2 `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's 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 `SQLite3i18n.pas` unit) or the following function from `SynCommons.pas`:

```
/// convert any UTF-8 encoded String into a generic VCL Text
/// - it's preferred to use TLanguageFile.UTF8ToString() in SQLite3i18n,
/// which will handle full i18n of your application
/// - it will work as is with Delphi 2009/2010/XE/XE2 (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 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.

#### 1.4.6.2. 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:

```
function CompCurrency(var 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 will be faster than the default implementation, which uses the FPU (or SSE2 under x64 architecture) instructions.

You some direct currency handling 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): PtrInt;`
- 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 *External database access* (page 112).

#### 1.4.6.3. Dynamic array wrapper

The `SynCommons` unit has been enhanced, since version 1.13:

- `BinToBase64` and `Base64ToBin` conversion functions;
- 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 (access is optimized, and all array content will be allocated at once, therefore reducing the memory fragmentation and CPU cache slow-down).

They 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).

#### 1.4.6.3.1. TList-like properties

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

```
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 records...

Records need only to be packed and have only not reference counted fields (byte, integer, double...) or string reference-counted fields (no Variant nor Interface within). But TDynArray is able to handle records within records, and even *dynamic arrays* within records.

Yes, you read well: it will handle a *dynamic array* of records, in which you can put some strings 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.

#### 1.4.6.3.2. Enhanced features

Some methods were defined in the `TDynArray` record/object, 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:

```
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 then // 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 XE2, without the need of the slow enhanced RTTI...

#### 1.4.6.3.3. 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.

```
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));
```

#### 1.4.6.3.4. JSON serialization

##### 1.4.6.3.4.1. TDynArray JSON features

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

Most common kind of *dynamic arrays* (array of byte, word, integer, cardinal, `Int64`, double, currency, `RawUTF8`, `SynUnicode`, `WinAnsiString`, string) 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).

Applications can supply a custom JSON serialization for any other dynamic array, via the `TTextWriter.RegisterCustomJSONSerializer()` class method. Two callbacks are to be supplied for a dynamic array type information, in order to handle proper serialization and un-serialization of the JSON array.

Other not-known *dynamic arrays* (like any array of packed record) will be serialized as binary, then *Base64* encoded. This method will always work, but won't be easy to deal with from an AJAX client.

If you have any ideas of standard *dynamic arrays* which should be handled, feel free to post your proposal in the forum!

These methods were the purpose of adding in `SynCommons` both `BinToBase64` and `Base64ToBin` functions, very optimized for speed. In fact, we will use the *Base64* encoding to load or save any *dynamic array* of records from a `TSQLRecord` published property... using *SQLite3* BLOB fields for storage, but *Base64* for JSON transmission (much more efficient than hexadecimal, and still JSON compatible).

This JSON serialization will indeed be used in our main ORM to support *dynamic arrays* as enhanced properties (stored as BLOB), and in the interface-based SOA architecture of the framework, for content transmission.

#### 1.4.6.3.4.2. Custom JSON serialization of dynamic arrays

As just stated, any *dynamic array* can be serialized using a custom JSON format, via the `TTextWriter.RegisterCustomJSONSerializer()` class method. It will use the same method used for record custom serialization - see *Custom JSON serialization of records* (page 171). In fact, if you register a *dynamic array* custom serializer, it will also be used for the associated internal record.

For instance, we would like to serialize a dynamic array of the following record:

```
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 as a *Base64* encoded binary buffer. This won't be easy to understand from an AJAX client, for instance.

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.:

```
[1,2001,3001,4001,"1","1001"]
```

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:

```
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:

```
class function TCollTstDynArray.FVReader(P: PUTF8Char; var aValue;
  out aValid: Boolean): PUTF8Char;
var V: TFV absolute aValue;
begin // '[1,2001,3001,4001,"1","1001"],[2,2002,3002,4002,"2","1002"],...'
  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:

```
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:

```
{"Major":1,"Minor":2001,"Release":3001,"Build":4001,"Main":"1","Detailed":"1001"}
```

Then the corresponding *Reader* callback could be written as:

```
class function TCollTstDynArray.FVReader2(P: PUTF8Char; var aValue;  
  out aValid: Boolean): PUTF8Char;  
var V: TFV absolute aValue;  
  Values: TPUtf8CharDynArray;  
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 := GetInteger(Values[0]);  
  V.Minor := GetInteger(Values[1]);  
  V.Release := GetInteger(Values[2]);  
  V.Build := GetInteger(Values[3]);  
  V.Main := UTF8DecodeToString(Values[4],StrLen(Values[4]));  
  V.Detailed := UTF8DecodeToString(Values[5],StrLen(Values[5]));  
  aValid := true;  
end;
```

Most of the JSON decoding process is performed within the JSONDecode() function, which will let Values[] point to null-terminated un-escaped content within the P^ buffer. In fact, such process will do only one memory allocation (for Values[]), 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:

```
TTextWriter.RegisterCustomJSONSerializer(TypeInfo(TFVs),nil,nil);
```

You can define now your custom JSON serializers, starting for the above code as reference.

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 *Custom JSON serialization of records* (page 171).

#### 1.4.6.3.5. 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 `TIntegerDynArray` 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`:

```
TSQLStatementCache = record
  StatementSQL: RawUTF8;
  Statement: TSQLRequest;
end;
TSQLStatementCacheDynArray = array of TSQLStatementCache;

TSQLStatementCached = object
  Cache: TSQLStatementCacheDynArray;
  Count: integer;
  Caches: TDynArrayHashed;
  DB: TSQlite3DB;
  procedure Init(aDB: TSQlite3DB);
  function Prepare(const GenericSQL: RawUTF8): PSQLRequest;
  procedure ReleaseAllDBStatements;
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: TSQlite3DB);
begin
  fStatementCache.Init(aDB);
  (...)
```

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.

#### 1.4.6.4. Enhanced logging

A new logging mechanism has been introduced with revision 1.13 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.

##### 1.4.6.4.1. Using logging

It's now used by 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*0-15-1027 OS=2.3=5.1.2600 Wow64=0 Freq=3579545
TSynLogTest 1.13 2011-04-13 05:40:25

```

```

20110413 05402559 fail TTestLowLevelCommon(00B31D70) Low level common: TDynArray "" stack trace
0002FE0B SynCommons.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:

```

procedure TSynTestsLogged.Failed(const msg: string; aTest: TSynTestCase);
begin
  inherited;
  with TestCase[fCurrentMethod] do
    fLogFile.Log(sllFail, '%: % "%"',
      [Ident, TestName[fCurrentMethodIndex], msg], aTest);
end;

```

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:

```

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;
20110325 19325801 - MyDBUnit.TMyDB(004E11F4).SQLExecute

```



Note that by default you have human-readable *time and date* written to the log, but it's 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.

#### 1.4.6.4.2. 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.

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):

```
0000000000000B56 + TTestCompression(00AB3570).000E6C79 SynSelfTests.TTestCompression.TestLog
(376)
0000000000001785 - TTestCompression(00AB3570).000E6D09 SynSelfTests.TTestCompression.TestLog
(385)
```

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 `.mab` 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.

You have several debugging levels available. And even 4 custom types. It's worth saying that the logging level is a SET, and not an enumerated: that is, you can select several kind of logging information to be logged at once, on request:

```
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);

/// 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;
```

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'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 any internal caching mechanism (it's 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);
- sllCustom\* items can be used for any purpose by your programs.

#### 1.4.6.4.3. 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:

```
000000000000090B EXCOS EAccessViolation (C0000005) at 000E9C7A SynSelfTests.Proc1 (785) stack
trace 000E9D51 SynSelfTests.Proc2 (801) 000E9CC1 SynSelfTests.Proc1 (790) 000E9D51
SynSelfTests.Proc2 (801) 000E9CC1 SynSelfTests.Proc1 (790) 000E9D51 SynSelfTests.Proc2 (801)
000E9CC1 SynSelfTests.Proc1 (790) 000E9D51 SynSelfTests.Proc2 (801) 000E9CC1 SynSelfTests.Proc1
(790) 000E9D51 SynSelfTests.Proc2 (801) 000E9CC1 SynSelfTests.Proc1 (790) 000E9E2E
SynSelfTests.TestsLog (818) 000EA0FB SynSelfTests (853) 00003BF4 System.InitUnits 00003C5B
System.@StartExe 000064AB SysInit.@InitExe 000EA3EC TestSQL3 (153)
```

The TSynLogInfo logging level makes a difference between high-level Delphi exceptions (sllException) and lowest-level OS exceptions (sllExceptionOS) like EAccessViolation.

##### 1.4.6.4.3.1. Intercepting exceptions

In order to let our TSynLog logging class intercept all exceptions, we use the low-level global RtlUnwindProc pointer, defined in System.pas.

Alas, under Delphi 5, this global RtlUnwindProc variable is not existing. The code calls directly the RtlUnWind Windows API function, with no hope of custom interception.

Two solutions could be envisaged:

- Modify the Sytem.pas source code, adding the new RtlUnwindProc variable, just like Delphi 7;
- Patch the assembler code, directly in the process memory.

The first solution is simple. Even if compiling System.pas is a bit more difficult than compiling other units, we already made that for our *Enhanced RTL units*. But you'll have to change the whole build chain in order to use your custom System.dcu instead of the default one. And some third-party units (only available in .dcu form) may not like the fact that the System.pas interface changed...

So we used the second solution: change the assembler code in the running process memory, to let call our RtlUnwindProc variable instead of the Windows API.

#### 1.4.6.4.3.2. One patch to rule them all

The first feature we have to do is to allow on-the-fly change of the assembler code of a process.

In fact, we already use this in order to provide class-level variables, as stated by *SDD # DI-2.1.3*.

We have got the PatchCodePtrUInt function at hand to change the address of each a RtlUnwind call.

We'll first define the missing global variable, available since Delphi 6, for the Delphi 5 compiler:

```
{$ifdef DELPHI5OROLDER}
// Delphi 5 doesn't define the needed RTLUnwindProc variable :(
// so we will patch the System.pas RTL in-place
var
  RTLUnwindProc: Pointer;
```

The RtlUnwind API call we have to hook is defined as such in System.pas:

```
procedure RtlUnwind; external kernel name 'RtlUnwind';
0040115C FF25CC14100 jmp dword ptr [$0041c15c]
```

The \$0041c15c is a pointer to the address of RtlUnwind in kernel32.dll, as retrieved during linking of this library to the main executable process.

The patch will consist in changing this asm call into this one:

```
0040115C FF25??????? jmp dword ptr [RTLUnwindProc]
```

Where ???????? is a pointer to the global RTLUnwindProc variable.

The problem is that we do not have any access to this procedure RtlUnwind declaration, since it was declared only in the implementation part of the System.pas unit. So its address has been lost during the linking process.

So we will have to retrieve it from the code which in fact calls this external API, i.e. from this assembler content:

```
procedure _HandleAnyException;
asm
  (...)
  004038B6 52          push edx // Save exception object
  004038B7 51          push ecx // Save exception address
  004038B8 8B542428     mov edx,[esp+$28]
  004038BC 83480402     or dword ptr [eax+$04],$02
  004038C0 56          push esi // Save handler entry
  004038C1 6A00        push $00
  004038C3 50          push eax
  004038C4 68CF384000   push $004038cf // @@returnAddress
  004038C9 52          push edx
  004038CA E88DD8FFFF   call RtlUnwind
```

So we will retrieve the RtlUnwind address from this very last line.

The E8 byte is in fact the *opcode* for the asm call instruction. Then the called function is stored as an *integer* offset, starting from the current pointing value.

The E8 8D D8 FF FF byte sequence is executed as "*call the function available at the current execution address, plus integer(\$ffffd88d)*". As you may have guessed, \$004038CA+\$ffffd88d+5 points to the RtlUnwind definition.

So here is the main function of this patching:

```
procedure Patch(P: PAnsiChar);
var i: Integer;
```

```

    addr: PAnsiChar;
begin
  for i := 0 to 31 do
    if (PCardinal(P) ^= $6850006a) and // push 0; push eax; push @@returnAddress
      (PWord(P+8) ^= $E852) then begin // push edx; call RtlUnwind
      inc(P,10); // go to call RtlUnwind address
      if PInteger(P) ^ < 0 then begin
        addr := P+4+PInteger(P)^;
        if PWord(addr) ^= $25FF then begin // jmp dword ptr []
          PatchCodePtrUInt(Pointer(addr+2), cardinal(@RTLUnwindProc));
          exit;
        end;
      end;
    end else
      inc(P);
    end;
  end;
end;

```

We will call this Patch subroutine from the following code:

```

procedure PatchCallRtlUnwind;
asm
  mov eax, offset System.@HandleAnyException+200
  call Patch
end;

```

You can note that we need to retrieve the `_HandleAnyException` address from asm code. In fact, the compiler does not let access from plain pascal code to the functions of `System.pas` having a name beginning with an underscore.

Then the following lines:

```

  for i := 0 to 31 do
    if (PCardinal(P) ^= $6850006a) and // push 0; push eax; push @@returnAddress
      (PWord(P+8) ^= $E852) then begin // push edx; call RtlUnwind

```

will look for the expected opcode asm pattern in `_HandleAnyException` routine.

Then we will compute the position of the `jmp dword ptr []` call, via this line:

```

    addr := P+4+PInteger(P)^;

```

After checking that this is indeed a `jmp dword ptr []` instruction (expected opcodes are `FF 25`), we will simply patch the absolute address with our `RTLUnwindProc` procedure variable.

With this code, each call to `RtlUnwind` in `System.pas` will indeed call the function set by `RTLUnwindProc`.

In our case, it will launch the following procedure:

```

procedure SynRtlUnwind(TargetFrame, TargetIp: pointer;
  ExceptionRecord: PExceptionRecord; ReturnValue: Pointer); stdcall;
asm
  pushad
  cmp byte ptr SynLogExceptionEnabled, 0
  jz @oldproc
  mov eax, TargetFrame
  mov edx, ExceptionRecord
  call LogExcept
@oldproc:
  popad
  pop ebp // hidden push ebp at asm level
{$ifdef DELPHI50ROLDER}
  jmp RtlUnwind
{$else}
  jmp oldUnWindProc
{$endif}

```

```
end;
```

This code will therefore:

- Save the current register context via pushad / popad opcodes pair;
- Check if TSynLog should intercept exceptions (i.e. if the global SynLogExceptionEnabled boolean is true);
- Call our logging function LogExcept;
- Call the default Windows RtlUnwind API, as expected by the Operating System.

#### 1.4.6.4.4. Serialization

*dynamic arrays* can also be serialized as JSON in the log on request, via the default TSynLog class, as defined in SynCommons unit - see *Dynamic array wrapper* (page 212).

The TSQLLog class (using the enhanced RTTI methods defined in SQLite3Commons 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:

```
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)  
000000000000171B info  
{ "TSQLRecordPeople(00AB92E0)": { "ID": 16, "FirstName": "Louis", "LastName": "Croivébaton", "Data": "", "YearOfBirth": 1754, "YearOfDeath": 1793 } }  
0000000000001731 - 000EA005 SynSelfTests.TestPeopleProc (794)
```

For instance, if you add to your program:

```
uses  
  SynCommons;  
(...)  
  TSynLog.Family.Level := [sllExceptionOS];
```

all OS exceptions (excluding pure Delphi exception like EConvertError and such) will be logged to a separate log file.

```
TSynLog.Family.Level := [sllException, sllExceptionOS];
```

will trace also Delphi exceptions, for instance.

#### 1.4.6.4.5. Family matters

You can have 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:

```
with TSynLogDB.Family do
begin
  Level := LOG_VERBOSE;
  PerThreadLog := true;
  DestinationPath := 'C:\Logs';
end;
```

#### 1.4.6.4.6. Automated log archival

Log archives can be created with the following settings:

```
with TSynLogDB.Family do
begin
  (...)
  OnArchive := EventArchiveZip;
  ArchivePath := '\\Remote\WKS2302\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` 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.

#### 1.4.6.4.7. Log Viewer

Since the log files tend to be huge (for instance, if you set the logging for our unitary tests, the 7,000,000 unitary tests create a 320 MB log file), 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.

##### 1.4.6.4.7.1. Open log files

You can run it with a specified log file on the command line, or use the "*Open*" 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's worth saying that the viewer was designed to be *fast*.

In fact, it takes no time to open any log file. For instance, a 320 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 320 MB content.

It uses internally memory mapped files and optimized data structures to access to the data as fast as possible - see TSynLogFile class.

#### **1.4.6.4.7.2. 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 (not shown by default).

The command panel allows to *Open* a .log file, see 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 an 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, IMHO).

Just below the "Open" 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 click on the *Filter* button (or 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 and Client-Server events traced in the log is definitively a huge benefit for customer support and bug tracking.

#### **1.4.6.4.7.3. 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 would have never thought about before.

## 1.4.7. Source code implementation

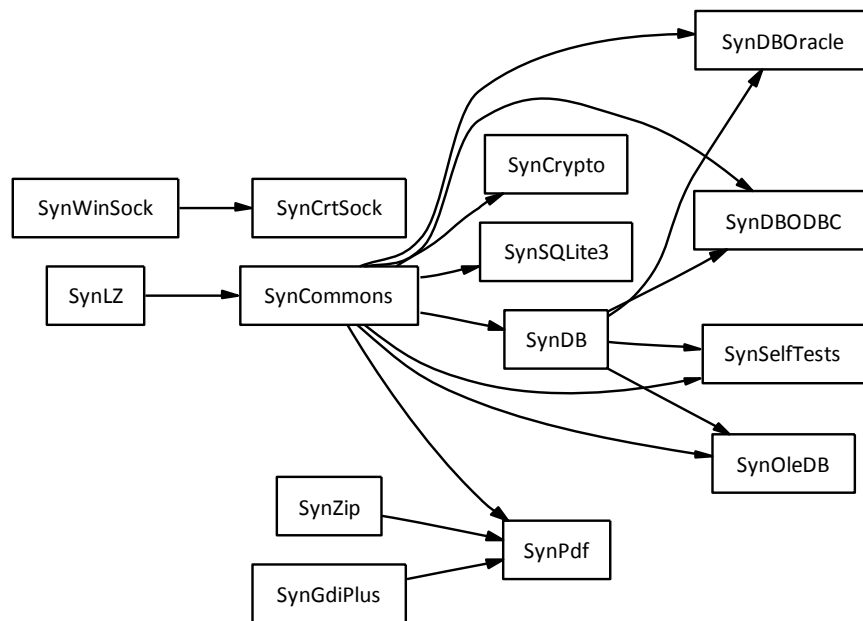
### 1.4.7.1. mORMot Framework used Units

The mORMot Framework makes use of the following units.

**Units located in the "Lib\" directory:**

Source File Name	Description	Page
<i>SynCommons</i>	Common functions used by most Synapse projects	229
<i>SynCrtSock</i>	Classes implementing HTTP/1.1 client and server protocol	369
<i>SynCrypto</i>	Fast cryptographic routines (hashing and cypher)	383
<i>SynDB</i>	Abstract database direct access classes	395
<i>SynDBODBC</i>	ODBC 3.x library direct access classes to be used with our SynDB architecture	428
<i>SynDBOracle</i>	Oracle DB direct access classes (via OCI)	432
<i>SynGdiPlus</i>	GDI+ library API access	438
<i>SynLZ</i>	SynLZ Compression routines	445
<i>SynLZO</i>	Fast LZO Compression routines	447
<i>SynOleDB</i>	Fast OleDB direct access classes	447
<i>SynPdf</i>	PDF file generation	459
<i>SynSelfTests</i>	Automated tests for common units of the Synapse Framework	500
<i>SynSQLite3</i>	SQLite3 embedded Database engine direct access	502
<i>SynTaskDialog</i>	Implement TaskDialog window (native on Vista/Seven, emulated on XP)	553

Source File Name	Description	Page
<i>SynWinSock</i>	Low level access to network Sockets for the Win32 platform	559
<i>SynZip</i>	Low-level access to ZLib compression (1.2.5 engine version)	560



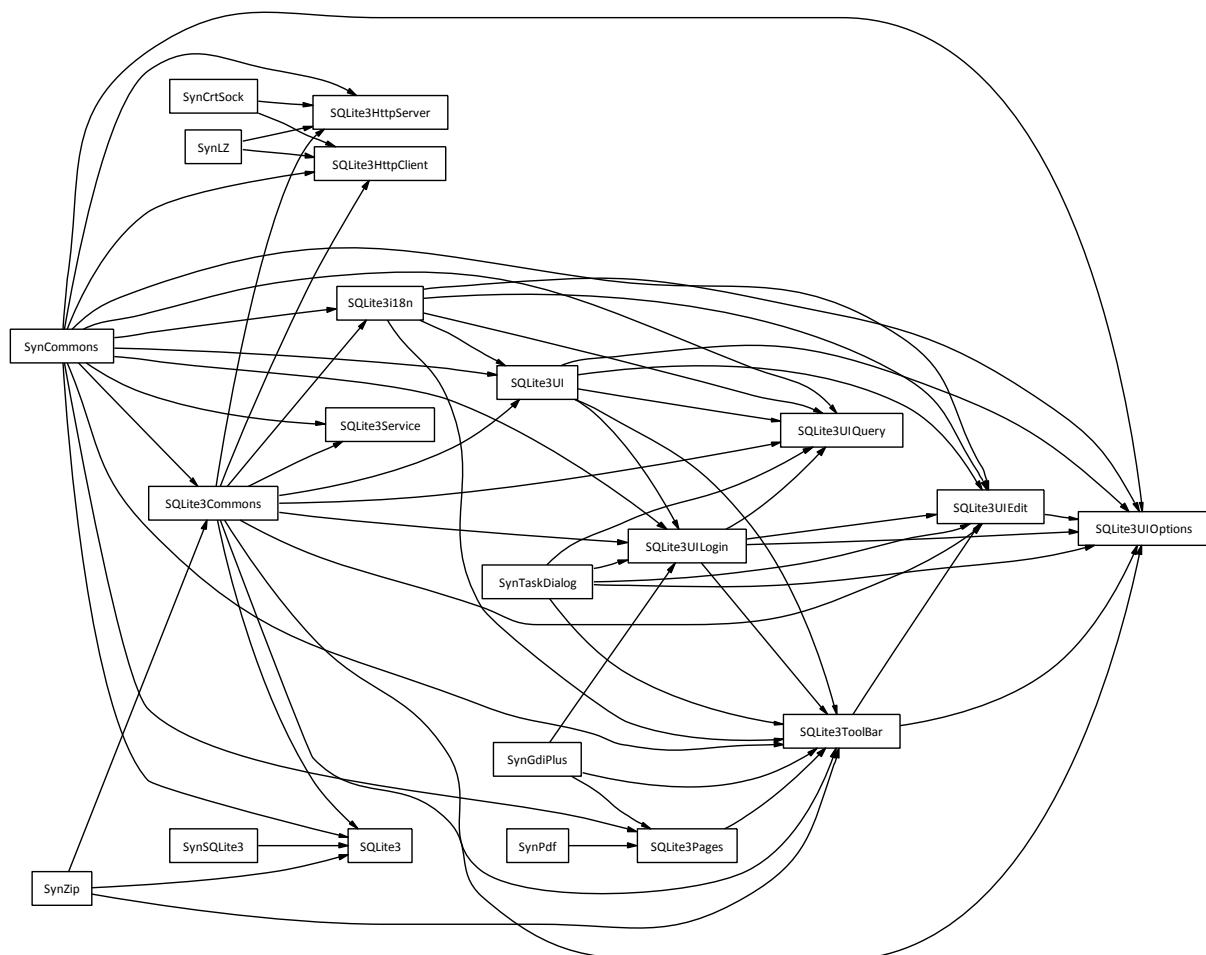
Unit dependencies in the "Lib" directory

#### Units located in the "Lib\SQLite3\" directory:

Source File Name	Description	Page
<i>SQLite3</i>	SQLite3 embedded Database engine used as the kernel of mORMot	567
<i>SQLite3Commons</i>	Common ORM and SOA classes	575
<i>SQLite3HttpClient</i>	HTTP/1.1 RESTFUL JSON mORMot Client classes	731
<i>SQLite3HttpServer</i>	HTTP/1.1 RESTFUL JSON mORMot Server classes	733
<i>SQLite3i18n</i>	Internationalization (i18n) routines and classes	736
<i>SQLite3Pages</i>	Reporting unit	745
<i>SQLite3SelfTests</i>	Automated tests for common units of the Synopse mORMot Framework	760
<i>SQLite3Service</i>	Win NT Service management classes	761
<i>SQLite3ToolBar</i>	Database-driven Office 2007 Toolbar	768



Source File Name	Description	Page
<i>SQLite3UI</i>	Grid to display Database content	781
<i>SQLite3UIEdit</i>	Record edition dialog, used to edit record content on the screen	790
<i>SQLite3UILogin</i>	Some common User Interface functions and dialogs	793
<i>SQLite3UIOptions</i>	General Options setting dialog	796
<i>SQLite3UIQuery</i>	Form handling queries to a User Interface Grid	798



*Unit dependencies in the "Lib\SQLite3" directory*

#### 1.4.7.2. SynCommons unit

**Purpose:** Common functions used by most Synapse projects

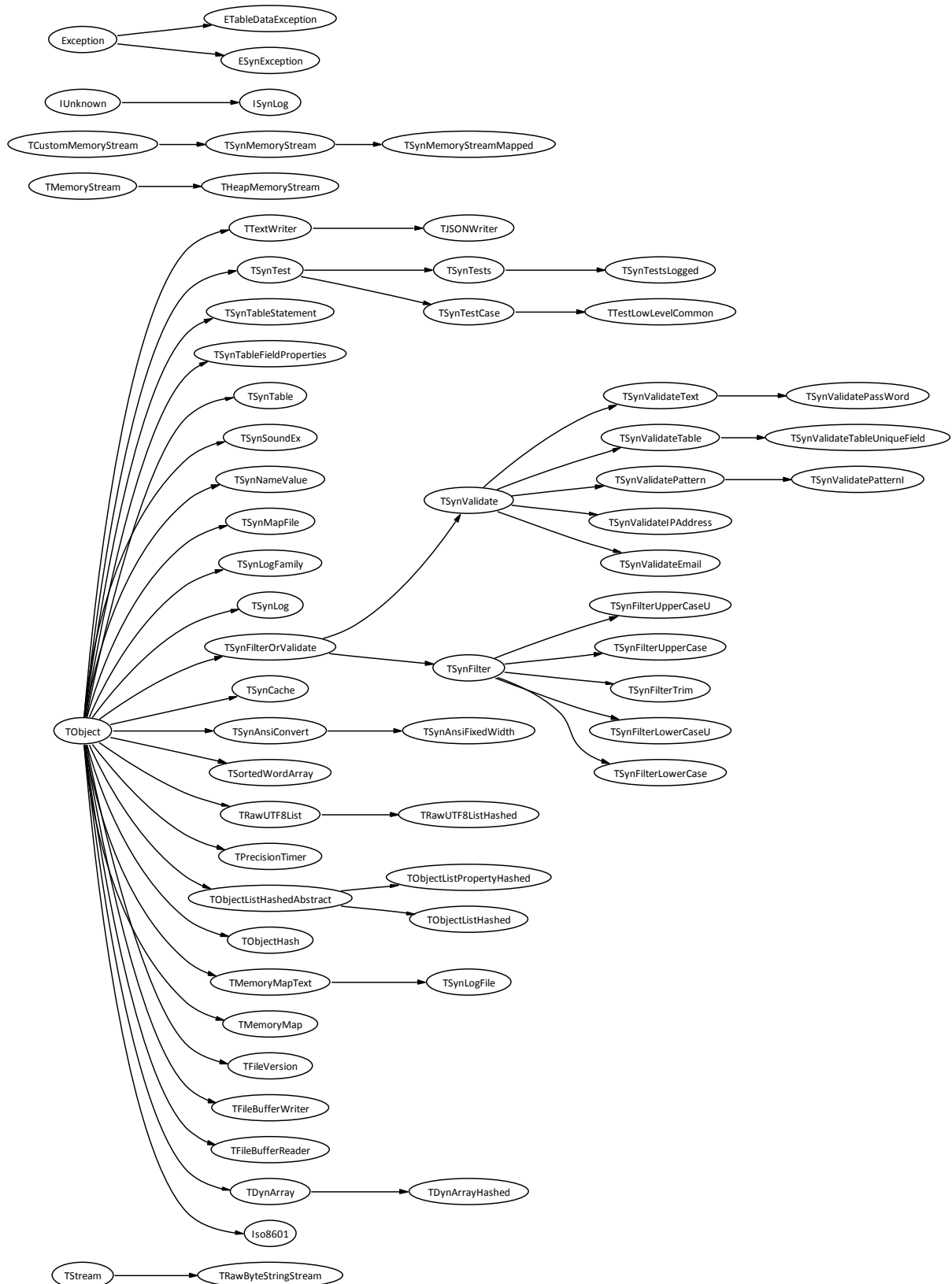
- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**The *SynCommons* unit is quoted in the following items:**

SWRS #	Description	Page
DI-2.1.2	UTF-8 JSON format shall be used to communicate	830
DI-2.2.2	The framework libraries, including all its <i>SQLite3</i> related features, shall be tested using Unitary testing	833

**Units used in the *SynCommons* unit:**

Unit Name	Description	Page
<i>SynLZ</i>	SynLZ Compression routines - licensed under a MPL/GPL/LGPL tri-license; version 1.17	445



*SynCommons class hierarchy*

**Objects implemented in the *SynCommons* unit:**

Objects	Description	Page
ESynException	Generic parent class of all custom Exception types of this unit	234
ETableDataException	Exception raised by all TSynTable related code	234
Iso8601	Usefull object to type cast TTimeLog type into Iso-8601 or TDateTime	272
ISynLog	A generic interface used for logging a method	290
TDynArray	A wrapper around a dynamic array with one dimension	239
TDynArrayHashed	Used to access any dynamic array elements using fast hash	247
TFileBufferReader	This structure can be used to speed up reading from a file	265
TFileBufferWriter	This class can be used to speed up writing to a file	264
TFileVersion	To have existing RTTI for published properties used to retrieve version information from any EXE	273
THeapMemoryStream	To be used instead of TMemoryStream, for speed	274
TJSONWriter	Simple writer to a Stream, specialized for the JSON format and SQL export	256
TMemoryMap	Handle memory mapping of a file content used to store and retrieve Words in a sorted array	261
TMemoryMapText	Able to read a UTF-8 text file using memory map	262
TObjectHash	Abstract class able to use hashing to find an object in O(1) speed	249
TObjectListHashed	This class behaves like TList/TObjectList, but will use hashing for (much) faster IndexOf() method	258
TObjectListPropertyHashed	This class will hash and search for a sub property of the stored objects	258
TOSVersionInfoEx	Not defined in older Delphi versions	274
TPrecisionTimer	High resolution timer (for accurate speed statistics)	283
TRawByteStringStream	A TStream using a RawByteString as internal storage	263
TRawUTF8List	This class is able to emulate a TStringList with our native UTF-8 string type	259
TRawUTF8ListHashed	A TRawUTF8List which will use an internal hash table for faster IndexOf()	261
TSortCompareTmp	Internal value used by TSynTableFieldProperties.SortCompare() method to avoid stack allocation	275
TSortedWordArray	Used to store and retrieve Words in a sorted array	239
TSynAnsiConvert	An abstract class to handle Ansi to/from Unicode translation	235

Objects	Description	Page
TSynAnsiFixedWidth	A class to handle Ansi to/from Unicode translation of fixed width encoding (i.e. non MBCS)	237
TSynCache	Implement a cache of some key/value pairs, e.g. to improve reading speed	257
TSynFilter	Will define a filter to be applied to a Record field content (typically a TSQLRecord)	271
TSynFilterLowerCase	A custom filter which will convert the value into Lower Case characters	271
TSynFilterLowerCaseU	A custom filter which will convert the value into Lower Case characters	272
TSynFilterOrValidate	Will define a filter or a validation process to be applied to a database Record content (typically a TSQLRecord)	267
TSynFilterTrim	A custom filter which will trim any space character left or right to the value	272
TSynFilterUpperCase	A custom filter which will convert the value into Upper Case characters	271
TSynFilterUpperCaseU	A custom filter which will convert the value into Upper Case characters	271
TSynHash	Internal structure used to store one item hash	246
TSynLog	A per-family and/or per-thread log file content	294
TSynLogCurrentIdent	Used by ISynLog/TSynLog.Enter methods to handle recursivity calls tracing	293
TSynLogFamily	Regroup several logs under an unique family name	291
TSynLogFile	Used to parse a .log file, as created by TSynLog, into high-level data	298
TSynLogFileProc	Used by TSynLogFile to refer to a method profiling in a .log file	298
TSynMapFile	Retrieve a .map file content, to be used e.g. with TSynLog to provide additional debugging information	289
TSynMapSymbol	A debugger symbol, as decoded by TSynMapFile from a .map file	288
TSynMapUnit	A debugger unit, as decoded by TSynMapFile from a .map file	289
TSynMemoryStream	A TStream pointing to some in-memory data, for instance UTF-8 text	263
TSynMemoryStreamMapped	A TStream created from a file content, using fast memory mapping	263
TSynNameValue	Pseudo-class used to store Name/Value RawUTF8 pairs	250
TSynNameValueItem	Store one Name/Value pair, as used by TSynNameValue class	249
TSynSoundEx	Fast search of a text value, using the Soundex searching mechanism	238
TSynTable	Store the description of a table with records, to implement a Database	280

Objects	Description	Page
TSynTableFieldProperties	Store the type properties of a given field / database column	275
TSynTableStatement	Used to parse a SELECT SQL statement	274
TSynTest	A generic class for both tests suit and cases	283
TSynTestCase	A class implementing a test case	284
TSynTests	A class used to run a suit of test cases	285
TSynTestsLogged	This overridden class will create a .log file in case of a test case failure	297
TSynValidate	Will define a validation to be applied to a Record (typically a TSQLRecord) field content	267
TSynValidateEmail	IP address validation to be applied to a Record field content (typically a TSQLRecord)	268
TSynValidateIPAddress	IP v4 address validation to be applied to a Record field content (typically a TSQLRecord)	268
TSynValidatePassWord	Strong password validation for a Record field content (typically a TSQLRecord)	271
TSynValidatePattern	Grep-like case-sensitive pattern validation of a Record field content (typically a TSQLRecord)	269
TSynValidatePatternI	Grep-like case-insensitive pattern validation of a Record field content (typically a TSQLRecord)	269
TSynValidateTable	Will define a validation to be applied to a TSynTableFieldProperties field	279
TSynValidateTableUniqueField	Will define a validation for a TSynTableFieldProperties Unique field	280
TSynValidateText	Text validation to be applied to a Record field content (typically a TSQLRecord)	269
TTestLowLevelCommon	This test case will test most functions, classes and types defined and implemented in the SynCommons unit	287
TTextWriter	Simple writer to a Stream, specialized for the TEXT format	250
TUpdateFieldEvent	An opaque structure used for TSynTable.UpdateFieldEvent method	279

**ESynException = class(Exception)**

*Generic parent class of all custom Exception types of this unit*

**ETableDataException = class(Exception)**

*Exception raised by all TSynTable related code*

**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: integer); reintroduce; virtual;`

*Initialize the internal conversion engine*

**function** `AnsiBufferToRawUTF8(Source: PAnsiChar; SourceChars: Cardinal): RawUTF8; overload;`

*Direct conversion of a PAnsiChar buffer into a UTF-8 encoded string*

- will call `AnsiBufferToUnicode()` overloaded virtual method

**function** `AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal): 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

**function** `AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal): PUTF8Char; overload; virtual;`

*Direct conversion of a PAnsiChar buffer into a UTF-8 encoded buffer*

- Dest^ buffer must be reserved with at least `SourceChars*3` bytes
- a #0 char is appended at the end (and result will point to it)
- this default implementation will use the Operating System APIs

**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 an 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(Source: PAnsiChar; SourceChars: Cardinal): SynUnicode;`

*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;

*Convert any Ansi Text into an UTF-8 encoded String*  
- internally calls AnsiBufferToUTF8 virtual method

**class function** Engine(aCodePage: integer): 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

**function** RawUnicodeToAnsi(**const** Source: RawUnicode): RawByteString;

*Convert any Unicode-encoded String into Ansi Text*  
- internally 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** UnicodeBufferToAnsi(Source: PWideChar; SourceChars: Cardinal): RawByteString; overload;

*Direct conversion of an Unicode buffer into an Ansi Text*

**function** UTF8BufferToAnsi(Dest: PAnsiChar; Source: PUTF8Char; SourceChars: Cardinal): PAnsiChar; overload; **virtual**;

*Direct conversion of an UTF-8 encoded buffer into a PAnsiChar buffer*  
- Dest^ buffer must be reserved with at least SourceChars bytes

**function** UTF8BufferToAnsi(Source: PUTF8Char; SourceChars: Cardinal): RawByteString; overload;

*Convert any UTF-8 encoded buffer into Ansi Text*  
- internally calls UTF8BufferToAnsi virtual method

**function** UTF8ToAnsi(**const** UTF8: RawUTF8): RawByteString;

*Convert any UTF-8 encoded String into Ansi Text*  
- internally calls UTF8BufferToAnsi virtual method

**property** CodePage: Integer **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: integer); override;`

*Initialize the internal conversion engine*

**function** `AnsiBufferToUnicode(Dest: PWideChar; Source: PAnsiChar; SourceChars: Cardinal): PWideChar; override;`

*Direct conversion of a PAnsiChar buffer into an Unicode buffer*

- `Dest` buffer must be reserved with at least `SourceChars*2` bytes

**function** `AnsiBufferToUTF8(Dest: PUTF8Char; Source: PAnsiChar; SourceChars: Cardinal): 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
- a `#0` char is appended at the end (and result will point to it)

**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: integer): 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** `IsValidAnsiU(UTF8Text: 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

**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** AnsiToWide: TWordDynArray **read** fAnsiToWide;

*Direct access to the Ansi-To-Unicode lookup table*

- use this array like AnsiToWide: 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('?')

**TSynSoundEx = object(TObject)**

*Fast search of a text value, using the Soundex searching 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: TSynSoundExPronunciation=sndxEnglish): boolean;

*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^

**TSortedWordArray = object(TObject)**

*Used to store and retrieve Words in a sorted array*

- 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** 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

**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.

**function** Add(const Elem): integer;

*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 will be a bit slower than e.g. with a TList: the list is reallocated every time a record is added - but in practice, with FastMM4 or SynScaleMM, there is no big speed penalty - for even better speed, you can also specify an external count variable in Init(...,@Count) method

**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): RawByteString; overload;

*Load an array element as saved by the ELEMsave method*

- this overloaded method will retrieve the element as a memory buffer and caller MUST call ElemLoadClear() method to finalize its content

**function** ElemLoadFind(Source: PAnsiChar): 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** ElemSave(const Elem): RawByteString;

*Save an array element into a serialized buffer*

- 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 Find(i+10) e.g.)

**function** Find(const Elem; const aIndex: TIntegerDynArray; aCompare: TDynArraySortCompare): integer; 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 binary search
- if an indexed lookup is not supplied (i.e aIndex=nil), this function will use slower but accurate iterating search
- warning; the lookup index should be synchronized if array content is modified (in case of adding or deletion)

**function** Find(const Elem): integer; 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 binary search
- if the array is not sorted, it will use slower 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 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  $\geq 0$ )
- 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 binary search
- if the array is not sorted, it will use slower 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(var 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  $\geq 0$ )
- return the index deleted (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast binary search
- if the array is not sorted, it will use slower 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  $\geq 0$ )
- return the index found (0..Count-1), or -1 if Elem was not found
- if the array is sorted, it will use fast binary search
- if the array is not sorted, it will use slower 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 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 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 binary search
- if the array is not sorted, it will use slower 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** IndexOf(const Elem): integer;

*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 eElement: 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 shorstring, or a record with byte or word fields with {\$A+}): in this case, the padding data (i.e. the bytes between the aligned feeds 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): PAnsiChar;

*Load the dynamic array content from a memory buffer*

- return nil if the Source buffer is incorrect (invalid type or internal checksum e.g.)
- in case of success, return the memory buffer pointer just after the read content
- return a pointer at the end of the data read from Source, nil on error

**function** LoadFromJSON(P: PUTF8Char): 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 TIntegerDynArray, 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 preprocessed 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
- warning: the content of P^ will be modified during parsing: please make a local copy if it will be needed later

**function** New: integer;

*Add an element to the dynamic array*

- this version add a void element to the array, and returns its index

**function** SaveTo: RawByteString; overload;

*Save the dynamic array content into a RawByteString*

**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

**function** SaveToLength: integer;

*Compute the number of bytes needed to save a dynamic array content*

**procedure** AddArray(const DynArray; aStartIndex: integer=0; aCount: integer=-1);

*Add elements from a given dynamic array*

- the supplied source DynArray MUST be of the same exact type as the current used for this TDynArray
- 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*

**procedure** Copy(Source: TDynArray);

*Set all content of one dynamic array to the current array*

- both must be of the same exact type

**procedure** CreateOrderedIndex(**var** aIndex: TIntegerDynArray; aCompare: TDynArraySortCompare);

*Sort the dynamic array elements using a lookup array of indexes*

- it won't change the dynamic array content: only create or update the given integer lookup array, using the specified comparison function
- you should provide 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** Delete(Index: Integer);

*Delete one item inside the dynamic array*

- the deleted element is finalized if necessary

**procedure** ElemClear(**var** Elem);

*Will reset the element content*

**procedure** ElemCopy(**const** A; **var** B);

*Will copy one element content*

**procedure** ElemLoad(Source: PAnsiChar; **var** Elem); overload;

*Load an array element as saved by the ELEMsave method*

- 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.)

**procedure** ElemLoadClear(**var** ElemLoaded: RawByteString);

*Release memory allocated by ELEMload(): RawByteString*



```
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);
  (...)
```

- 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 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 would 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..djSynUnicode comparison

```
procedure Insert(Index: Integer; 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

```
procedure Reverse;
```

*Will reverse all array elements, in place*

**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
- 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;

*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** Void;

*Initialize the wrapper to point to no dynamic array*

**property** Capacity: integer **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

**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, SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArrayInt64, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiStringI, SortDynArrayString, SortDynArrayStringI, SortDynArrayUnicodeString, SortDynArrayUnicodeStringI

**property** Count: integer **read** GetCount **write** SetCount;

*Retrieve or set the number of elements of the dynamic array*

- same as length(DynArray) or SetLength(DynArray)

**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 use binary search in an unsorted array, and miss its purpose

**TSynHash = record**

*Internal structure used to store one item hash*

- used e.g. by TDynArrayHashed or TObjectHash via TSynHashDynArray

**Hash:** cardinal;

*Unsigned integer hash of the item*

**Index:** cardinal;

*Index of the item in the main storage array*

**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 internal FindHashedForAdding method
- this version will set the field content with the unique value
- returns a pointer to the newly added element

**function** FindHashed(const Elem): integer;

*Search for an element value inside the dynamic array using hashing*

- Elem should be of the same exact type than the dynamic array, or at least matches the fields used by both the hash function and Equals method: e.g. if the searched/hashed field in a record is a string as first field, you may use a string variable as Elem: other fields will be ignored
- returns -1 if not found, or the index in the dynamic array if found

**function** FindHashedAndDelete(var Elem): 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
- this will rehash all content: this method could be slow in the current implementation
- warning: Elem must be of the same exact type than the dynamic array, and must refer to a variable (you can't write FindHashedAndDelete(i+10) e.g.)

**function** FindHashedAndFill(var Elem): 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
- 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** FindHashedAndUpdate(**var** 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*

- 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
- warning: Elem must be of the same exact type than the dynamic array, and must refer to a variable (you can't write FindHashedAndUpdate(i+10) e.g.)

**function** FindHashedForAdding(**const** Elem; **out** wasAdded: boolean; aHashCode: cardinal=0): integer;

*Search for an element value inside the dynamic array using hashing, and add a void entry to the array if was not found*

- this method will use hashing for fast retrieval
- Elem should be of the same exact type than the dynamic array, or at least matches the fields used by both the hash function and Equals method: e.g. if the searched/hashed field in a record is a string as first field, you may use a string variable as Elem: other fields will be ignored
- 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 for filling the newly created entry in the array

**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 kr32() by default - i.e. the well known Kernighan & Ritchie hash function
- if CaseInsensitive 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, using TDynArrayKind kind of first field
- djNone and djCustom are too vague, and would 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..djSynUnicode comparison

**procedure** ReHash(aHasher: TOnDynArrayHashOne=nil);

*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

**property** EventCompare: TEventDynArraySortCompare **read** fEventCompare **write** fEventCompare;

*Alternative event-oriented Compare function to be used for Sort and Find*

- will be used instead of Compare, to allow object-oriented callbacks

**property** Hash[aIndex: Integer]: Cardinal **read** GetHashFromIndex;

*Retrieve the hash value of a given item, from its index*

**TObjectHash = class(TObject)**

*Abstract class able to use hashing to find an object in O(1) speed*

- all protected abstract methods shall be overridden and implemented

**function** Find(Item: TObject): integer;

*Search one item in the internal hash array*

**function** JustAdded: boolean;

*To be called when an item is added*

- return FALSE if this item is already existing (i.e. insert error)
- return TRUE if has been added to the internal hash table
- the index of the latest added item should be Count-1

**procedure** Invalidate;

*To be called when an item is modified*

- for Delete/Update will force a full rehash on next Find() call

**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 internally 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
- 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 :(

**Count: integer;**

*The number of Name/Value pairs*

**List: TSynNameValueItemDynArray;**

*The internal Name/Value storage*

**function Find(const aName: RawUTF8): integer;**

*Search for a Name, return the index in List*

**function Initialized: boolean;**

*Returns true if the Init() method has been called*

**function Value(const aName: RawUTF8): RawUTF8;**

*Search for a Name, return the associated Value*

**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 Init(aCaseSensitive: boolean);**

*Initialize the storage*

**property BlobData: RawByteString read GetBlobData write SetBlobData;**

*Can be used to set or retrieve all stored data as one BLOB content*

## **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 escape

*Used for DI-2.1.2 (page 830).*

**constructor Create(aStream: TStream; aBufSize: integer=8192);**

*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 830).*

**constructor** CreateOwnedStream;

*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

**destructor** Destroy; **override**;

*Release fStream is is owned*

**function** Text: RawUTF8;

*Retrieve the data as a string*

- only works if the associated Stream Inherits from TMemoryStream or TRawByteStringStream: will return "" if it is not the case

**procedure** Add(Value: integer); **overload**;

*Append an Integer Value as a String*

**procedure** Add(P: PUTF8Char; Len: PtrInt; Escape: TTextWriterKind); **overload**;

*Write some #0 ended UTF-8 text, according to the specified format*

**procedure** Add(Value: double); **overload**;

*Append a floating-point Value as a String*

**procedure** Add(Value: Int64); **overload**;

*Append an Integer Value as a String*

**procedure** Add(c: AnsiChar); **overload**;

*Append one char to the buffer*

**procedure** Add(P: PUTF8Char; Escape: TTextWriterKind); **overload**;

*Write some #0 ended UTF-8 text, according to the specified format*

**procedure** Add(c1,c2: AnsiChar); **overload**;

*Append two chars to the buffer*

**procedure** Add2(Value: integer);

*Append an Integer Value as a 2 digits String with comma*

**procedure** Add3(Value: integer);

*Append an Integer Value as a 3 digits String without any added comma*

**procedure** Add4(Value: integer);

*Append an Integer Value as a 4 digits String with comma*

**procedure** AddBinToHex(P: Pointer; Len: 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 encode it in big-endian order (most-significant byte first): use it for display

- up to 128 bytes may be converted

**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 chars
```

```
procedure AddCSV(const Values: array of RawUTF8); overload;  
  Append an array of RawUTF8 as CSV
```

```
procedure AddCSV(const Integers: array of Integer); overload;  
  Append an array of integers as CSV
```

```
procedure AddCSV(const Doubles: array of double); overload;  
  Append an array of doubles as CSV
```

```
procedure AddCurr64(Value: PInt64); 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 AddCurr64(const Value: currency); overload;  
  Append a Currency from its Int64 in-memory representation
```

```
procedure AddCurrentLogTime;  
  Append the current date and time, in a log-friendly format  
  - e.g. append '20110325 19241502 '  
  - this method is very fast, and avoid most calculation or API calls
```

```
procedure AddDateTime(Value: PDateTime; FirstChar: AnsiChar='T'; QuoteChar:  
AnsiChar=#0); overload;  
  Append a TDateTime value, expanded as Iso-8601 encoded text
```

```
procedure AddDateTime(const Value: TDateTime); overload;  
  Append a TDateTime value, expanded as Iso-8601 encoded text
```



**procedure** AddDynArrayJSON(const DynArray: TDynArray);

*Append a dynamic array content as UTF-8 encoded JSON array*

- expect a dynamic array TDynArray wrapper as incoming parameter
- TIntegerDynArray, TInt64DynArray, TCardinalDynArray, TDoubleDynArray, TCurrencyDynArray, TWordDynArray and TByteDynArray will be written as numerical JSON values
- TRawUTF8DynArray, TWinAnsiDynArray, TRawByteStringDynArray, TStringDynArray, TWideStringDynArray, 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 \uFFFF0 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 '["\uFFFF0base64encodedbinary"]'

**procedure** AddFieldName(const FieldName: RawUTF8);

*Append a RawUTF8 property name, as "FieldName":'*

**procedure** AddFloatStr(P: PUTF8Char);

*Append a floating-point text buffer*

- will correct on the fly '.5' -> '0.5' and '-.5' -> '-0.5'

**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);

*Append an Instance name and pointer, as 'TObjectList(00425E68)' + SepChar*

- Instance must be not nil

**procedure** AddJSONEscape(P: Pointer; Len: PtrInt=0); overload;

*Append some UTF-8 encoded chars to the buffer*

- if Len is 0, Len is calculated from zero-ended char
- escapes chars according to the JSON RFC

*Used for DI-2.1.2 (page 830).*

**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 830).*

**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}'`

- note that cardinal values should be type-casted to `Int64()` (otherwise the integer mapped value will be transmitted, therefore wrongly)

*Used for DI-2.1.2 (page 830).*

**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)))`

- if Len is 0, Len is calculated from zero-ended char

- 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 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*

**procedure AddNoJSONEscape(P: Pointer; Len: integer=0);**

*Append some chars to the buffer*

- if Len is 0, Len 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 AddNoJSONEscapeW(P: 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 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** 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** 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 hexa chars, ready to be displayed*

**procedure** AddPropName(const PropName: ShortString);

*Append a ShortString property name, as "PropName":'*

**procedure** AddRecordJSON(const Rec; TypeInfo: pointer);

*Append a dynamic array content as UTF-8 encoded JSON*

- 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

**procedure** AddShort(const Text: ShortString);

*Append a ShortString*

**procedure** AddString(const Text: RawUTF8);

*Append a String*

**procedure** AddTimeLog(Value: PInt64);

*Append a TTimeLog value, expanded as Iso-8601 encoded text*

**procedure** AddU(Value: cardinal);

*Append an Unsigned Integer Value as a String*

**procedure** AddW(P: PWord; Len: PtrInt; Escape: TTextWriterKind);

*Write some #0 ended Unicode text as UTF-8, according to the specified format*

**procedure** CancelAll;

*Rewind the Stream to the position when Create() was called*

**procedure** CancelLastChar;

*The last char appended is canceled*

**procedure** CancelLastComma;

*The last char appended is canceled if it was a ','*

**procedure** Flush;

*Write pending data to the Stream*

- will append the internal memory buffer to the Stream

- if you don't call Flush, some pending characters may be not yet copied to the Stream: you should call it before using the Stream property

```
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)

```
procedure WrBase64(P: PAnsiChar; Len: cardinal; withMagic: boolean);
```

*Write some data Base64 encoded*

- if withMagic is TRUE, will write as ""\uFFFF0base64encodedbinary"

```
procedure WrHex(P: PAnsiChar; Len: integer);
```

*Write some data as hexa chars*

```
procedure WriteObject(Value: TObject; HumanReadable: boolean=false;
DontStoreDefault: boolean=true; FullExpand: boolean=false); 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 FullExpand=true) or string

```
procedure WrRecord(const Rec; TypeInfo: pointer);
```

*Write some record content as binary, Base64 encoded with our magic prefix*

```
property Stream: TStream read fStream write fStream;
```

*The internal TStream used for storage*

- you should call the Flush method before using this TStream content, to flush all pending characters to the stream

```
property TextLength: integer read GetLength;
```

*Count of add byte to the stream*

```
TJSONWriter = class(TTextWriter)
```

*Simple writer to a Stream, specialized for the JSON format and SQL export*

- use an internal buffer, faster than string+string

```
ColNames: TRawUTF8DynArray;
```

*Used internally to store column names and count for AddColumns*

```
constructor Create(aStream: TStream; Expand, withID: boolean; const Fields:
TSQLFieldBits=[]); 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(aKnownRowCount: integer=0);

*Write or init field names for appropriate JSON Expand later use*

- ColNames[] must have been initialized before calling this procedure
- if aKnownRowCount 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** 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** FieldMax: integer **read** fFieldMax **write** fFieldMax;

*Read-Only access to the higher field index to be stored*

- i.e. the highest bit set in Fields

**property** Fields: TSQLFieldBits **read** fFields **write** 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 usefull Row of data; i.e. 'val11' position in:*

```
{ "fieldCount":1,"values":["col1","col2",val11,"val12",val21,..] }
```

**property** WithID: boolean **read** fWithID **write** fWithID;

*Is set to TRUE if the ID field must be appended to the resulting JSON*

**TSynCache = class(TObject)**

*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)

**constructor** Create(aMaxCacheRamUsed: cardinal=16384\*1024);

*Initialize the internal storage*

- aMaxCacheRamUsed can set the maximum RAM to be used for values, in bytes (default is 16 MB per cache): cache will be reset when so much value will be reached

**function** Find(const aKey: RawUTF8; aResultTag: PPtrInt): RawUTF8;

*Find a Key in the cache entries*

- return "" if nothing found
- return the associated Value otherwise, and the associated integer tag if aResultTag address is supplied

**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

**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

**property** Count: integer **read** fNameValue.Count;

*Number of entries in the cache*

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

TObjectListPropertyHashed = **class**(TObjectListHashedAbstract)

*This class will hash and search for a sub property of the stored objects*

**constructor** Create(aSubPropAccess: TObjectListPropertyHashedAccessProp;  
 aHashElement: 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 SQLite3Commons unit as example:

```
function WeakZeroClassSubProp(aObject: TObject): TObject;
begin
  result := TSetWeakZeroInstance(aObject).fInstance;
end;
```

- by default, aHashElement/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

**TRawUTF8List = class**(TObject)

*This class is able to emulate a TStringList with our native UTF-8 string type*  
- cross-compiler, from Delphi 6 up to XE2, i.e is Unicode Ready

**constructor** Create;

*Initialize the class instance*

**function** Add(const aText: RawUTF8): PtrInt;

*Store a new RawUTF8 item*  
- returns -1 and raise no exception in case of self=nil

**function** AddObject(const aText: RawUTF8; aObject: TObject): PtrInt;

*Store a new RawUTF8 item, and its associated TObject*  
- returns -1 and raise no exception in case of self=nil

**function** DeleteFromName(const Name: RawUTF8): PtrInt;

*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

**function** Get(Index: PtrInt): RawUTF8;

*Get a stored RawUTF8 item*  
- returns '' and raise no exception in case of out of range supplied index

**function** GetObject(Index: PtrInt): TObject;

*Get a stored Object item*  
- returns nil and raise no exception in case of out of range supplied index

**function** GetText(const Delimiter: RawUTF8=#13#10): RawUTF8;

*Retrieve the all lines, separated by the supplied delimiter*

**function** GetValueAt(Index: PtrInt): RawUTF8;

*Access to the Value of a given 'Name=Value' pair*

**function** IndexOf(const aText: RawUTF8): PtrInt; **virtual**;

*Find a RawUTF8 item in the stored Strings[] list*  
- this search is case sensitive

**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

**function** IndexOfObject(aObject: TObject): PtrInt;

*Find a TObject item index in the stored Objects[] list*

**procedure** AddRawUTF8List(List: TRawUTF8List);

*Append a specified list to the current content*



**procedure** BeginUpdate;

*The OnChange event will be raised only when EndUpdate will be called*

**procedure** Clear;

*Erase all stored RawUTF8 items*

**procedure** Delete(Index: PtrInt);

*DeLete a stored RawUTF8 item, and its associated TObject*  
 - raise no exception in case of out of range supplied index

**procedure** EndUpdate;

*Call the OnChange event if changes occurred*

**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

**procedure** SetText(const aText: RawUTF8; const Delimiter: RawUTF8=#13#10);

*Set all lines, separated by the supplied delimiter*

**procedure** UpdateValue(const Name: RawUTF8; var Value: RawUTF8; ThenDelete: boolean);

*Update Value from an existing Name=Value, then optionally deLete the entry*

**property** Capacity: PtrInt **read** GetCapacity **write** SetCapacity;

*Set or retrieve the current memory capacity of the RawUTF8 list*

**property** Count: PtrInt **read** GetCount;

*Return the count of stored RawUTF8*

**property** ListPtr: PUtf8CharArray **read** GetListPtr;

*Direct access to the memory of the RawUTF8 array*

**property** Names[Index: PtrInt]: RawUTF8 **read** GetName;

*Retrieve the corresponding Name when stored as 'Name=Value' pairs*

**property** NameValueSep: AnsiChar **read** fNameValueSep **write** fNameValueSep;

*The char separator between 'Name=Value' pairs*  
 - equals '=' by default

**property** ObjectPtr: PPointerArray **read** GetObjectPtr;

*Direct access to the memory of the Objects array*

**property** Objects[Index: PtrInt]: TObject **read** GetObject **write** PutObject;

*Get or set a Object item*  
 - returns nil and raise no exception in case of out of range supplied index

**property** OnChange: TNotifyEvent **read** fOnChange **write** fOnChange;

*Event triggered when an entry is modified*

**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



**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)

**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

TRawUTF8ListHashed = **class**(TRawUTF8List)

*A TRawUTF8List which will use an internal hash table for faster IndexOf()*

- this is a rather rough implementation: all values are re-hashed after change: but purpose of this class is to allow faster access of a static list of identifiers

**constructor** Create;

*Initialize the class instance*

**function** IndexOf(**const** aText: RawUTF8): PtrInt; **override**;

*Find a RawUTF8 item in the stored Strings[] list*

- this overridden method will update the internal hash table (if needed), then use it to retrieve the corresponding matching index

**property** CaseInsensitive: boolean **read** fHashCaseInsensitive **write** SetHashCaseInsensitive;

*Specify if the IndexOf() hashed search is case sensitive or not*

- is FALSE by default, as for the default TRawUTF8List.IndexOf() method

TMemoryMap = **object**(TObject)

*Handle memory mapping of a file content used to store and retrieve Words in a sorted array*

**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: cardinal=0; aCustomOffset: Int64=0): boolean; **overload**;

*Map the corresponding file handle*

- if aCustomSize and aCustomOffset are specified, the corresponding map view is created (by default, will map whole file)

**procedure** Map(aBuffer: pointer; aBufferSize: cardinal); **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** Size: cardinal **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[]

**destructor** Destroy; **override**;

*Release the memory map and internal LinePointers[]*

**function** LineContains(**const** aUpperSearch: RawUTF8; aIndex: Integer): Boolean;

*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

**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

**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

**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: integer); 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: cardinal=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 is created (by default, will map whole file)

**constructor** Create(const aFileName: TFileName; aCustomSize: cardinal=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 is created (by default, will map whole file)

**destructor** Destroy; **override**;

*Release any internal mapped file instance*

**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

**constructor** Create(const aFileName: TFileName; BufLen: integer=65536); overload;

*Initialize the buffer, and specify a file 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(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** CreateInMemoryStream;

*Initialize the buffer, using an internal TMemoryStream*

- use Flush then TMemoryStream(Stream) to retrieve its content

**destructor** Destroy; override;

*Release internal TStream (after AssignToHandle call)*

**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

**procedure** Write(const Text: RawUTF8); overload;

*Append some UTF-8 encoded text at the current position*

**procedure** Write(Data: pointer; DataLen: integer); overload;

*Append some data at the current position*

**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** 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** 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** 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 32-bit variable-length integer encoding (in this case, a fixed-sized record storage is also handled separately)

**property** Stream: TStream read fStream;

*The associated writing stream*

**property** TotalWritten: Int64 read fTotalWritten;

*Get the byte count written since last FLush*

**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)  
- 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** CurrentMemory: pointer;

*Retrieve the current in-memory pointer*

- if file was not memory-mapped, returns nil

**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: RawUTF8): integer; overload;

*Read some UTF-8 encoded text at the current position*

- returns the resulting text length, in bytes

**function** Read(Data: pointer; DataLen: integer): integer; overload;

*Read some bytes from the given reading position*

- returns the number of bytes which was read

**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

**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** 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))

**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: 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

**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

**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);

*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, methods will use default slower file API

**procedure** OpenFrom(aBuffer: pointer; aBufferSize: cardinal); overload;

*Initialize the buffer from an already existing memory block*  
- may be e.g. a resource or a TMemoryStream

**property** FileSize: Int64 read fMap.fFileSize;

*Read-only access to the global file size*

**TSynFilterOrValidate = class(TObject)**

*Will define a filter 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='');

*Initialize the filter or validation instance*

**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 adress 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 SQLite3i18n 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"}'
```

- 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 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)**

*Grep-like case-sensitive pattern validation of a Record field content (typically a TSQLRecord)*

- parameter is NOT JSON encoded, but is some basic grep-like 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
- [!abc] Matches anything but a or b or c at that position
- [a-e] Matches a through e 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

**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]'
- this method will implement both TSynValidatePattern and TSynValidatePatternI, checking the current class

**TSynValidatePatternI = class(TSynValidatePattern)**

*Grep-like case-insensitive pattern validation of a Record field content (typically a TSQLRecord)*

- parameter is NOT JSON encoded, but is some basic grep-like pattern
- same as TSynValidatePattern, but is NOT case sensitive

**TSynValidateText = class(TSynValidate)**

*Text validation to be applied to a Record field content (typically a TSQLRecord)*

- expects optional JSON parameters of the allowed text length range as  
 '{"MinLength":5,"MaxLength":10,"MinAlphaCount":1,"MinDigitCount":1,"MinPunctCount":1,"MinLowerCount":1,"MinUpperCount":1}'
- default MinLength value is 1, MaxLength is maxInt: so you can specify a blank TSynValidateText to avoid any void textual field
- 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

**constructor** Create(**const** aParameters: RawUTF8='');

*Initialize the validation instance*

**function** Process(aFieldIndex: integer; **const** Value: RawUTF8; **var** ErrorMsg: string): boolean; **override**;

*Perform the text length validation action to the specified value*

**property** MaxLeftTrimCount: cardinal read fProps[9] write fProps[9];

*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 Unicode glyphs, not with UTF-8 encoded char count

- default is maxInt, i.e. no maximum length is set

**property** MaxRightTrimCount: cardinal read fProps[10] write fProps[10];

*Maximal space count allowed on the Right side*

- default is maxInt, i.e. any Right space allowed

**property** MaxSpaceCount: cardinal read fProps[8] write fProps[8];

*Maximal space count inside the value text*

- default is maxInt, i.e. any space number allowed

**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** MinDigitCount: cardinal read fProps[3] write fProps[3];

*Minimal digit character [0-9] 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 Unicode glyphs, not with UTF-8 encoded char count

- 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

**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":10,"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 filter to be applied to a Record field content (typically a TSQLRecord)*

- 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 filtering action to the specified value*

- the value is converted into UTF-8 text, as expected by TPropInfo.GetValue / TPropInfo.SetValue  
e.g.

**TSynFilterUpperCase = class(TSynFilter)**

*A custom filter which will convert the value into 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)**

*A custom filter which will convert the value into 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)**

*A custom filter which will convert the value into 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)**

*A custom filter which will convert the value into 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)**

*A custom filter which will 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 trimming conversion to the specified value*

**Iso8601 = object(TObject)**

*Usefull object to type cast TTimeLog type into Iso-8601 or TDateTime*

- typecast TTimeLog from PIso8601, not with Iso8601(Time).From():
- PIso8601(@aTime)^.From()
- to get current time, simply use Time := Iso8601Now
- since Iso8601.Value is bit-oriented, you can't just use add or subtract two TTimeLog values when doing such date/time computation: use a TDateTime temporary conversion in such case

**Value: Int64;**

*The value itself*

- 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..38 = Year (0..4095)

**function** Text(Dest: PUTF8Char; Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): integer; **overload;**

*Convert to Iso-8601 encoded text*

**function** Text(Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): RawUTF8; **overload;**

*Convert to Iso-8601 encoded text*

**function** ToDate: TDateTime;

*Convert to a delphi Date*

**function** ToDateTime: TDateTime;

*Convert to a delphi Date and Time*

**function** ToTime: TDateTime;

*Convert to a delphi Time*

**procedure** Expand(out Date: TSystemTime);

*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** FromNow;

*Fill Value from current local system Date and Time*

**TFileVersion = class**(TObject)

*To have existing RTTI for published properties used to retrieve version information from any EXE*

**Build: Integer;**

*Executable release build number*

**BuildYear: integer;**

*Build year of this exe file*

**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*

**Release: Integer;**

*Executable release version number*

**constructor** Create(const FileName: TFileName; DefaultVersion: integer);

*Retrieve application version from exe file name*

*- DefaultVersion is used if no information Version was included into the executable resources (on compilation time)*

*- to retrieve version information from current executable, just call ExeVersionRetrieve function, then use ExeVersion global variable*

**function** Version32: integer;

*Retrieve the version as a 32 bits integer with Major.Minor.Release*

**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+

TOSVersionInfoEx = **record**

*Not defined in older Delphi versions*

THeapMemoryStream = **class**(TMemoryStream)

*To be used instead of TMemoryStream, for speed*

- allocates memory from Delphi heap (i.e. FastMM4/SynScaleMM) and not GlobalAlloc()

- uses bigger growing size of the capacity

TSynTableStatement = **class**(TObject)

*Used to parse a SELECT SQL statement*

- handle basic REST commands, no complete 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

- handle also basic "SELECT Count(\*) FROM TableName;" SQL statement

**Fields:** TSQLFieldBits;

*The fields selected for the SQL statement*

**TableName:** RawUTF8;

*The retrieved table name*

**WhereField:** integer;

*The index of the field used for the WHERE clause*

- SYNTABLESTATEMENTWHEREID=0 is ID, 1 for field # 0, 2 for field #1, and so on... (i.e.

WhereField = RTTI field index +1)

- equal SYNTABLESTATEMENTWHEREALL=-1, means all rows must be fetched (no WHERE clause: "SELECT \* FROM TableName")

- if SYNTABLESTATEMENTWHERECOUNT=-2, means SQL statement was "SELECT Count(\*) FROM TableName"

**WhereValue:** RawUTF8;

*The value used for the WHERE clause*

**WhereValueInteger:** integer;

*An integer representation of WhereValue (used for ID check e.g.)*

**WhereValueSBF:** TSBFString;

*Used to fast compare with SBF binary compact formatted data*

**WithID:** boolean;

*Is TRUE if ID/RowID was set in the WHERE clause*

**Writer: TJSONWriter;**

*Optional associated writer*

**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  
 Fields, TableName, WhereField, WhereValue*

- the supplied GetFieldIndex() method is used to populate the Fields and WhereField parameters
- SimpleFieldsBits is used for '\*' field names
- WhereValue is left '' if the SQL statement is not correct
- if WhereValue is set, the caller must check for TableName to match the expected value, then use the WhereField value to retrieve the content
- if FieldProp is set, then the WhereValueSBF property is initialized with the SBF equivalence of the WhereValue

**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: TObjectList;**

*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:** TObjectList;

*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** 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** 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 a GetIteraring(), because will use binary search using the OrderedIndex[] array

- returns the resulting indexes as 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 formatted 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: 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** OrderedIndexRefresh;

*Will force refresh the OrderedIndex[] array*  
- to be called e.g. if OrderedIndexNotSorted = TRUE, if you want to access to the  
OrderedIndex[] array

**procedure** SaveTo(WR: TFileBufferWriter);

*Save entry to a specified file writer*

**property** SBFDefault: TSBFString **read** fDefaultFieldData;  
*Some default SBF compact binary format content*

**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 TSQLRest 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** 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 to the corresponding data address of a given field*

**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, handy 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  $\geq 0$ , 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  $\geq 0$ , aNewIndex refers to a just created item (i.e. called in case of a data Add)
- if aOldIndex is  $\geq 0$  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** 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** 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 fUnique 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

**TPrecisionTimer = object(TObject)**

*High resolution timer (for accurate speed statistics)*

**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: RawUTF8;

*Stop the timer, returning the time elapsed, with appened time resolution (us,ms,s)*

**function** Time: RawUTF8;

*Return 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*

**procedure** Start;

*Start the high resolution timer*

**TSynTest = class(TObject)**

*A generic class for both tests suit and cases*

- 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
- see @<http://synopse.info/forum/viewtopic.php?pid=277>

**constructor** Create(const Ident: string = '');

*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(aMethod: TSynTestEvent; const aName: string);

*Register a specified test to this class instance*

**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 uncamed 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** TestMethod[Index: integer]: TSynTestEvent read GetTestMethod;

*Get the event of a specified test*

- Index range is from 0 to Count-1 (including)

**property** TestName[Index: integer]: string read GetTestName;

*Get the name of a specified test*

- Index range is from 0 to Count-1 (including)

**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 = ''); 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

**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** 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:

```
if CheckNot(A<>10) then exit;
```

**function** CheckSame(const Value1, Value2: double; const Precision: double=1E-12; const msg: string = ''): Boolean;

*Used by the published methods to run a test assertion about two double values*



**class function** RandomString(CharCount: Integer): RawByteString;

*Create a temporary string random content*  
 - it somewhat faster if CharCount is a multiple of 5

**class function** RandomUTF8(CharCount: Integer): RawUTF8;

*Create a temporary string random content*  
 - it somewhat faster if CharCount is a multiple of 5

**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** TestFailed(**const** msg: string);

*This method is triggered internally - 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 uncamed text from the class name

**property** MethodIndex: integer **read** fMethodIndex;

*The index of the associated Owner.TestMethod[] which created this test*

**property** Owner: TSynTests **read** fOwner;

*The test suit which owns this test case*

**property** TestCaseIndex: integer **read** fTestCaseIndex;

*The index of the test case, starting at 0 for the associated MethodIndex*

**TSynTests = class**(TSynTest)

*A class used to run a suit of test cases*

**CustomVersions: string;**

*If set to a text file address, some debug messages will be reported to this text file*

- for example, use the following line to report to the console:

`ToConsole := @Output;`

- you can also use the SaveToFile() method to create an external file 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 = '');

*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

**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
- the TestCase[] list is created first, by running all published methods, which must call the AddCase() method above to register test cases
- the Failed[] list is cleared at the beginning of the run
- Assertions and AssertionsFailed properties are reset and computed during the run

**procedure** AddCase(const TestCase: array of TSynTestCaseClass); **overload;**

*Register a specified Test case instance*

- an instance of the supplied class is created, and will be freed by TSynTests.Destroy
- 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(TestCase: TSynTestCase); **overload;**

*Register a specified Test case instance*

- all these instances will be freed by the TSynTests.Destroy
- 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.Create(self));

**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** FailedCase[Index: integer]: TSynTestCase read GetFailedCase;

*Retrieve the TSynTestCase instance associated with this failure*

- returns nil if this failure was not triggered by a TSynTestCase, but directly by a method

**property** FailedCaseIdent[Index: integer]: string read GetFailedCaseIdent;

*Retrieve the ident of the case test associated with this failure*

**property** FailedCount: integer read GetFailedCount;

*Number of failed tests after the last call to the Run method*

**property** FailedMessage[Index: integer]: string read GetFailedMessage;

*Retrieve the error message associated with this failure*

**property** TestCase[Index: integer]: TSynTestCase read GetTestCase;

*An array containing all registered Test case instances*

- Test cases are registered by the AddCase() method above, mainly by published methods of the children

- Test cases instances are freed by TSynTests.Destroy

**property** TestCaseCount: Integer read GetTestCaseCount;

*The number of items in the TestCase[] array*

**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 833).*

**procedure** Bits;

*The low-level bit management functions*

**procedure** Curr64;

*The new fast Currency to/from string conversion*

**procedure** FastStringCompare;

*Test StrIComp() and AnsiIComp() functions*

**procedure** IniFiles;

*The fast .ini file content direct access*

**procedure** Iso8601DateAndTime;

*The ISO-8601 date and time encoding*

- test especially the conversion to/from text

**procedure** NumericalConversions;

*Low level fast Integer or Floating-Point to/from string conversion*

- especially the RawUTF8 or PUTF8Char relative versions

**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** Unicode\_UTF8;

*Test UTF-8 and Win-Ansi conversion (from or to, through RawUnicode)*

**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** \_IdemPropName;

*Test IdemPropName() and IdemPropNameU() functions*

**procedure** \_IsMatch;

*Test IsMatch() function*

**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** \_TSynCache;

*Test the TSynCache class*

**procedure** \_TSynFilter;

*Low-level TSynFilter classes*

**procedure** \_TSynLogFile;

*Low-level TSynLogFile class*

**procedure** \_TSynTable;

*Test TSynTable class and TSynTableVariantType new variant type*

**procedure** \_TSynValidate;

*Low-level TSynValidate classes*

**TSynMapSymbol = record**

*A debugger symbol, as decoded by TSynMapFile from a .map file*

**Name:** RawUTF8;

*Symbol internal name*

**Start:** cardinal;

*Starting offset of this symbol in the executable*

**Stop:** cardinal;

*End offset of this symbol in the executable*

## **TSynMapUnit = 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 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** FindLocation(aAddr: Cardinal): RawUTF8;

*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

**function** FindSymbol(aAddr: cardinal): integer;

*Retrieve a symbol according to an absolute code address*

**function** FindUnit(aAddr: cardinal; out LineNumber: integer): integer;

*Retrieve an unit and source line, according to an absolute code address*

**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; Addr: PtrUInt);

*Add some debugging information according to the specified 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 address as hexadecimal

**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** 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*

**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=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; aName: PWinAnsiChar; aTypeInfo: pointer; var aValue; Instance: TObject=nil); 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; const Text: RawUTF8; Instance: TObject=nil);  
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 - TSQLLog will write the object JSON content

```
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:

```
with TSynLogDB.Family do begin  
  Level := LOG_VERBOSE;  
  PerThreadLog := true;  
  DestinationPath := 'C:\Logs';  
end;
```

- then use the logging system inside a method:

```
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);
```

*Intialize 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 SynLog: TSynLog;
```

*Retrieve the corresponding log file of this thread and family*

- creates the TSynLog if not already existing for this current thread

```
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 SetAutoFlush;

*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 - 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 writting to disk
- is set to 4096 by default (4 KB is the standard hard drive cluster size)

**property** DefaultExtension: TFileName read fDefaultExtension write fDefaultExtension;

*The file extension to be used*

- is '.log' by default

**property** DestinationPath: TFileName read fDestinationPath write fDestinationPath;

*The folder where the log must be stored*

- by default, is in the executable folder

**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

**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)

**property** Ident: integer read fIdent;

*Index in global SynLogFileFamily[] and threadvar SynLogFileIndex[] 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 sllError, sllException, sllExceptionOS, sllFail, sllLastError and sllStackTrace
- exceptions will always trace the stack

**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 proprietary 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 aOldLogFileName="" in order to close any pending archive (used e.g. by EventArchiveZip to open the .zip only once)

**property** PerThreadLog: boolean **read** fPerThreadLog **write** fPerThreadLog;

*If TRUE, each thread will have its own logging file*

**property** StackTraceLevel: byte **read** fStackTraceLevel **write** fStackTraceLevel;

*The recursive depth of stack trace symbol to write*

- used only if exceptions are handled, or by sllStackTrace level
- default value is 20, maximum is 255

**property** StackTraceUseOnlyAPI: boolean **read** fStackTraceUseOnlyAPI **write** fStackTraceUseOnlyAPI;

*If 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 this property to TRUE
- default is FALSE, i.e. use RtlCaptureStackBackTrace() API and perform a manual stack walk if the API returned to address

**property** SynLogClass: TSynLogClass **read** fSynLogClass;

*The associated TSynLog class*

**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

**TSynLogCurrentIdent = packed record**

*Used by ISynLog/TSynLog.Enter methods to handle recursivity calls tracing*

**Caller: PtrUInt;**

*The caller address, ready to display stack trace dump if needed*

**ClassType: TClass;**  
*Associated class type to be displayed*

**EnterTimeStamp: Int64;**  
*The time stamp at enter time*

**Instance: TObject;**  
*Associated class instance to be displayed*

**Method: PUTF8Char;**  
*The method name (or message) to be displayed*

**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=TRUE`) 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, `slStackTrace`, and `Enter/Leave` labelling

**constructor** `Create(aFamily: TSynLogFamily=nil);`

*Intialize for a `TSynLog` class instance*

- WARNING: not to be called directly! Use `Enter` or `Add` class function instead

**destructor** `Destroy; override;`

*Release all memory and internal handles*

**class function** `Add: TSynLog;`

*Retrieve the current instance of this `TSynLog` class*

- to be used for direct logging, without any `Enter/Leave`:

`TSynLogDB.Add.Log(11Error, 'The % statement didn't work', [SQL]);`

- to be used for direct logging, without any `Enter/Leave` (one parameter version - just the same as previous):

`TSynLogDB.Add.Log(11Error, 'The % statement didn't work', SQL);`

- is just a wrapper around `Family.SynLog` - the same code will work:

`TSynLogDB.Family.SynLog.Log(11Error, 'The % statement didn't work', [SQL]);`

```
class function Enter(aClassType: TClass; aMethodName: PUTF8Char=nil): ISynLog;
overload;
```

*To be called and assigned to a ISynLog interface at the beginning of a method*

- this is the main method to be called within a class method:

```
class function TMyDB.SQLValidate(const SQL: RawUTF8): boolean;
var ILog: ISynLog;
begin
  ILog := TSynLogDB.Enter(self, 'SQLValidate');
  // do some stuff
  ILog.Log(sllInfo, 'SQL=% returned %', [SQL, result]);
end;
```

- the use of a ISynLog interface will allow you to have an automated log entry created when the method is left

- 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

- if you just need to access the log inside the method block, you may not need any ISynLog interface:

```
class 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):

```
class 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

- 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:

```
20110325 19325801 +   MyDBUnit.TMyDB.SQLValidate
20110325 19325801 info   SQL=SELECT * FROM Table returned 1;
20110325 19325801 -   MyDBUnit.TMyDB.SQLValidate
```

```
class function Enter(aInstance: TObject=nil; aMethodName: PUTF8Char=nil):  
ISynLog; overload;
```

*Overloaded method which will Log the*

- this is the main method to be called within a method:

```
procedure TMyDB.SQLExecute(const SQL: RawUTF8);  
var ILog: ISynLog;  
begin  
  ILog := TSynLogDB.Enter(self, 'SQLExecute');  
  // do some stuff  
  ILog.Log(sllInfo, 'SQL=%', [SQL]);  
end;
```

- the use of a ISynLog interface will allow you to have an automated log entry created when the method is left

- 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

- if you just need to access the log inside the method block, you may not need any ISynLog interface:

```
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):

```
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

- 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:

```
20110325 19325801 +   MyDBUnit.TMyDB(004E11F4).SQLExecute  
20110325 19325801 info   SQL=SELECT * FROM Table;  
20110325 19325801 -   MyDBUnit.TMyDB(004E11F4).SQLExecute
```

```
class function Family: TSynLogFamily; overload;
```

*Retrieve the family of this TSynLog class type*

```
procedure Flush(ForceDiskWrite: boolean);
```

*Flush all Log content to file*

- if ForceDiskWrite is TRUE, will wait until written on disk (slow)

```
procedure Log(Level: TSynLogInfo; aName: PWinAnsiChar; aTypeInfo: pointer; var  
aValue; Instance: TObject=nil); 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 SQLite3Commons 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 Text: RawUTF8; aInstance: TObject=nil); 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:

000000000002DB9 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; 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 TStringList - 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 SQLite3Commons unit to add the record content, written as human readable JSON

**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** GenericFamily: TSynLogFamily read fFamily;

*The associated logging family*

**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 = '');

*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*

**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

**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

**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** 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

**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** 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** 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** LogProc: PSynLogFileProcArray read fLogProcCurrent;

*Profiled methods information*

- is calculated by Create() constructor
- will contain the slEnter 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

**property** LogProcMerged: boolean read fLogProcIsMerged write SetLogProcMerged;

*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*

**property** RunningUser: RawUTF8 read fUser;

*The computer user name who launched the process*

**property** ServicePack: integer read fOSServicePack;

*The Operating System Service Pack number*

**property** StartDateTime: TDateTime read fStartDateTime;

*The date and time at which the log file was started*

```
property Wow64: boolean read fWow64;
```

*If the 32 bit process was running under WOW 64 virtual emulation*

#### **Types implemented in the SynCommons unit:**

```
PIso8601 = ^Iso8601;
```

*Pointer to Iso8601*

```
PPtrInt = ^PtrInt;
```

*A CPU-dependent signed integer type cast of a pointer of pointer*

- used for 64 bits compatibility, native under Free Pascal Compiler

```
PPtrUInt = ^PtrUInt;
```

*A CPU-dependent unsigned integer type cast of a pointer of pointer*

- used for 64 bits compatibility, native under Free Pascal Compiler

```
PRawByteString = ^RawByteString;
```

*Pointer to a RawByteString*

```
PtrInt = integer;
```

*A CPU-dependent signed integer type cast of a pointer / register*

- used for 64 bits compatibility, native under Free Pascal Compiler

```
PtrUInt = cardinal;
```

*A CPU-dependent unsigned integer type cast of a pointer / register*

- used for 64 bits compatibility, native under Free Pascal Compiler

```
PUTF8Char = type PAnsiChar;
```

*A simple wrapper to UTF-8 encoded zero-terminated PAnsiChar*

- PAnsiChar is used only for Win-Ansi encoded text

- the Synopse SQLite3 framework uses mostly this PUTF8Char type, because all data is internally stored and expected to be UTF-8 encoded

```
QWord = Int64;
```

*Unsigned Int64 doesn't exist under older Delphi, but is defined in FPC*

```
RawByteString = 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

```
RawUnicode = type AnsiString;
```

*RawUnicode is an Unicode String stored in an AnsiString*

- faster than WideString, which are allocated in Global heap (for COM)

- an AnsiChar(#0) is added at the end, for having a true WideChar(#0) at ending

- 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)

- pointer(RawUnicode) is compatible with Win32 'Wide' API call

- mimic Delphi 2009 UnicodeString, without the WideString or Ansi conversion overhead

- all conversion to/from AnsiString or RawUTF8 must be explicit



**RawUTF8 = type AnsiString;**

*RawUTF8 is an UTF-8 String stored in an AnsiString*

- 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
- mimic Delphi 2009 UTF8String, without the charset conversion overhead
- all conversion to/from AnsiString or RawUnicode must be explicit

**SynUnicode = WideString;**

*SynUnicode is the fastest available Unicode native string type, depending on the compiler used*

- 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)
- 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)
- starting with Delphi 2009, it uses fastest UnicodeString type, which allow Copy On Write, Reference Counting and fast heap memory allocation

**TCompareOperator =**

**( soEqualTo, soNotEqualTo, soLessThan, soLessThanOrEqualTo, soGreaterThan, soGreaterThanOrEqualTo, soBeginWith, soContains, soSoundsLikeEnglish, soSoundsLikeFrench, soSoundsLikeSpanish );**

*SQL Query comparison operators*

- these operators are e.g. used by CompareOperator() functions

**TDateTimeDynArray = array of TDateTime;**

*A dynamic array of TDateTime values*

**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): 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, djByte, djWord, djInteger, djCardinal, djInt64, djDouble, djCurrency,  
 djTimeLog, djDateTime, djRawUTF8, djWinAnsi, djString, djWideString, djSynUnicode,  
 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)
- djByte .. djTimeLog match numerical JSON values
- djDateTime .. djSynUnicode match textual JSON values
- 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

**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. SortDynArrayByte, SortDynArrayWord, SortDynArrayInteger, SortDynArrayCardinal, SortDynArrayInt64, SortDynArrayDouble, SortDynArrayAnsiString, SortDynArrayAnsiStringI, SortDynArrayUnicodeString, SortDynArrayUnicodeStringI, SortDynArrayString, SortDynArrayStringI
- 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

**TEventDynArraySortCompare = function(const A,B): integer of object;**

*Event oriented version of TDynArraySortCompare*

**TFileBufferWriterKind =**  
 ( wkUInt32, wkVarUInt32, wkVarInt32, wkSorted, wkOffsetU, wkOffsetI );

*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=-1,3=2,4=-2...)
- wkSorted will write an increasing array of integers, handling the special case of a difference of 1 between two values (therefore is very optimized to store an array of IDs)
- wkOffsetU and wkOffsetI will write the difference between two successive values, handling constant difference (Unsigned or Integer) in an optimized manner

```

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
  - a good candidate is our crc32() function in optimized asm in SynZip unit
  - TDynArrayHashed.Init will use kr32() if no custom function is supplied, which is the standard
    Kernighan & Ritchie hash function

TLogProcSortOrder = ( soNone, soByName, soByOccurrence, soByTime, soByProperTime );
  Used by TSynLogFile.LogProcSort method

TObjectListPropertyHashedAccessProp = function(aObject: TObject): pointer;
  Function prototype used to retrieve the hashed property of a TObjectListPropertyHashed list

TOnDynArrayHashOne = function(const Elem): cardinal of object;
  Event handler to be used for hashing of a dynamic array element

TPAnsiCharArray = array[0..MaxInt div SizeOf(PAnsiChar)-1] of PAnsiChar;
  A pointer to a PAnsiChar array

TPtrUIntArray = array[0..MaxInt div SizeOf(PtrUInt)-1] of PtrUInt;
  A pointer to a PtrUInt array

TPtrUIntDynArray = array of PtrUInt;
  A dynamic array of PtrUInt values

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

TRawByteStringDynArray = array of RawByteString;
  A dynamic array of RawByteString

TRawUTF8Array = array[0..MaxInt div SizeOf(RawUTF8)-1] of RawUTF8;
  A pointer to a RawUTF8 array

TRawUTF8DynArray = array of RawUTF8;
  A dynamic array of UTF-8 encoded strings

TSBFString = type RawByteString;
  An custom RawByteString type used to store internaly a data in our SBF compact binary format

TSQLFieldBits = set of 0..MAX_SQLFIELDS-1;
  Used to store bit set for all available fiels in a Table
  - with current MAX_SQLFIELDS value, 256 bits uses 64 bytes of memory

TStringDynArray = array of string;
  A dynamic array of generic VCL strings

TSynHashDynArray = array of TSynHash;
  Internal structure used to store hashes of items
  - used e.g. by TDynArrayHashed or TObjectHash

TSynLogArchiveEvent = function(const aOldLogFileName, aDestinationPath: TFileName):

```

**boolean;**

*This event can be set for TSynLogFamily to archive any deprecated log into a custom compressed format*

- will be called by TSynLogFamily when TSynLogFamily.Destroy identify some outdated files
- the aOldLogFileName 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 aOldLogFileName="" in order to close any pending archive (used e.g. by EventArchiveZip to open the .zip only once)

**TSynLogCurrentIdents = array[0..maxInt div sizeof(TSynLogCurrentIdent)-1] of TSynLogCurrentIdent;**

*Used to store the identification of all recursivity levels*

**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

**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 );**

*The available logging events, as handled by TSynLog*

- 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'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

**TSynLogInfoDynArray = array of TSynLogInfo;**

*A dynamic array of logging event levels*

**TSynLogInfos = set of TSynLogInfo;**

*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

**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*

**TSynNameValueItemDynArray = array of TSynNameValueItem;**

*Name/Value pairs storage, as used by TSynNameValue class*

**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: shortstring): 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

**TSynTestEvent = procedure of object;**

*The prototype of an individual test*

- to be used with TSynTest descendants

**TSynUnicodeDynArray = array of SynUnicode;**

*A dynamic array of SynUnicode values*

**TTextWriterKind = ( twNone, twJSONEscape, twOnSameLine );**

*Kind of adding in a TTextWriter*

**TTimeLog = type Int64;**

*Fast integer-encoded date and time value*

- faster than Iso-8601 text and TDateTime
- e.g. can be used as published property field in TSQLRecord
- convenient for current date and time process (logging e.g.)
- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds - same as Iso8601ToSeconds()
- type cast any value of TTimeLog with the Iso8601 object below for easy access to its content
- since TTimeLog type is bit-oriented, you can't just use add or subtract two TTimeLog values when doing such date/time computation: use a TDateTime temporary conversion in such case

**TTimeLogDynArray = array of TTimeLog;**

*Dynamic array of TTimeLog*

- used by TDynArray JSON serialization to handle textual serialization

**TUTF8Compare = function(P1,P2: PUTF8Char): PtrInt;**

*Function prototype used internally for UTF-8 buffer comparaison*

- used in SQLite3commons during TSQLTable rows sort and by TSQLQuery

**TVarDataDynArray = array of TVarData;**

*A dynamic array of TVarData values*

**TwideStringDynArray = array of WideString;**

*A dynamic array of WideString values*

**TWinAnsiDynArray = array of WinAnsiString;**

*A dynamic array of WinAnsi encoded strings*

**TWindowsVersion =**

( wUnknown, w2000, wXP, wXP\_64, wServer2003, wServer2003\_R2, wVista, wVista\_64, wServer2008, wServer2008\_64, wServer2008\_R2, wServer2008\_R2\_64, wSeven, wSeven\_64, wEight, wEight\_64, wEightServer, wEightServer\_64 );

*The recognized Windows versions*



**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:**

**CODEPAGE\_US = 1252;**

*US English Windows Code Page, i.e. WinAnsi standard character encoding*

**CP\_UTF16 = 1200;**

*Internal Code Page for 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*

**DOUBLE\_PRECISION = 15;**

*Best possible precision when rendering a "double" kind of float*

- can be used as parameter for ExtendedToString/ExtendedToStr

**HEADER\_CONTENT\_TYPE = 'Content-Type: ';**

*HTTP header, as defined in the corresponding RFC*

**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*

**IsIdentifier: set of byte =**

**[ord('\_'),ord('0')..ord('9'),ord('a')..ord('z'),ord('A')..ord('Z')];**

*Used internally for fast identifier recognition (32 bytes const)*

- can be used e.g. for field or table name
- this char set matches the classical pascal definition of identifiers

**IsWord: set of byte = [ord('0')..ord('9'),ord('a')..ord('z'),ord('A')..ord('Z')];**

*Used internally for fast word recognition (32 bytes const)*

**JSON\_BASE64\_MAGIC = \$b0bfef;**

*UTF-8 encoded \uFFFF0 special code -> mark Base64 TDynArray.SaveTo in JSON*

- Unicode special char U+FFFF0 is UTF-8 encoded as EF BF B0 bytes

**JSON\_BASE64\_MAGIC\_QUOTE = ord('"')+cardinal(JSON\_BASE64\_MAGIC) shl 8;**

*"" + UTF-8 encoded \uFFFF0 special code*

**JSON\_BASE64\_MAGIC\_QUOTE\_VAR: cardinal = JSON\_BASE64\_MAGIC\_QUOTE;**

*"" + UTF-8 encoded \uFFFF0 special code*

**JSON\_BOOLEAN: array[boolean] of RawUTF8 = ('false','true');**

*JSON compatible representation of a boolean value*

**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 830).*

**JSON\_SQLDATE\_MAGIC = \$b1bfef;**

*UTF-8 encoded \uFFF1 special code -> mark ISO-8601 SQLDATE in JSON*

- e.g. ""\uFFF12012-05-04"" pattern
- Unicode special char U+FFF1 is UTF-8 encoded as EF BF B0 bytes
- as generated by DateToSQL/DateTimeToSQL functions, and expected by SQLParamContent and ExtractInlineParameters functions

**JSON\_SQLDATE\_MAGIC\_QUOTE = ord('"' + UTF-8 encoded \uFFF1 special code**

- as generated by DateToSQL/DateTimeToSQL functions, and expected by SQLParamContent and ExtractInlineParameters functions

**LOG\_LEVEL\_TEXT: array[TSynLogInfo] of string[7] = ( ' ', ' info ', ' debug ', ' trace ', ' warn ', ' ERROR ', ' + ', ' - ', ' OSERR ', ' EXC ', ' EXCOS ', ' mem ', ' stack ', ' fail ', ' SQL ', ' cache ', ' res ', ' DB ', ' http ', ' clnt ', ' srvr ', ' call ', ' ret ', ' auth ', ' cust1 ', ' cust2 ', ' cust3 ', ' cust4 ');**

*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 = [sllError, sllException, sllExceptionOS];**

*Contains the logging levels for which stack trace should be dumped*

- i.e. when such a log event occur, and the available recursive stack has not been traced yet (if sllEnter if not in the current selected levels)

**LOG\_VERBOSE: TSynLogInfos = [succ(sllNone)..high(TSynLogInfo)];**

*Can be set to TSynLogFamily.Level in order to log all available events*

**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

**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)
- default is 64, but can be set to any value (64, 128, 192 and 256 optimized)
- this constant is used internally to optimize memory usage in the generated asm code, and statically allocate some arrays for better speed

**MAX\_SYNLOGFAMILY = 15;**



*Up to 16 TSynLogFamily, i.e. TSynLog children classes can be defined*

**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)

**SYNOPSIS\_FRAMEWORK\_VERSION = '1.17';**

*The corresponding version of the freeware Synopse framework*

**SYNTABLESTATEMENTWHEREALL = -1;**

*Used by TSynTableStatement.WhereField for "SELECT \* FROM TableName"*

**SYNTABLESTATEMENTWHERECOUNT = -2;**

*Used by TSynTableStatement.WhereField for "SELECT Count(\*) FROM TableName"*

**SYNTABLESTATEMENTWHEREID = 0;**

*Used by TSynTableStatement.WhereField for "SELECT .. FROM TableName WHERE ID=?"*

**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*

```
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',
'30','31','32','33','34','35','36','37','38','39',
'40','41','42','43','44','45','46','47','48','49',
'50','51','52','53','54','55','56','57','58','59',
'60','61','62','63','64','65','66','67','68','69',
'70','71','72','73','74','75','76','77','78','79',
'80','81','82','83','84','85','86','87','88','89',
'90','91','92','93','94','95','96','97','98','99');
```

*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

**varInt64 = \$0014;**

*Delphi 5 doesn't have those base types defined :(*

## Functions or procedures implemented in the SynCommons unit:

Functions or procedures	Description	Page
AddInteger	Add an integer value at the end of a dynamic array of integers	326
AddInteger	Add an integer value at the end of a dynamic array of integers	326
AddPrefixToCSV	Append some prefix to all CSV values	326
AddRawUTF8	True if Value was added successfully in Values[]	326

Functions or procedures	Description	Page
AddSortedInteger	Add an integer value in a sorted dynamic array of integers	327
AddSortedRawUTF8	Add a RawUTF8 value in an alphabetically sorted dynamic array of RawUTF8	327
Ansi7ToString	Convert any Ansi 7 bit encoded String into a generic VCL Text	327
Ansi7ToString	Convert any Ansi 7 bit encoded String into a generic VCL Text	327
AnsiCharToUTF8	Convert an AnsiChar buffer (of a given code page) into a UTF-8 string	327
AnsiIComp	Fast WinAnsi comparison using the NormToUpper[] array for all 8 bits values	327
AnsiICompW	Fast case-insensitive Unicode comparison	327
AppendBufferToRawUTF8	Fast add some characters to a RawUTF8 string	327
AppendCSVValues	Append some text lines with the supplied Values[]	328
AppendRawUTF8ToBuffer	Fast add some characters from a RawUTF8 string into a given buffer	328
AppendToTextFile	Log a message to a local text file	328
Base64Decode	Direct decoding of a Base64 encoded buffer	328
Base64ToBin	Fast conversion from Base64 encoded text into binary data	328
Base64ToBin	Fast conversion from Base64 encoded text into binary data	328
Base64ToBinLength	Retrieve the expected length of a Base64 encoded buffer	328
BinToBase64	Fast conversion from binary data into Base64 encoded text	328
BinToBase64	Fast conversion from binary data into Base64 encoded text	328
BinToBase64URI	Fast conversion from binary data into Base64-like URI-compatible encoded text	328
BinToBase64WithMagic	Fast conversion from binary data into Base64 encoded text with JSON_BASE64_MAGIC prefix (UTF-8 encoded \uFFFF0 special code)	328
BinToBase64WithMagic	Fast conversion from binary data into Base64 encoded text with JSON_BASE64_MAGIC prefix (UTF-8 encoded \uFFFF0 special code)	328
BinToHex	Fast conversion from binary data into hexa chars	329
BinToHex	Fast conversion from binary data into hexa chars	329
BinToHexDisplay	Fast conversion from binary data into hexa chars, ready to be displayed	329
CardinalToHex	Fast conversion from a Cardinal data into hexa chars, ready to be displayed	329
CharSetToCodePage	Convert a char set to a code page	329

Functions or procedures	Description	Page
CodePageToCharSet	Convert a code page to a char set	329
CompareMem	Use our fast asm version of CompareMem()	329
CompareOperator	Low-level integer comparison according to a specified operator	330
CompareOperator	Low-level floating-point comparison according to a specified operator	330
CompareOperator	Low-level text comparison according to a specified operator	330
ContainsUTF8	Return true if up^ is contained inside the UTF-8 buffer p^	330
ConvertCaseUTF8	Fast conversion of the supplied text into 8 bit case sensitivity	330
CopyAndSortInteger	Copy an integer array, then sort it, low values first	330
CreateInternalWindow	This function can be used to create a GDI compatible window, able to receive GDI messages for fast local communication	330
CSVOfValue	Return a CSV list of the iterated same value	330
CSVToIntegerDynArray	Add the strings in the specified CSV text into a dynamic array of integer	330
CSVToRawUTF8DynArray	Add the strings in the specified CSV text into a dynamic array of UTF-8 strings	330
Curr64ToPChar	Convert an INTEGER Curr64 (value*10000) into a string	331
Curr64ToStr	Convert an INTEGER Curr64 (value*10000) into a string	331
Curr64ToString	Convert a currency value from its Int64 binary representation into its numerical text equivalency	331
DateTimeToIso8601	Basic Date/Time conversion into ISO-8601	331
DateTimeToIso8601Text	Write a TDateTime into strict ISO-8601 date and/or time text	331
DateTimeToSQL	Convert a date/time to a ISO-8601 string format for SQL '?' inlined parameters	331
DateToIso8601	Basic Date conversion into ISO-8601	331
DateToIso8601	Basic Date conversion into ISO-8601	331
DateToIso8601PChar	Write a Date to P^ Ansi buffer	332
DateToIso8601PChar	Write a Date to P^ Ansi buffer	332
DateToIso8601Text	Convert a date into 'YYYY-MM-DD' date format	332
DateToSQL	Convert a date to a ISO-8601 string format for SQL '?' inlined parameters	332
DeleteInteger	Delete any integer in Values[]	332

Functions or procedures	Description	Page
DeleteInteger	Delete any integer in Values[]	332
DeleteRawUTF8	Delete a RawUTF8 item in a dynamic array of RawUTF8	332
DeleteSection	Delete a whole [Section]	333
DeleteSection	Delete a whole [Section]	333
DirectoryExists	DirectoryExists returns a boolean value that indicates whether the specified directory exists (and is actually a directory)	333
DoubleToStr	Convert a floating-point value to its numerical text equivalency	333
DoubleToString	Convert a floating-point value to its numerical text equivalency	333
DynArray	Initialize the structure with a one-dimension dynamic array	333
EventArchiveDelete	A TSynLogArchiveEvent handler which will delete older .log files	333
EventArchiveSynLZ	A TSynLogArchiveEvent handler which will compress older .log files using our proprietary SynLZ format	333
ExeVersionRetrieve	Initialize ExeVersion global variable, if not already done	334
ExistsIniName	Return TRUE if Value of UpperName does exist in P, till end of current section	334
ExistsIniNameValue	Return TRUE if the Value of UpperName exists in P, till end of current section	334
ExtendedToStr	Convert a floating-point value to its numerical text equivalency	334
ExtendedToString	Convert a floating-point value to its numerical text equivalency	334
FastFindIntegerSorted	Fast binary search of an integer value in a sorted integer array	334
FastFindPUTF8CharSorted	Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array	334
FastFindPUTF8CharSorted	Retrieve the index where is located a PUTF8Char in a sorted PUTF8Char array	334
FastLocateIntegerSorted	Retrieve the index where to insert an integer value in a sorted integer array	334
FastLocatePUTF8CharSorted	Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array	335
FastLocatePUTF8CharSorted	Retrieve the index where to insert a PUTF8Char in a sorted PUTF8Char array	335
FieldNameValid	Returns TRUE if the given text buffer contains A..Z,0..9 characters	335
FileAgeToDateTime	Get the file date and time	335

Functions or procedures	Description	Page
FileFromString	Create a File from a string content	335
FileSeek64	FileSeek() overloaded function, working with huge files	335
FileSize	Get the file size	335
FileSynLZ	Compress a file content using the SynLZ algorithm a file content	335
FileUnSynLZ	Compress a file content using the SynLZ algorithm a file content	335
FillChar	Faster implementation of Move() for Delphi versions with no FastCode inside	335
FillIncreasing	Fill some values with i,i+1,i+2...i+Count-1	335
FindAnsi	Return true if UpperValue (Ansi) is contained in A^ (Ansi)	335
FindCSVIndex	Return the index of a Value in a CSV string	336
FindIniEntry	Find a Name= Value in a [Section] of a INI RawUTF8 Content	336
FindIniEntryFile	Find a Name= Value in a [Section] of a .INI file	336
FindIniEntryInteger	Find a Name= numeric Value in a [Section] of a INI RawUTF8 Content and return it as an integer, or 0 if not found	336
FindIniNameValue	Find the Value of UpperName in P, till end of current section	336
FindIniNameValueInteger	Find the integer Value of UpperName in P, till end of current section	336
FindNextUTF8WordBegin	Points to the beginning of the next word stored in U	336
FindObjectEntry	Retrieve a property value in a text-encoded class	336
FindObjectEntryWithoutExt	Retrieve a filename property value in a text-encoded class	336
FindRawUTF8	Return the index of Value in Values[], -1 if not found	337
FindSectionFirstLine	Find the position of the [SEARCH] section in source	337
FindSectionFirstLineW	Find the position of the [SEARCH] section in source	337
FindUnicode	Return true if Uppe (Unicode encoded) is contained in U^ (UTF-8 encoded)	337
FindUTF8	Return true if UpperValue (Ansi) is contained in U^ (UTF-8 encoded)	337
FindWinAnsiIniEntry	Find a Name= Value in a [Section] of a INI WinAnsi Content	337
FindWinAnsiIniNameValue	Find the Value of UpperName in P, till end of current section	337
FormatUTF8	Fast Format() function replacement, handling % and ? parameters	338

Functions or procedures	Description	Page
FormatUTF8	Fast Format() function replacement, optimized for RawUTF8	338
FromVarInt32	Convert a 32-bit variable-length integer buffer into an integer	338
FromVarInt64	Convert a 64-bit variable-length integer buffer into a Int64	338
FromVarString	Retrieve a variable-length text buffer	338
FromVarUInt32	Convert a 32-bit variable-length integer buffer into a cardinal	338
FromVarUInt32High	Convert a 32-bit variable-length integer buffer into a cardinal	338
FromVarUInt32Up128	Convert a 32-bit variable-length integer buffer into a cardinal	338
FromVarUInt64	Convert a 64-bit variable-length integer buffer into a UInt64	338
GetBit	Retrieve a particular bit status from a bit array	338
GetBit64	Retrieve a particular bit status from a Int64 bit array (max aIndex is 63)	338
GetBitCSV	Convert a set of bit into a CSV content	338
GetBitsCount	Compute the number of bits set in a bit array	339
GetCaptionFromClass	UnCamelCase and translate the class name, trimming any left 'T', 'TSyn', 'TSQL' or 'TSQLRecord'	339
GetCaptionFromEnum	UnCamelCase and translate the enumeration item	339
GetCaptionFromPCharLen	UnCamelCase and translate a char buffer	339
GetCardinal	Get the unsigned 32 bits integer value stored in P^	339
GetCardinalDef	Get the unsigned 32 bits integer value stored in P^	339
GetCardinalW	Get the unsigned 32 bits integer value stored as Unicode string in P^	339
GetCSVItem	Return n-th indexed CSV string in P, starting at Index=0 for first one	339
GetCSVItemString	Return n-th indexed CSV string in P, starting at Index=0 for first one	339
GetDelphiCompilerVersion	Return the Delphi Compiler Version	339
GetDisplayNameFromClass	Will get a class name as UTF-8	339
GetEnumName	Helper to retrieve the text of an enumerate item	339
GetExtended	Get the extended floating point value stored in P^	340
GetExtended	Get the extended floating point value stored in P^	340
GetFileNameExtIndex	Extract a file extension from a file name, then compare with a comma separated list of extensions	340

Functions or procedures	Description	Page
GetFileNameWithoutExt	Extract file name, without its extension	340
GetFileVersion	GetFileVersion returns the most significant 32 bits of a file's binary version number	340
GetInt64	Get the 64 bits integer value stored in P^	340
GetInt64	Get the 64 bits integer value stored in P^	340
GetInteger	Get the signed 32 bits integer value stored in P^	340
GetInteger	Get the signed 32 bits integer value stored in P^	340
GetJSONField	Decode a JSON field in an UTF-8 encoded buffer (used in TSQLTableJSON.Create)	341
GetLineSize	Compute the line length from source array of chars	341
GetLineSizeSmallerThan	Returns true if the line length from source array of chars is not less than the specified count	341
GetMimeType	Retrieve the MIME content type from a supplied binary buffer	341
GetModuleName	Retrieve the full path name of the given execution module (e.g. library)	341
GetNextItem	Return next CSV string from P, nil if no more	341
GetNextItemCardinal	Return next CSV string as unsigned integer from P, 0 if no more	341
GetNextItemCardinalW	Return next CSV string as unsigned integer from P, 0 if no more	341
GetNextItemDouble	Return next CSV string as double from P, 0.0 if no more	341
GetNextItemShortString	Return next CSV string from P, nil if no more	341
GetNextItemString	Return next CSV string from P, nil if no more	342
GetNextLine	Extract a line from source array of chars	342
GetNextLineBegin	Return line begin from source array of chars, and go to next line	342
GetNextStringLineToRawUnicode	Return next string delimited with #13#10 from P, nil if no more	342
GetNextUTF8Upper	Retrieve the next UCS2 value stored in U, then update the U pointer	342
GetSectionContent	Retrieve the whole content of a section as a string	342
GetSectionContent	Retrieve the whole content of a section as a string	342
GetUTF8Char	Get the WideChar stored in P^ (decode UTF-8 if necessary)	342
GotoEndOfQuotedString	Get the next character after a quoted buffer	342
GotoNextJSONField	Reach the position of the next JSON field in the supplied UTF-8 buffer	342

Functions or procedures	Description	Page
GotoNextJSONObjectOrArray	Reach the position of the next JSON object of JSON array	342
GotoNextNotSpace	Get the next character not in [#1..' ']	342
GotoNextVarInt	Jump a value in the 32-bit or 64-bit variable-length integer buffer	342
GotoNextVarString	Jump a value in variable-length text buffer	343
Hash32	Our custom hash function, specialized for Text comparaison	343
Hash32	Our custom hash function, specialized for Text comparaison	343
HexDisplayToBin	Fast conversion from hexa chars into a pointer	343
HexDisplayToCardinal	Fast conversion from hexa chars into a cardinal	343
HexToBin	Fast conversion from hexa chars into binary data	343
IdemFileExt	Returns true if the file name extension contained in p^ is the same same as extup^	343
IdemPChar	Returns true if the beginning of p^ is the same as up^	343
IdemPCharAndGetNextLine	Return true if IdemPChar(source,search), and go to the next line of source	343
IdemPCharArray	Returns the index of a matching beginning of p^ in upArray[]	344
IdemPCharU	Returns true if the beginning of p^ is the same as up^	344
IdemPCharW	Returns true if the beginning of p^ is same as up^	344
IdemPropName	Case unsensitive test of P1 and P2 content	344
IdemPropName	Case unsensitive test of P1 and P2 content	344
IdemPropNameU	Case unsensitive test of P1 and P2 content	344
InsertInteger	Insert an integer value at the specified index position of a dynamic array of integers	344
Int32ToUtf8	Use our fast RawUTF8 version of IntToStr()	344
Int64ToUInt32	Copy some Int64 values into an unsigned integer array	344
Int64ToUtf8	Use our fast RawUTF8 version of IntToStr()	345
IntegerDynArrayLoadFrom	Wrap an Integer dynamic array BLOB content as stored by TDynArray.SaveTo	345
IntegerDynArrayToCSV	Return the corresponding CSV text from a dynamic array of integer	345
IntegerScan	Fast search of an unsigned integer position in an integer array	345
IntegerScanExists	Fast search of an unsigned integer position in an integer array	345



Functions or procedures	Description	Page
IntegerScanIndex	Fast search of an unsigned integer position in an integer array	345
IntToString	Faster version than default SysUtils.IntToStr implementation	345
IntToString	Faster version than default SysUtils.IntToStr implementation	345
IntToString	Faster version than default SysUtils.IntToStr implementation	345
IntToThousandString	Convert an integer value into its textual representation with thousands marked	345
IsAnsiCompatible	Return TRUE if the supplied buffer only contains 7-bits Ansi characters	346
IsAnsiCompatible	Return TRUE if the supplied text only contains 7-bits Ansi characters	346
IsAnsiCompatible	Return TRUE if the supplied buffer only contains 7-bits Ansi characters	346
IsAnsiCompatible	Return TRUE if the supplied buffer only contains 7-bits Ansi characters	346
IsAnsiCompatible	Return TRUE if the supplied buffer only contains 7-bits Ansi characters	346
IsBase64	Check if the supplied text is a valid Base64 encoded stream	346
IsBase64	Check if the supplied text is a valid Base64 encoded stream	346
IsContentCompressed	Retrieve if some content is compressed, from a supplied binary buffer	346
IsIso8601	Test if P^ contains a valid ISO-8601 text encoded value	346
IsMatch	Return TRUE if the supplied content matchs to a grep-like pattern	346
Iso8601FromDateTime	Get TTimeLog value from a given Delphi date and time	346
Iso8601FromFile	Get TTimeLog value from a file date and time	346
Iso8601Now	Get TTimeLog value from current date and time	346
Iso8601ToDateTime	Date/Time conversion from ISO-8601	347
Iso8601ToDateTimePUTF8Char	Date/Time conversion from ISO-8601	347
Iso8601ToDateTimePUTF8CharVar	Date/Time conversion from ISO-8601	347
Iso8601ToSeconds	Convert a Iso8601 encoded string into a "fake" second count	347
Iso8601ToSecondsPUTF8Char	Convert a Iso8601 encoded string into a "fake" second count	347
Iso8601ToSQL	Convert an Iso8601 date/time (bit-encoded as Int64) to a ISO-8601 string format for SQL '?' inlined parameters	347
IsRowID	Returns TRUE if the specified field name is either 'ID', either 'ROWID'	347
IsRowID	Returns TRUE if the specified field name is either 'ID', either 'ROWID'	347

Functions or procedures	Description	Page
isSelect	Return true if the parameter is void or begin with a 'SELECT' SQL statement	348
IsString	Test if the supplied buffer is a "string" value or a numerical value (floating point or integer), according to the characters within	348
IsStringJSON	Test if the supplied buffer is a "string" value or a numerical value (floating or integer), according to the JSON encoding schema	348
IsValidEmail	Return TRUE if the supplied content is a valid email address	348
IsValidIP4Address	Return TRUE if the supplied content is a valid IP v4 address	348
IsWinAnsi	Return TRUE if the supplied unicode buffer only contains WinAnsi characters	348
IsWinAnsi	Return TRUE if the supplied unicode buffer only contains WinAnsi characters	348
IsWinAnsiU	Return TRUE if the supplied UTF-8 buffer only contains WinAnsi characters	348
IsWinAnsiU8Bit	Return TRUE if the supplied UTF-8 buffer only contains WinAnsi 8 bit characters	348
IsZero	Returns TRUE if no bit inside this TSQLFieldBits is set	348
IsZero	Returns TRUE if all bytes equal zero	348
JSONDecode	Decode the supplied UTF-8 JSON content for the one supplied name	349
JSONDecode	Decode the supplied UTF-8 JSON content for the supplied names	349
JSONDecode	Decode the supplied UTF-8 JSON content for the supplied names	349
JSONEncode	Encode the supplied data as an UTF-8 valid JSON object content	349
JSONEncodeArray	Encode the supplied RawUTF8 array data as an UTF-8 valid JSON array content	350
JSONEncodeArray	Encode the supplied integer array data as an UTF-8 valid JSON array content	350
JSONEncodeArray	Encode the supplied floating-point array data as an UTF-8 valid JSON array content	350
KB	Convert a size to a human readable value	350
kr32	Standard Kernighan & Ritchie hash from "The C programming Language", 3rd edition	350
LogToTextFile	Log a message to a local text file	350
LowerCase	Fast conversion of the supplied text into lowercase	350

Functions or procedures	Description	Page
LowerCaseU	Fast conversion of the supplied text into 8 bit lowercase	350
LowerCaseUnicode	Accurate conversion of the supplied UTF-8 content into the corresponding lower-case Unicode characters	350
MicroSecToString	Convert a micro seconds elapsed time into a human readable value	350
Move	Faster implementation of Move() for Delphi versions with no FastCode inside	350
NextUTF8Char	Get the WideChar stored in P^ (decode UTF-8 if necessary) and set new pos to Next	351
NowToString	Retrieve the current Date, in the ISO 8601 layout, but expanded and ready to be displayed	351
PatchCode	Self-modifying code - change some memory buffer in the code segment	351
PatchCodePtrUInt	Self-modifying code - change one PtrUInt in the code segment	351
PointerToHex	Fast conversion from a pointer data into hexa chars, ready to be displayed	351
PosChar	Fast retrieve the position of a given character	351
PosEx	Faster RawUTF8 Equivalent of standard StrUtils.PosEx	351
PosI	A non case-sensitive RawUTF8 version of Pos()	351
PosIU	A non case-sensitive RawUTF8 version of Pos()	351
QuickSortInteger	Sort an Integer array, low values first	351
QuickSortRawUTF8	Sort a dynamic array of RawUTF8 items	351
QuotedStr	Format a buffered text content with quotes	352
QuotedStr	Format a text content with quotes	352
RawByteArrayConcat	Fast concatenation of several AnsiStrings	352
RawUnicodeToString	Convert any Raw Unicode encoded buffer into a generic VCL Text	352
RawUnicodeToString	Convert any Raw Unicode encoded buffer into a generic VCL Text	352
RawUnicodeToString	Convert any Raw Unicode encoded string into a generic VCL Text	352
RawUnicodeToSynUnicode	Convert any Raw Unicode encoded String into a generic SynUnicode Text	352
RawUnicodeToSynUnicode	Convert any Raw Unicode encoded String into a generic SynUnicode Text	352
RawUnicodeToUtf8	Convert a RawUnicode PWideChar into a UTF-8 string	352

Functions or procedures	Description	Page
RawUnicodeToUtf8	Convert a RawUnicode PWideChar into a UTF-8 string	352
RawUnicodeToUtf8	Convert a RawUnicode PWideChar into a UTF-8 buffer	352
RawUnicodeToUtf8	Convert a RawUnicode string into a UTF-8 string	352
RawUnicodeToWinAnsi	Convert a RawUnicode string into a WinAnsi (code page 1252) string	352
RawUnicodeToWinAnsi	Convert a RawUnicode PWideChar into a WinAnsi (code page 1252) string	352
RawUnicodeToWinPChar	Direct conversion of a Unicode encoded buffer into a WinAnsi PAnsiChar buffer	353
RawUTF8ArrayToCSV	Return the corresponding CSV text from a dynamic array of UTF-8 strings	353
RawUTF8ArrayToQuotedCSV	Return the corresponding CSV quoted text from a dynamic array of UTF-8 strings	353
RawUTF8DynArrayEquals	True if both TRawUTF8DynArray are the same	353
RawUTF8DynArrayLoadFromContains	Search in a RawUTF8 dynamic array BLOB content as stored by TDynArray.SaveTo	353
ReadStringFromStream	Read an UTF-8 text from a TStream	353
RecordClear	Clear a record content	353
RecordCopy	Copy a record content from source to Dest	353
RecordEquals	Check equality of two records by content	353
RecordLoad	Fill a record content from a memory buffer as saved by RecordSave()	353
RecordLoadJSON	Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON	353
RecordSave	Save a record content into a RawByteString	354
RecordSave	Save a record content into a destination memory buffer	354
RecordSaveLength	Compute the number of bytes needed to save a record content using the RecordSave() function	354
RedirectCode	Self-modifying code - add an asm JUMP to a redirected function	354
RedirectCodeRestore	Self-modifying code - restore a code from its RedirectCode() backup	354
ReleaseInternalWindow	Delete the window resources used to receive GDI messages	354
ReplaceSection	Replace a whole [Section] content by a new content	354
ReplaceSection	Replace a whole [Section] content by a new content	354

Functions or procedures	Description	Page
RoundTo2Digits	Truncate a Currency value to only 2 digits	355
SameTextU	SameText() overloaded function with proper UTF-8 decoding	355
SameValue	Compare to floating point values, with IEEE 754 double precision	355
SetBit	Set a particular bit into a bit array	355
SetBit64	Set a particular bit into a Int64 bit array (max aIndex is 63)	355
SetBitCSV	Retrieve the next CSV separated bit index	355
SetInt64	Get the 64 bits integer value stored in P^	355
SimpleDynArrayLoadFrom	Wrap a simple dynamic array BLOB content as stored by TDynArray.SaveTo	355
SortDynArrayAnsiString	Compare two "array of AnsiString" elements, with case sensitivity	355
SortDynArrayAnsiStringI	Compare two "array of AnsiString" elements, with no case sensitivity	355
SortDynArrayByte	Compare two "array of byte" elements	355
SortDynArrayCardinal	Compare two "array of cardinal" elements	355
SortDynArrayDouble	Compare two "array of double" elements	356
SortDynArrayInt64	Compare two "array of Int64 or array of Currency" elements	356
SortDynArrayInteger	Compare two "array of integer" elements	356
SortDynArrayPointer	Compare two "array of TObject/pointer" elements	356
SortDynArrayString	Compare two "array of generic string" elements, with case sensitivity	356
SortDynArrayStringI	Compare two "array of generic string" elements, with no case sensitivity	356
SortDynArrayUnicodeString	Compare two "array of WideString/UnicodeString" elements, with case sensitivity	356
SortDynArrayUnicodeStringI	Compare two "array of WideString/UnicodeString" elements, with no case sensitivity	356
SortDynArrayWord	Compare two "array of word" elements	356
SoundExAnsi	Retrieve the Soundex value of a text word, from Ansi buffer	356
SoundExUTF8	Retrieve the Soundex value of a text word, from UTF-8 buffer	356
Split	Split a RawUTF8 string into two strings, according to SepStr separator	356
SQLBegin	Go to the beginning of the SQL statement, ignoring all blanks and comments	357

Functions or procedures	Description	Page
StrComp	Use our fast version of StrComp(), to be used with PUTF8Char	357
StrCompIL	Use our fast version of StrCompIL(), to be used with PUTF8Char	357
StrCompL	Use our fast version of StrCompL(), to be used with PUTF8Char	357
StrCompW	Use our fast version of StrComp(), to be used with PWideChar	357
StrCurr64	Internal fast INTEGER Curr64 (value*10000) value to text conversion	357
StreamSynLZ	Compress a data content using the SynLZ algorithm from one stream into another	357
StreamUnSynLZ	Uncompress using the SynLZ algorithm from one stream into another	357
StreamUnSynLZ	Uncompress using the SynLZ algorithm from one file into another	357
StrIComp	Use our fast version of StrIComp()	357
StringBufferToUtf8	Convert any generic VCL Text buffer into an UTF-8 encoded buffer	358
StringFromFile	Read a File content into a String	358
StringReplaceAll	Fast replacement of StringReplace(S, OldPattern, NewPattern,[rfReplaceAll]);	358
StringReplaceChars	Fast replace of a specified char into a given string	358
StringToAnsi7	Convert any generic VCL Text into Ansi 7 bit encoded String	358
StringToRawUnicode	Convert any generic VCL Text into a Raw Unicode encoded String	358
StringToRawUnicode	Convert any generic VCL Text into a Raw Unicode encoded String	358
StringToSynUnicode	Convert any generic VCL Text into a SynUnicode encoded String	358
StringToUTF8	Convert any generic VCL Text into an UTF-8 encoded String	359
StringToWinAnsi	Convert any generic VCL Text into WinAnsi (Win-1252) 8 bit encoded String	359
StrInt32	Internal fast integer val to text conversion	359
StrInt64	Internal fast Int64 val to text conversion	359
StrLen	Our fast version of StrLen(), to be used with PUTF8Char	359
StrLenW	Our fast version of StrLen(), to be used with PWideChar	359
StrToCurr64	Convert a string into its INTEGER Curr64 (value*10000) representation	359
StrToCurrency	Convert a string into its currency representation	359
StrUInt32	Internal fast unsigned integer val to text conversion	359

Functions or procedures	Description	Page
SynUnicodeToString	Convert any SynUnicode encoded string into a generic VCL Text	359
SynUnicodeToUtf8	Convert a SynUnicode string into a UTF-8 string	360
TimeToIso8601	Basic Time conversion into ISO-8601	360
TimeToIso8601PChar	Write a Time to P^ Ansi buffer	360
TimeToIso8601PChar	Write a Time to P^ Ansi buffer	360
TimeToString	Retrieve the current Time (whithout Date), in the ISO 8601 layout	360
ToSBFStr	Convert any AnsiString content into our SBF compact binary format storage	360
ToVarInt32	Convert an integer into a 32-bit variable-length integer buffer	360
ToVarInt64	Convert a Int64 into a 64-bit variable-length integer buffer	360
ToVarUInt32	Convert a cardinal into a 32-bit variable-length integer buffer	360
ToVarUInt32Length	Return the number of bytes necessary to store a 32-bit variable-length integer	360
ToVarUInt32LengthWith Data	Return the number of bytes necessary to store some data with a its 32-bit variable-length integer legnth	360
ToVarUInt64	Convert a UInt64 into a 64-bit variable-length integer buffer	360
Trim	Use our fast asm RawUTF8 version of Trim()	360
TrimLeft	Trims leading whitespace characters from the string by removing new line, space, and tab characters	361
TrimLeftLowerCase	Trim first lowercase chars ('otDone' will return 'Done' e.g.)	361
TrimRight	Trims trailing whitespace characters from the string by removing trailing newline, space, and tab characters	361
TryEncodeTime	Try to encode a time	361
UInt32ToUtf8	Optimized conversion of a cardinal into RawUTF8	361
UnCamelCase	Convert a CamelCase string into a space separated one	361
UnCamelCase	Convert a CamelCase string into a space separated one	361
UnicodeBufferToString	Convert an Unicode buffer into a generic VCL string	361
UnicodeBufferToWinAnsi	Convert an Unicode buffer into a WinAnsi (code page 1252) string	361
UnicodeCharToUtf8	UTF-8 encode one Unicode character into Dest	361
UnQuoteSQLString	Unquote a SQL-compatible string	362

Functions or procedures	Description	Page
UnSetBit	Unset/clear a particular bit into a bit array	362
UnSetBit64	Unset/clear a particular bit into a Int64 bit array (max aIndex is 63)	362
UpdateIniEntry	Update a Name= Value in a [Section] of a INI RawUTF8 Content	362
UpdateIniEntryFile	Update a Name= Value in a [Section] of a .INI file	362
UpperCase	Fast conversion of the supplied text into uppercase	362
UpperCaseU	Fast conversion of the supplied text into 8 bit uppercase	362
UpperCaseUnicode	Accurate conversion of the supplied UTF-8 content into the corresponding upper-case Unicode characters	362
UpperCopy	Copy source into dest^ with 7 bits upper case conversion	362
UpperCopy255	Copy source into dest^ with 7 bits upper case conversion	363
UpperCopy255W	Copy WideChar source into dest^ with upper case conversion	363
UpperCopyShort	Copy source into dest^ with 7 bits upper case conversion	363
UrlDecode	Decode a string compatible with URI encoding into its original value	363
UrlDecode	Decode a string compatible with URI encoding into its original value	363
UrlDecodeCardinal	Decode a specified parameter compatible with URI encoding into its original cardinal numerical value	363
UrlDecodeExtended	Decode a specified parameter compatible with URI encoding into its original floating-point value	363
UrlDecodeInt64	Decode a specified parameter compatible with URI encoding into its original Int64 numerical value	363
UrlDecodeInteger	Decode a specified parameter compatible with URI encoding into its original integer numerical value	364
UrlDecodeNeedParameters	Returns TRUE if all supplied parameters does exist in the URI encoded text	364
UrlDecodeValue	Decode a specified parameter compatible with URI encoding into its original textual value	364
UrlEncode	Encode a string to be compatible with URI encoding	364
Utf8DecodeToRawUnicode	Convert a UTF-8 encoded buffer into a RawUnicode string	364
Utf8DecodeToRawUnicode	Convert a UTF-8 string into a RawUnicode string	364
Utf8DecodeToRawUnicodeUI	Convert a UTF-8 string into a RawUnicode string	364



Functions or procedures	Description	Page
UTF8DecodeToString	Convert any UTF-8 encoded buffer into a generic VCL Text	364
Utf8FirstLineToUnicodeLength	Calculate the character count of the first line UTF-8 encoded in source^	365
UTF8IComp	Fast UTF-8 comparaison using the NormToUpper[] array for all 8 bits values	365
UTF8ILComp	Fast UTF-8 comparaison using the NormToUpper[] array for all 8 bits values	365
Utf8ToRawUTF8	Direct conversion of a UTF-8 encoded zero terminated buffer into a RawUTF8 String	365
UTF8ToShortString	Direct conversion of a UTF-8 encoded buffer into a WinAnsi shortstring buffer	365
UTF8ToString	Convert any UTF-8 encoded String into a generic VCL Text	365
UTF8ToSynUnicode	Convert any UTF-8 encoded String into a generic SynUnicode Text	365
UTF8ToSynUnicode	Convert any UTF-8 encoded String into a generic SynUnicode Text	365
Utf8ToUnicodeLength	Calculate the Unicode character count (i.e. the glyph count), UTF-8 encoded in source^	365
UTF8ToWideChar	Convert an UTF-8 encoded text into a WideChar array	366
UTF8ToWideChar	Convert an UTF-8 encoded text into a WideChar array	366
UTF8ToWideString	Convert any UTF-8 encoded String into a generic WideString Text	366
UTF8ToWideString	Convert any UTF-8 encoded String into a generic WideString Text	366
UTF8ToWideString	Convert any UTF-8 encoded String into a generic WideString Text	366
Utf8ToWinAnsi	Direct conversion of a UTF-8 encoded zero terminated buffer into a WinAnsi String	366
Utf8ToWinAnsi	Direct conversion of a UTF-8 encoded string into a WinAnsi String	366
UTF8ToWinPChar	Direct conversion of a UTF-8 encoded buffer into a WinAnsi PAnsiChar buffer	366
UTF8UpperCopy	Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for all 8 bits values, encoding the result as UTF-8	366
UTF8UpperCopy255	Copy WideChar source into dest^ with upper case conversion, using the NormToUpper[] array for all 8 bits values, encoding the result as UTF-8	366
VariantToUTF8	Convert any Variant into UTF-8 encoded String	366
VarRecToUTF8	Convert an open array (const Args: array of const) argument to an UTF-8 encoded text	366
WideCharToWinAnsi	Conversion of a wide char into a WinAnsi (CodePage 1252) char index	367

Functions or procedures	Description	Page
WideCharToWinAnsiChar	Conversion of a wide char into a WinAnsi (CodePage 1252) char	367
WideStringToUTF8	Convert a WideString into a UTF-8 string	367
WideStringToWinAnsi	Convert a WideString into a WinAnsi (code page 1252) string	367
WinAnsiBufferToUtf8	Direct conversion of a WinAnsi PAnsiChar buffer into a UTF-8 encoded buffer	367
WinAnsiToRawUnicode	Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode encoded String	367
WinAnsiToUnicodeBuffer	Direct conversion of a WinAnsi (CodePage 1252) string into a Unicode buffer	367
WinAnsiToUtf8	Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 encoded String	367
WinAnsiToUtf8	Direct conversion of a WinAnsi (CodePage 1252) string into a UTF-8 encoded String	367
WriteStringToStream	Write an UTF-8 text into a TStream	367
YearToPChar	Add the 4 digits of integer Y to P^	367

**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*  
- 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=false): 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)  
- true if Value was added successfully in Values[], in this case length(Values) will be increased

**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'

**function** AddRawUTF8(**var** Values: TRawUTF8DynArray; **const** Value: RawUTF8; NoDuplicates: boolean=false; CaseSensitive: boolean=true): boolean;

*True if Value was added successfully in Values[]*

```
function AddSortedInteger(var Values: TIntegerDynArray; var ValuesCount: integer;
Value: integer; CoValues: PIntegerDynArray=nil): PtrInt;
```

*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 binary search)
- if CoValues is set, its content will be moved to allow inserting a new value at CoValues[result] position

```
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 alphabetically 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 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

```
function Ansi7ToString(Text: PWinAnsiChar; Len: integer): 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

```
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: PWinAnsiChar): 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 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 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*

```
procedure AppendToTextFile(aLine: RawUTF8; const aFileName: TFileName);
```

*Log a message to a local text file*

- this version expect 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'

```
procedure Base64Decode(sp,rp: PAnsiChar; len: PtrInt);
```

*Direct decoding of a Base64 encoded buffer*

```
function Base64ToBin(sp: PAnsiChar; len: PtrInt): RawByteString; overload;
```

*Fast conversion from Base64 encoded text into binary data*

```
function Base64ToBin(const s: RawByteString): RawByteString; overload;
```

*Fast conversion from Base64 encoded text into binary data*

```
function Base64ToBinLength(sp: PAnsiChar; len: PtrInt): PtrInt;
```

*Retrieve the expected length of a Base64 encoded buffer*

```
function BinToBase64(Bin: PAnsiChar; BinBytes: integer): RawByteString; overload;
```

*Fast conversion from binary data into Base64 encoded text*

```
function BinToBase64(const s: RawByteString): RawByteString; overload;
```

*Fast conversion from binary data into Base64 encoded text*

```
function BinToBase64URI(Bin: PAnsiChar; BinBytes: integer): RawByteString;
```

*Fast conversion from binary data into Base64-like URI-compatible encoded text*

- will trim any right-sided '=' insignificant characters, and replace '+' or '/' by '\_' or '-'

```
function BinToBase64WithMagic(const s: RawByteString): RawByteString; overload;
```

*Fast conversion from binary data into Base64 encoded text with JSON\_BASE64\_MAGIC prefix (UTF-8 encoded \uFFFF0 special code)*

```
function BinToBase64WithMagic(Data: pointer; DataLen: integer): RawByteString;
overload;
```

*Fast conversion from binary data into Base64 encoded text with JSON\_BASE64\_MAGIC prefix (UTF-8 encoded \uFFFF0 special code)*

```
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);

*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** CardinalToHex(aCardinal: Cardinal): RawUTF8;

*Fast conversion from a Cardinal data into hexa chars, ready to be displayed*

- use internally BinToHexDisplay()

**function** CharSetToCodePage(CharSet: integer): cardinal;

*Convert a char set to a code page*

**function** CodePageToCharSet(CodePage: Cardinal): Integer;

*Convert a code page to a char set*

**function** CompareMem(P1, P2: Pointer; Length: Integer): Boolean;

*Use our fast asm version of CompareMem()*

**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>=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>=SBFEnd (can be used for tftArray)
- returns true if both values match, or false otherwise

**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...soGreaterThanOrEqual 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** ContainsUTF8(p, up: PUTF8Char): boolean;

*Return true if up^ is contained inside the UTF-8 buffer p^*

- search up^ 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^ 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** CopyAndSortInteger(Values: PIntegerArray; ValuesCount: integer; var Dest: TIntegerDynArray);

*Copy an integer array, then sort it, low values first*

**function** CreateInternalWindow(const aWindowName: string; aObject: TObject): HWND;

*This function can be used to create a GDI compatible window, able to receive GDI 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 GDI message: for instance, define such a method in any object definition:

```
procedure WMCopyData(var Msg : TWMCopyData); message WM_COPYDATA;
```

**function** CSVOfValue(const Value: RawUTF8; Count: cardinal; const Sep: RawUTF8=' , '): RawUTF8;

*Return a CSV list of the iterated same value*

- e.g. CSVOfValue('?',3)='?',',',','

**procedure** CSVToIntegerDynArray(CSV: PUTF8Char; var Result: TIntegerDynArray);

*Add the strings in the specified CSV text into a dynamic array of integer*

**procedure** CSVToRawUTF8DynArray(CSV: PUTF8Char; var Result: TRawUTF8DynArray; Sep: AnsiChar = ' , ');

*Add the strings in the specified CSV text into a dynamic array of UTF-8 strings*

**function** Curr64ToPChar(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(Value: Int64): RawUTF8;

*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*

**function** DateTimeToIso8601(D: TDateTime; Expanded: boolean; FirstChar: AnsiChar='T'): RawUTF8;

*Basic Date/Time conversion into ISO-8601*

- use 'YYYYMMDDThhmmss' format if not Expanded
- use 'YYYY-MM-DDThh:mm:ss' format if Expanded

**function** DateTimeToIso8601Text(DT: TDateTime; FirstChar: AnsiChar='T'): 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'
- used e.g. by TPropInfo.GetValue() and TPropInfo.NormalizeValue() methods

**function** DateTimeToSQL(DT: TDateTime): 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 '\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'
- (JSON\_SQLDATE\_MAGIC will be used as prefix to create '\uFFF1...' pattern)
- to be used e.g. as in:  
aRec.CreateAndFillPrepare(Client, 'Datum<=?',[DateTimeToSQL(Now)]);

**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



**procedure** DateToIso8601PChar(P: PUTF8Char; Expanded: boolean; Y,M,D: cardinal); 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

**procedure** DateToIso8601PChar(Date: TDateTime; P: PUTF8Char; Expanded: boolean); 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** DateToIso8601Text(Date: TDateTime): RawUTF8;

*Convert a date into 'YYYY-MM-DD' date format*

- resulting text is compatible with all ISO-8601 functions

**function** DateToSQL(Date: TDateTime): RawUTF8;

*Convert a date to a ISO-8601 string format for SQL '?' inlined parameters*

- will return the date encoded as '\uFFFF1YYYY-MM-DD' - therefore '(:("\uFFFF12012-05-04")):' pattern will be recognized as a sftDateTime inline parameter in SQLParamContent() / ExtractInlineParameters() functions (JSON\_SQLDATE\_MAGIC will be used as prefix to create '\uFFFF1...' pattern)
- to be used e.g. as in:  
 aRec.CreateAndFillPrepare(Client, 'Datum=?', [DateToSQL(EncodeDate(2012,5,4))]);

**procedure** DeleteInteger(var Values: TIntegerDynArray; var ValuesCount: Integer; Index: PtrInt); overload;

*Delete any integer in Values[]*

**procedure** DeleteInteger(var Values: TIntegerDynArray; Index: PtrInt); overload;

*Delete any integer in Values[]*

**function** DeleteRawUTF8(var Values: TRawUTF8DynArray; var ValuesCount: integer; Index: integer; CoValues: PIntegerDynArray=nil): boolean;

*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

```
function DirectoryExists(const Directory: string): Boolean;
```

*DirectoryExists returns a boolean value that indicates whether the specified directory exists (and is actually a directory)*

```
function DoubleToStr(Value: Double): RawUTF8;
```

*Convert a 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:
- ```
var A: TIntegerDynArray;
begin
  with DynArray(TypeInfo(TIntegerDynArray),A) do
    begin
      (...)
    end;
```

```
function EventArchiveDelete(const aOldLogFileName, aDestinationPath: TFileName):
boolean;
```

*A TSynLogArchiveEvent handler which will delete older .log files*

```
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

**procedure** ExeVersionRetrieve(DefaultVersion: integer=0);

*Initialize ExeVersion global variable, if not already done*

**function** ExistsIniName(P: PUTF8Char; UpperName: PUTF8Char): boolean;

*Return TRUE if Value of UpperName does exist in P, till end of current section*

- expect UpperName as 'NAME='

**function** ExistsIniNameValue(P: PUTF8Char; UpperName: PUTF8Char): boolean;

*Return TRUE if the Value of UpperName exists in P, till end of current section*

- expect UpperName as 'NAME='

**function** ExtendedToStr(Value: Extended; Precision: integer): RawUTF8;

*Convert a floating-point value to its numerical text equivalency*

**function** ExtendedToString(var S: ShortString; Value: Extended; Precision: integer): integer;

*Convert a floating-point value to its numerical text equivalency*

- returns the count of chars stored into S (S[0] is not set)

**function** FastFindIntegerSorted(P: PIntegerArray; R: PtrInt; Value: integer): PtrInt;

*Fast 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^[index]=Value

- return -1 if Value was not found

**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 (so will work with any PAnsiChar)

- returns -1 if the specified Value was not found

**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 is case-sensitive (so will work with any PAnsiChar)

- returns -1 if the specified Value was not found

**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): 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 (so will work with any PAnsiChar)

- 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)

**function** FieldNameValid(P: PUTF8Char): boolean;

*Returns TRUE if the given text buffer contains A..Z,0..9 characters*  
 - i.e. can be tested via IdemPropName() functions  
 - first char must be alphabetical, following chars can be alphanumerical

**function** FileAgeToDateTime(const FileName: TFileName): TDateTime;

*Get the file date and time*  
 - returns 0 if file doesn't exist

**function** FileFromString(const Content: RawByteString; const FileName: TFileName; FlushOnDisk: boolean=false): boolean;

*Create a File from a string content*  
 - uses RawByteString for byte storage, whatever the codepage is

**function** FileSeek64(Handle: THandle; const Offset: Int64; Origin: DWORD): 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** FileSize(const FileName: TFileName): Int64;

*Get the file size*  
 - returns 0 if file doesn't exist

**function** FileSynLZ(const Source, Dest: TFileName; Magic: Cardinal): boolean;

*Compress a file content using the SynLZ algorithm a file content*  
 - source file is split into 128 MB blocks for fast in-memory compression of any file size  
 - you should specify a Magic number to be used to identify the compressed file format

**function** FileUnSynLZ(const Source, Dest: TFileName; Magic: Cardinal): boolean;

*Compress a file content using the SynLZ algorithm a file content*  
 - you should specify a Magic number to be used to identify the compressed file format

**procedure** FillChar(var Dest; Count: Integer; Value: Byte);

*Faster implementation of Move() for Delphi versions with no FastCode inside*  
 - Delphi RTL will be patched in memory to run this faster version

**procedure** FillIncreasing(Values: PIntegerArray; StartValue, Count: integer);

*Fill some values with i,i+1,i+2...i+Count-1*

**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): 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** 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: PUTF8Char): RawUTF8;

*Find the Value of UpperName in P, till end of current section*

- expect UpperName as 'NAME='

**function** FindIniNameValueInteger(P: PUTF8Char; UpperName: PUTF8Char): integer;

*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** 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 FindRawUTF8(const Values: TRawUTF8DynArray; const Value: RawUTF8;  
CaseSensitive: boolean=true): integer;
```

*Return the index of Value in Values[], -1 if not found*

```
function FindSectionFirstLine(var source: PUTF8Char; 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

```
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 expect source^ to point to an Unicode char array

```
function FindUnicode(PW: PWideChar; Upper: PWideChar; UpperLen: integer):  
boolean;
```

*Return true if Uppe (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 FindWinAnsiIniNameValue(P: PUTF8Char; UpperName: PUTF8Char): RawUTF8;
```

*Find the Value of UpperName in P, till end of current section*

- expect UpperName as 'NAME='

- same as FindIniNameValue(), but the value is converted from WinAnsi into UTF-8

```
function FormatUTF8(Format: PUTF8Char; const Args: array of const): RawUTF8;  
overload;
```

*Fast Format() function replacement, optimized for RawUTF8*

- only supported token is %, which will be inlined in the resulting string according to each Args[] supplied item

- resulting string has no length limit and uses fast concatenation

- maximum count of supplied argument in Args is 12

- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)

**function** FormatUTF8(Format: PUTF8Char; **const** Args, Params: array of **const**): 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 expected by SQLite3Commons, i.e. :(1234): for numerical values, and :('quoted " string'): for textual values
- resulting string has no length limit and uses fast concatenation
- maximum count of supplied argument in Args is 12
- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)

**function** FromVarInt32(**var** Source: PByte): PtrInt;

*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** FromVarString(**var** Source: PByte): RawUTF8;

*Retrieve a variable-length text buffer*

**function** FromVarUInt32(**var** Source: PByte): PtrUInt;

*Convert a 32-bit variable-length integer buffer into a cardinal*

**function** FromVarUInt32High(**var** Source: PByte): PtrUInt;

*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** FromVarUInt32Up128(**var** Source: PByte): PtrUInt;

*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

```
result := Source^;
inc(Source);
if result > $7f then
  result := (result and $7F) or FromVarUInt32Up128(Source);
```

**function** FromVarUInt64(**var** Source: PByte): QWord;

*Convert a 64-bit variable-length integer buffer into a UInt64*

**function** GetBit(**const** Bits; aIndex: PtrInt): boolean;

*Retrieve a particular bit status from a bit array*

**function** GetBit64(**const** Bits; aIndex: PtrInt): boolean;

*Retrieve a particular bit status from a Int64 bit array (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
- ',0' is always appended at the end of the CSV chunk to mark its end

**function** GetBitsCount(const Bits; Count: PtrInt): integer;

*Compute the number of bits set in a bit array*  
 - Count is the bit count, not byte size

**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+

**function** GetCardinal(P: PUTF8Char): PtrUInt;

*Get the unsigned 32 bits integer value stored in P^*  
 - we use the PtrInt result type, even if expected to be 32 bits, to use native CPU register size (don't want any 32 bits overflow here)

**function** GetCardinalDef(P: PUTF8Char; Default: PtrUInt): PtrUInt;

*Get the unsigned 32 bits integer value stored in P^*  
 - if P is nil or not start with a valid numerical value, returns Default

**function** GetCardinalW(P: PWideChar): PtrUInt;

*Get the unsigned 32 bits integer value stored as Unicode string in P^*

**function** GetCSVItem(P: PUTF8Char; Index: PtrUInt; Sep: AnsiChar = ','): RawUTF8;

*Return n-th indexed CSV string in P, starting at Index=0 for first one*

**function** GetCSVItemString(P: PChar; Index: PtrUInt; Sep: Char = ','): string;

*Return n-th indexed CSV string in P, starting at Index=0 for first one*  
 - this function return the generic string type of the compiler, and therefore can be used with ready to be displayed text (i.e. the VCL)

**function** GetDelphiCompilerVersion: RawUTF8;

*Return the Delphi Compiler Version*  
 - returns 'Delphi 2007' or 'Delphi 2010' e.g.

**function** GetDisplayNameFromClass(C: TClass): RawUTF8;

*Will get a class name as UTF-8*  
 - will trim 'T', 'TSyn', 'TSQL' or 'TSQLRecord' left side of the class name  
 - will encode the class name as UTF-8 (for Unicode Delphi versions)  
 - is used e.g. to extract the SQL table name for a TSQLRecord class

**function** GetEnumName(aTypeInfo: pointer; aIndex: integer): PShortString;

*Helper to retrieve the text of an enumerate item*  
 - you'd better use RTTI related classes of SQLite3Commons unit

**function** GetExtended(P: PUTF8Char): extended; 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; **out** err: integer): extended; 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): TFileName;

*Extract file name, without its extension*

**function** GetFileVersion(**const** FileName: TFileName): cardinal;

*GetFileVersion returns the most significant 32 bits of a file's binary version number*

- Typically, this includes the major and minor version placed together in one 32-bit integer

- It generally does not include the release or build numbers

- It returns Cardinal(-1) if it failed

**function** GetInt64(P: PUTF8Char; **var** err: integer): Int64; overload;

*Get the 64 bits 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 bits integer value stored in P^*

**function** GetInteger(P: PUTF8Char): PtrInt; overload;

*Get the signed 32 bits integer value stored in P^*

- we use the PtrInt result type, even if expected to be 32 bits, to use native CPU register size (don't want any 32 bits overflow here)

**function** GetInteger(P: PUTF8Char; **var** err: integer): PtrInt; overload;

*Get the signed 32 bits integer value stored in P^*

- this version return 0 in err if no error occurred, and 1 if an invalid character was found, not its exact index as for the val() function

- we use the PtrInt result type, even if expected to be 32 bits, to use native CPU register size (don't want any 32 bits overflow here)



**function** GetJSONField(P: PUTF8Char; out PDest: PUTF8Char; wasString: PBoolean=nil; EndOfObject: PUTF8Char=nil): PUTF8Char;

*Decode a JSON field in an UTF-8 encoded buffer (used in TSQLTableJSON.Create)*

- this function decodes in the P^ buffer memory itself (no memory allocation or copy), for faster process - so take care that it's an unique string
- PDest points to the next field to be decoded, or nil on any unexpected end
- null is decoded as nil
- "strings" are decoded as 'strings'
- strings are JSON unescaped (and \u0123 is converted to UTF-8 chars)
- any integer value is left as its ascii representation
- wasString is set to true if the JSON value was a "string"
- works for both field names or values (e.g. "FieldName":' or 'Value,')
- EndOfObject (if not nil) is set to the JSON value char (',' ':' or '}' e.g.)

*Used for DI-2.1.2 (page 830).*

**function** GetLineSize(P,PEnd: PUTF8Char): PtrUInt;

*Compute the line length from source array of chars*

- end counting at either #0, #13 or #10

**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*

**function** GetMimeType(Content: Pointer; Len: integer; const FileName: TFileName=''): RawUTF8;

*Retrieve the MIME content type from a supplied binary buffer*

- return the MIME type, ready to be appended to a 'Content-Type: ' HTTP header
- default is 'application/octet-stream' or 'application/extension' if FileName was specified
- see @[http://en.wikipedia.org/wiki/Internet\\_media\\_type](http://en.wikipedia.org/wiki/Internet_media_type) for most common values

**function** GetModuleName(Module: HMODULE): TFileName;

*Retrieve the full path name of the given execution module (e.g. library)*

**function** GetNextItem(var P: PUTF8Char; Sep: AnsiChar= ','): RawUTF8;

*Return next CSV string from P, nil if no more*

**function** GetNextItemCardinal(var P: PUTF8Char; Sep: AnsiChar= ','): PtrUInt;

*Return next CSV string as unsigned integer from P, 0 if no more*

**function** GetNextItemCardinalW(var P: PWideChar; Sep: WideChar= ','): PtrUInt;

*Return next CSV string as unsigned integer from P, 0 if no more*

- this version expect P^ to point to an Unicode char array

**function** GetNextItemDouble(var P: PUTF8Char; Sep: AnsiChar= ','): double;

*Return next CSV string as double from P, 0.0 if no more*

**procedure** GetNextItemShortString(var P: PUTF8Char; out Dest: ShortString; Sep: AnsiChar= ',');

*Return next CSV string from P, nil if no more*

**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** GetNextLine(source: PUTF8Char; **out** next: PUTF8Char): RawUTF8;

*Extract a line from source array of chars*

- next will contain the beginning of next line, or nil if source if ended

**function** GetNextLineBegin(source: PUTF8Char; **out** next: PUTF8Char): PUTF8Char;

*Return line begin from source array of chars, and go to next line*

- 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 RawUnicode string type

**function** GetNextUTF8Upper(**var** U: PUTF8Char): cardinal;

*Retrieve the next UCS2 value stored in U, then update the U pointer*

- this function will decode the UTF-8 content before using NormToUpper[]

- will return '?' if the UCS2 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** GetSectionContent(**const** Content, SectionName: RawUTF8): RawUTF8;  
overload;

*Retrieve the whole content of a section as a string*

- use SectionFirstLine() then previous GetSectionContent()

**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** GetUTF8Char(P: PUTF8Char): PtrUInt;

*Get the WideChar stored in P^ (decode UTF-8 if necessary)*

**function** GotoEndOfQuotedString(P: PUTF8Char): PUTF8Char;

*Get the next character after a quoted buffer*

- the first character in P^ must be either ', either "

**function** GotoNextJSONField(P: PUTF8Char; FieldCount: cardinal): PUTF8Char;

*Reach the position of the next JSON field in the supplied UTF-8 buffer*

**function** GotoNextJSONObjectOrArray(P: PUTF8Char): PUTF8Char;

*Reach the position of the next JSON object of JSON array*

- first char is expected to be either [ either {

- will return nil in case of parsing error or unexpected end (#0)

**function** GotoNextNotSpace(P: PUTF8Char): PUTF8Char;

*Get the next character not 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** Hash32(Data: pointer; Len: integer): cardinal; overload;

*Our custom hash function, specialized for Text comparaison*

- has less colision than Adler32 for short strings
- is faster than CRC32 or Adler32, since use DQWord (128 bytes) aligned read
- overloaded version for direct binary content hashing

**function** Hash32(const Text: RawByteString): cardinal; overload;

*Our custom hash function, specialized for Text comparaison*

- has less colision than Adler32 for short strings
- is faster than CRC32 or Adler32, since use DQWord (128 bytes) aligned read
- uses RawByteString for binary content hashing, thatever the codepage is

**function** HexDisplayToBin(Hex: PAnsiChar; Bin: PByte; BinBytes: integer): boolean;

*Fast conversion from hexa chars into a pointer*

**function** HexDisplayToCardinal(Hex: PAnsiChar; out aValue: cardinal): boolean;

*Fast conversion from hexa chars into a cardinal*

**function** HexToBin(Hex: PAnsiChar; Bin: PByte; BinBytes: Integer): boolean;

*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^
- using this function with Bin^ as an integer value will decode in big-endian order (most-significant byte first)

**function** IdemFileExt(p, extup: PUTF8Char): Boolean;

*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

**function** IdemPChar(p, up: PUTF8Char): boolean;

*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 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

**function** IdemPCharAndGetNextLine(var source: PUTF8Char; search: PUTF8Char): boolean;

*Return true if IdemPChar(source,search), and go to the next line of source*

**function** IdemPCharArray(p: PUTF8Char; const upArray: array of PUTF8Char): integer;

*Returns the index of a matching beginning of p^ in upArray[]*

- returns -1 if no item matched
- ignore case - up^ must be already Upper
- 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

**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. 'e' 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 expect p^ to point to an Unicode char array

**function** IdemPropName(const P1: shortstring; P2: PUTF8Char; P2Len: integer): boolean; overload;

*Case unsensitive test of P1 and P2 content*

- use it with properties only (A..Z,0..9 chars)
- this version expect P2 to be a PAnsiChar with a specified length

**function** IdemPropName(const P1,P2: shortstring): boolean; overload;

*Case unsensitive test of P1 and P2 content*

- use it with properties only (A..Z,0..9 chars)

**function** IdemPropNameU(const P1,P2: RawUTF8): boolean;

*Case unsensitive test of P1 and P2 content*

- use it with properties only (A..Z,0..9 chars)

**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** Int32ToUtf8(Value: integer): RawUTF8;

*Use our fast RawUTF8 version of IntToStr()*

- without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
- only usefull if our Enhanced Runtime (or LVCL) library is not installed

**procedure** Int64ToUInt32(Values64: PInt64Array; Values32: PCardinalArray; Count: integer);

*Copy some Int64 values into an unsigned integer array*

**function** Int64ToUtf8(Value: Int64): RawUTF8;

*Use our fast RawUTF8 version of IntToStr()*

- without any slow UnicodeString=String->AnsiString conversion for Delphi 2009
- only usefull if our Enhanced Runtime (or LVCL) library is not installed

**function** IntegerDynArrayLoadFrom(Source: PAnsiChar; var Count: integer): 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
- a bit faster than SimpleDynArrayLoadFrom(Source,TypeInfo(TIntegerDynArray),Count)

**function** IntegerDynArrayToCSV(const Values: TIntegerDynArray; ValuesCount: integer; const Prefix: RawUTF8=''; const Suffix: RawUTF8=''): RawUTF8;

*Return the corresponding CSV text from a dynamic array of integer*

- you can set some custom Prefix and Suffix text

**function** IntegerScan(P: PCardinalArray; Count: PtrInt; 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: PtrInt; Value: cardinal): boolean;

*Fast search of an unsigned integer position 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** IntToString(Value: Int64): 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: cardinal): string; overload;

*Faster version than default SysUtils.IntToStr implementation*

**function** IntToThousandString(Value: integer; const ThousandSep: RawUTF8=','): RawUTF8;

*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 IsAnsiCompatible(PW: PWideChar; Len: integer): boolean; overload;
    Return TRUE if the supplied buffer only contains 7-bits Ansi characters

function IsAnsiCompatible(PC: PAnsiChar): boolean; overload;
    Return TRUE if the supplied buffer only contains 7-bits Ansi characters

function IsAnsiCompatible(PC: PAnsiChar; Len: integer): boolean; overload;
    Return TRUE if the supplied buffer only contains 7-bits Ansi characters

function IsAnsiCompatible(PW: PWideChar): 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 IsBase64(const s: RawByteString): boolean; overload;
    Check if the supplied text is a valid Base64 encoded stream

function IsBase64(sp: PAnsiChar; len: PtrInt): boolean; overload;
    Check if the supplied text is a valid Base64 encoded stream

function IsContentCompressed(Content: Pointer; Len: integer): 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 zip/gz/gif/wma/png/jpeg markers

function IsIso8601(P: PUTF8Char; L: integer): boolean;
    Test if P^ contains a valid ISO-8601 text encoded value
    - calls internally Iso8601ToSecondsPUTF8Char() and returns true if contains at least a valid
    year (YYYY)

function IsMatch(const Pattern, Text: RawUTF8; CaseInsensitive: boolean=false):
boolean;
    Return TRUE if the supplied content matchs to a grep-like pattern
    - ?           Matches any single characer      - *           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  - [!abc] Matches anything but a or b or c at that position      - [a-e]   Matches a
    through e at that position
    - [abcx-z]   Matches a or b or c or x or y or 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'
    - initial C version by Kevin Boylan, first Delphi port by Sergey Seroukhov

function Iso8601FromDateTime(DateTime: TDateTime): Int64;
    Get TTimeLog value from a given Delphi date and time

function Iso8601FromFile(const FileName: TFileName): Int64;
    Get TTimeLog value from a file date and time

function Iso8601Now: Int64;
    Get TTimeLog value from current date and time

```

```
function Iso8601ToDateTime(const S: RawUTF8): TDateTime;
```

*Date/Time conversion from ISO-8601*

- handle 'YYYYMMDDThhmmss' and 'YYYY-MM-DD hh:mm:ss' format

```
function Iso8601ToDateTimePUTF8Char(P: PUTF8Char; L: integer=0): TDateTime;
```

*Date/Time conversion from ISO-8601*

- handle 'YYYYMMDDThhmmss' and 'YYYY-MM-DD hh:mm:ss' format

- if L is left to default 0, it will be computed from StrLen(P)

```
procedure Iso8601ToDateTimePUTF8CharVar(P: PUTF8Char; L: integer; var result:  
TDateTime);
```

*Date/Time conversion from ISO-8601*

- handle 'YYYYMMDDThhmmss' and 'YYYY-MM-DD hh:mm:ss' format

- if L is left to default 0, it will be computed from StrLen(P)

```
function Iso8601ToSeconds(const S: RawUTF8): Int64;
```

*Convert a Iso8601 encoded string into a "fake" second count*

- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds

- use this function only for fast comparison between two Iso8601 date/time

- conversion is faster than Iso8601ToDateTime: use only binary integer math

```
function Iso8601ToSecondsPUTF8Char(P: PUTF8Char; L: integer; ContainsNoTime:  
PBoolean=nil): QWord;
```

*Convert a Iso8601 encoded string into a "fake" second count*

- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds

- 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 Iso8601ToSQL(TimeStamp: TTimeLog): RawUTF8;
```

*Convert an Iso8601 date/time (bit-encoded as Int64) to a ISO-8601 string format for SQL '?' inlined parameters*

- 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<=?', [Iso8601ToSQL(Iso8601Now)]);
```

```
function IsRowID(FieldName: PUTF8Char): boolean; overload;
```

*Returns TRUE if the specified field name is either 'ID', either 'ROWID'*

```
function IsRowID(const FieldName: shortstring): boolean; overload;
```

*Returns TRUE if the specified field name is either 'ID', either 'ROWID'*



**function** `isSelect(P: PUTF8Char): 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' statement also returns true, since doesn't change the data content

**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. `IsString('0')=false`, `IsString('abc')=true`, `IsString('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** `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** `IsValidIPAddress(P: PUTF8Char): boolean;`

*Return TRUE if the supplied content is a valid IP v4 address*

**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(P: pointer; Length: integer): boolean; overload;`

*Returns TRUE if all bytes equal zero*

**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



```
procedure JSONDecode(var JSON: RawUTF8; const Names: array of PUTF8Char; var
Values: TPUTf8CharDynArray; 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]^='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 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

*Used for DI-2.1.2 (page 830).*

```
function JSONDecode(P: PUTF8Char; const Names: array of PUTF8Char; var Values:
TPUTF8CharDynArray; 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
- returns a pointer to the next content item in the JSON buffer

*Used for DI-2.1.2 (page 830).*

```
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 procedure will decode the JSON content in-memory, so must be called only once with the same JSON data

*Used for DI-2.1.2 (page 830).*

```
function JSONEncode(const NameValuePairs: array of const): RawUTF8;
```

*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}'
- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)

*Used for DI-2.1.2 (page 830).*

```
function JSONEncodeArray(const Values: array of double): RawUTF8; overload;
```

*Encode the supplied floating-point array data as an UTF-8 valid JSON array content*

*Used for DI-2.1.2 (page 830).*

**function** JSONEncodeArray(const Values: array of integer): RawUTF8; overload;

*Encode the supplied integer array data as an UTF-8 valid JSON array content*

*Used for DI-2.1.2 (page 830).*

**function** JSONEncodeArray(const Values: array of RawUTF8): RawUTF8; overload;

*Encode the supplied RawUTF8 array data as an UTF-8 valid JSON array content*

*Used for DI-2.1.2 (page 830).*

**function** KB(bytes: Int64): RawUTF8;

*Convert a size to a human readable value*

- append MB, KB or B symbol
- for MB and KB, add one fractional digit

**function** kr32(crc: cardinal; buf: PAnsiChar; len: cardinal): cardinal;

*Standard Kernighan & Ritchie hash from "The C programming Language", 3rd edition*

- not the best, but simple and efficient code - perfect for THasher

**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

**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 convert 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** MicroSecToString(Micro: Int64): RawUTF8;

*Convert a micro seconds elapsed time into a human readable value*

- append us, ms or s symbol
- for us and ms, add two fractional digits

**procedure** Move(const Source; var Dest; Count: Integer);

*Faster implementation of Move() for Delphi versions with no FastCode inside*

- Delphi RTL will be patched in memory to run this faster version

**function** NextUTF8Char(P: PUTF8Char; **out** Next: PUTF8Char): PtrUInt;

*Get the WideChar stored in P^ (decode UTF-8 if necessary) and set new pos to Next*

**function** NowToString(Expanded: boolean=true; FirstTimeChar: AnsiChar = ' '): RawUTF8;

*Retrieve the current Date, in the ISO 8601 layout, but expanded and ready to be displayed*

**procedure** PatchCode(Old, New: pointer; Size: integer; Backup: pointer=nil);

*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: PtrUInt);

*Self-modifying code - change one PtrUInt in the code segment*

**function** PointerToHex(aPointer: Pointer): RawUTF8;

*Fast conversion from a pointer data into hexa chars, ready to be displayed*

- use internally BinToHexDisplay()

**function** PosChar(Str: PUTF8Char; Chr: AnsiChar): PUTF8Char;

*Fast retrieve the position of a given character*

**function** PosEx(const SubStr, S: RawUTF8; Offset: Cardinal = 1): Integer;

*Faster RawUTF8 Equivalent of standard StrUtils.PosEx*

**function** PosI(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 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[]

**procedure** QuickSortInteger(ID: PIntegerArray; L, R: PtrInt);

*Sort an Integer array, low values first*

**procedure** QuickSortRawUTF8(var Values: TRawUTF8DynArray; ValuesCount: integer; CoValues: PIntegerDynArray=nil; Compare: TUTF8Compare=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

**function** QuotedStr(const S: RawUTF8; Quote: AnsiChar=''): RawUTF8; overload;

*Format a text content with 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(Text: PUTF8Char; Quote: AnsiChar): RawUTF8; overload;

*Format a buffered text content with quotes*

- 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** RawByteArrayConcat(const Values: array of RawByteString): RawByteString;

*Fast concatenation of several AnsiStrings*

**function** RawUnicodeToString(const U: RawUnicode): string; overload;

*Convert any Raw Unicode encoded string 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** RawUnicodeToString(P: PWideChar; L: integer): 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(P: PWideChar; WideCharCount: integer): SynUnicode; overload;

*Convert any Raw Unicode encoded String into a generic SynUnicode Text*

**function** RawUnicodeToUtf8(Dest: PUTF8Char; DestLen: PtrInt; Source: PWideChar; SourceLen: PtrInt): PtrInt; overload;

*Convert a RawUnicode PWideChar into a UTF-8 buffer*

- replace system.UnicodeToUtf8 implementation, which is rather slow since Delphi 2009+

**function** RawUnicodeToUtf8(const Unicode: RawUnicode): RawUTF8; overload;

*Convert a RawUnicode string into a UTF-8 string*

**function** RawUnicodeToUtf8(P: 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** RawUnicodeToUtf8(P: PWideChar; WideCharCount: integer): RawUTF8; overload;

*Convert a RawUnicode PWideChar into a UTF-8 string*

**function** RawUnicodeToWinAnsi(P: PWideChar; WideCharCount: integer): WinAnsiString; overload;

*Convert a RawUnicode PWideChar into a WinAnsi (code page 1252) string*

**function** RawUnicodeToWinAnsi(const Unicode: RawUnicode): WinAnsiString; overload;

*Convert a RawUnicode string 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

**function** RawUTF8DynArrayEquals(const A,B: TRawUTF8DynArray): boolean;

*True if both TRawUTF8DynArray are the same*  
 - comparison is case-sensitive

**function** RawUTF8DynArrayLoadFromContains(Source: PAnsiChar; Value: PUTF8Char; ValueLen: integer; CaseSensitive: boolean): integer;

*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** 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

**procedure** RecordClear(var Dest; TypeInfo: pointer);

*Clear a record content*

**procedure** RecordCopy(var Dest; const Source; TypeInfo: pointer);

*Copy a record content from source to Dest*  
 - this unit includes a fast optimized asm version

**function** RecordEquals(const RecA, RecB; TypeInfo: pointer): 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 Rec; Source: PAnsiChar; TypeInfo: pointer): PAnsiChar;

*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

**function** RecordLoadJSON(var Rec; JSON: PUTF8Char; TypeInfo: pointer; EndOfObject: PUTF8Char=nil): PUTF8Char;

*Fill a record content from a JSON serialization as saved by TTextWriter.AddRecordJSON*  
 - will handle both default (Bin64 encoding of Record Save binary) and custom true JSON format (as set by TTextWriter.RegisterCustomJSONSerializer)

**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
- 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 or even RawUnicode

**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
- 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 or even RawUnicode

**function** RecordSaveLength(const Rec; TypeInfo: pointer): 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 a variant or a dynamic array)

**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 Size array of bytes, 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** ReleaseInternalWindow(var aWindowName: string; var aWindow: HWND):  
 boolean;

*Delete the window resources used to receive GDI messages*

- must be called for each CreateInternalWindow() function
- both parameter values are then reset to "/0

**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** 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



**function** RoundTo2Digits(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** 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 the UTF-8 content before using NormToUpper[]

**function** SameValue(const A, B: Double; DoublePrec: double = 1E-12): 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** SetBit(var Bits; aIndex: PtrInt);

*Set a particular bit into a bit array*

**procedure** SetBit64(var Bits: Int64; aIndex: PtrInt);

*Set a particular bit into a Int64 bit array (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

**procedure** SetInt64(P: PUTF8Char; var result: Int64);

*Get the 64 bits integer value stored in P^*

**function** SimpleDynArrayLoadFrom(Source: PAnsiChar; aTypeInfo: pointer; var Count, ElemSize: integer): 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** 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** 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** 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** 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** SortDynArrayWord(**const** A,B): integer;

*Compare two "array of word" elements*

**function** SoundExAnsi(A: PAnsiChar; next: PPAnsiChar=nil; Lang:

TSynSoundExPronunciation=sndxEnglish): 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

**procedure** Split(**const** Str, SepStr: RawUTF8; **var** LeftStr, RightStr: RawUTF8;  
ToUpperCase: boolean=false);

*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** 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** StrComp(Str1, Str2: PUTF8Char): PtrInt;

*Use our fast version of StrComp(), to be used with PUTF8Char*

**function** StrCompIL(P1,P2: PUTF8Char; L, Default: Integer): 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** 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** 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: PUTF8Char): PtrInt;

*Use our fast version of StrIComp()*

**function** StringBufferToUtf8(Dest: PUTF8Char; Source: PChar; SourceChars: PtrInt): PUTF8Char;

*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)

**function** StringFromFile(const FileName: TFileName): RawByteString;

*Read a File content into a String*

- content can be binary or text
- returns "" if file was not found or any read error occurred
- uses RawByteString for byte storage, whatever the codepage is

**function** StringReplaceAll(const S, OldPattern, NewPattern: RawUTF8): RawUTF8;

*Fast replacement of StringReplace(S, OldPattern, NewPattern, [rfReplaceAll]);*

**function** StringReplaceChars(const Source: RawUTF8; OldChar, NewChar: AnsiChar): RawUTF8;

*Fast replace of a specified char into 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** 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 SQLite3i18n, 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 SQLite3i18n, 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;

*Convert any generic VCL Text into a SynUnicode encoded String*

- it's preferred to use TLanguageFile.StringToUTF8() method in SQLite3i18n, 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** StringToUTF8(const Text: string): RawUTF8;

*Convert any generic VCL Text into an UTF-8 encoded String*

- it's preferred to use TLanguageFile.StringToUTF8() method in SQLite3i18n, 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:

```
function Int32ToUTF8(Value : integer): RawUTF8;
var tmp: array[0..15] of AnsiChar;
    P: PAnsiChar;
begin
  P := StrInt32(@tmp[15],Value);
  SetString(result,P,@tmp[15]-P);
end;
```

- not to be called directly: use IntToStr() instead

**function** StrInt64(P: PAnsiChar; val: Int64): PAnsiChar;

*Internal fast Int64 val to text conversion*

- same calling convention as with StrInt32() above

**function** StrLen(S: PUTF8Char): PtrInt;

*Our fast version of StrLen(), to be used with PUTF8Char*

**function** StrLenW(S: PWideChar): PtrInt;

*Our fast version of StrLen(), to be used with PWideChar*

**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 StrToCurr64()

**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^)

**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** TimeToIso8601(Time: TDateTime; Expanded: boolean; FirstChar: AnsiChar='T'): RawUTF8;

*Basic Time conversion into ISO-8601*

- use 'Thhmmss' format if not Expanded
- use 'Thh:mm:ss' format if Expanded

**procedure** TimeToIso8601PChar(Time: TDateTime; P: PUTF8Char; Expanded: boolean; FirstChar: AnsiChar = 'T'); 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

**procedure** TimeToIso8601PChar(P: PUTF8Char; Expanded: boolean; H,M,S: cardinal; FirstChar: AnsiChar = 'T'); 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

**function** TimeToString: RawUTF8;

*Retrieve the current Time (without Date), in the ISO 8601 layout*

- usefull for direct on screen logging e.g.

**procedure** ToSBFStr(const Value: RawByteString; out Result: TSBFString);

*Convert any AnsiString content into our SBF compact binary format storage*

**function** ToVarInt32(Value: PtrInt; 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** ToVarUInt32(Value: PtrUInt; Dest: PByte): PByte;

*Convert a cardinal into a 32-bit variable-length integer buffer*

**function** ToVarUInt32Length(Value: PtrUInt): PtrUInt;

*Return the number of bytes necessary to store a 32-bit variable-length integer*

- i.e. the ToVarUInt32() buffer size

**function** ToVarUInt32LengthWithData(Value: PtrUInt): PtrUInt;

*Return the number of bytes necessary to store some data with a its 32-bit variable-length integer legnth*

**function** ToVarUInt64(Value: QWord; Dest: PByte): PByte;

*Convert a UInt64 into a 64-bit variable-length integer buffer*

**function** Trim(const S: RawUTF8): RawUTF8;

*Use our fast asm RawUTF8 version of Trim()*

**function** TrimLeft(const S: RawUTF8): RawUTF8;

*Trims leading whitespace characters from the string by removing new line, space, and tab characters*

**function** TrimLeftLowerCase(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+

**function** TrimRight(const S: RawUTF8): RawUTF8;

*Trims trailing whitespace characters from the string by removing trailing newline, space, and tab characters*

**function** TryEncodeTime(Hour, Min, Sec, MSec: Word; var Time: TDateTime): Boolean;

*Try to encode a time*

**function** UInt32ToUtf8(Value: cardinal): RawUTF8;

*Optimized conversion of a cardinal into RawUTF8*

**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'

- '\_' 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** 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'

- 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** 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** UnicodeCharToUtf8(Dest: PUTF8Char; aWideChar: PtrUInt): integer;

*UTF-8 encode one Unicode character into Dest*

- return the number of bytes written into Dest (i.e. 1,2 or 3)

**function** UnQuoteSQLString(P: PUTF8Char; out Value: RawUTF8): PUTF8Char;

*Unquote a SQL-compatible string*

- the first character in P^ must be either ' , either " then 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*

**procedure** UnSetBit64(var Bits: Int64; aIndex: PtrInt);

*Unset/clear a particular bit into a Int64 bit array (max aIndex is 63)*

**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 "", update the Name= value before any [Section]
- use internally fast UpdateIniEntry() function above

**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

**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 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 255 AnsiChar (expect the dest buffer to be array[byte] of AnsiChar)

**function** UpperCopy255(dest: PAnsiChar; **const** source: RawUTF8): PAnsiChar;

*Copy source into dest^ with 7 bits upper case conversion*

- 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*

- 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** UrlDecode(**const** s: RawUTF8; i: PtrInt = 1; len: PtrInt = -1): RawUTF8;  
overload;

*Decode a string compatible with URI encoding into its original value*

- you can specify the decoding range (as in copy(s,i,len) function)

**function** UrlDecode(U: PUTF8Char): RawUTF8; overload;

*Decode a string compatible with URI encoding into its original value*

**function** UrlDecodeCardinal(U, Upper: PUTF8Char; **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** UrlDecodeExtended(U, Upper: PUTF8Char; **var** Value: Extended; 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, Upper: PUTF8Char; **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, Upper: PUTF8Char; 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

**function** UrlDecodeNeedParameters(U, CSVUpper: PUTF8Char): boolean;

*Returns TRUE if all supplied parameters does exist in the URI encoded text*

- UrlDecodeNeedParameters('price=20.45&where=LastName%3D','PRICE,WHERE') will return TRUE

**function** UrlDecodeValue(U, Upper: PUTF8Char; var Value: RawUTF8; Next: PPUTF8Char=nil): boolean;

*Decode a specified parameter compatible with URI encoding into its original textual value*

-

UrlDecodeValue('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** UrlEncode(const svar: RawUTF8): RawUTF8; overload;

*Encode a string to be compatible with URI encoding*

**function** Utf8DecodeToRawUnicode(const S: RawUTF8): RawUnicode; overload;

*Convert a UTF-8 string into a RawUnicode string*

**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 WideString

**function** Utf8DecodeToRawUnicodeUI(const S: RawUTF8; DestLen: PInteger=nil): RawUnicode;

*Convert a UTF-8 string into a RawUnicode string*

- this version doesn't resize the length of the result RawUnicode and is therefore usefull 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): string;

*Convert any UTF-8 encoded buffer into a generic VCL Text*

- it's preferred to use TLanguageFile.UTF8ToString() in SQLite3i18n, 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** Utf8FirstLineToUnicodeLength(source: PUTF8Char): PtrInt;

*Calculate the character count of the first line UTF-8 encoded in source^*  
 - end the count at first #13 or #10 character

**function** UTF8IComp(u1, u2: PUTF8Char): PtrInt;

*Fast UTF-8 comparaison using the NormToUpper[] array for all 8 bits values*  
 - this version expects u1 and u2 to be zero-terminated  
 - this version will decode the UTF-8 content before using NormToUpper[]

**function** UTF8ILComp(u1, u2: PUTF8Char; L1,L2: cardinal): PtrInt;

*Fast UTF-8 comparaison 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[]

**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 SQLite3i18n, 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(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 Unicode character count (i.e. the glyph count), UTF-8 encoded in source^*  
 - faster than System.UTF8ToUnicode with dest=nil

**function** UTF8ToWideChar(dest: pWideChar; source: PUTF8Char; sourceBytes: PtrInt=0): PtrInt; overload;

*Convert an UTF-8 encoded text into a WideChar array*  
 - faster than System.UTF8ToUnicode  
 - sourceBytes can be 0, therefore length is computed from zero terminated source  
 - enough place must be available in dest  
 - a WideChar(#0) is added at the end (if something is written)  
 - returns the BYTE count written in dest, excluding the ending WideChar(#0)

```
function UTF8ToWideChar(dest: pWideChar; source: PUTF8Char; MaxDestChars,  
sourceBytes: PtrInt): PtrInt; overload;
```

*Convert an UTF-8 encoded text into a WideChar array*

- 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
- a WideChar(#0) is added at the end (if something is written)
- returns the byte count written in dest, excluding the ending WideChar(#0)

```
function UTF8ToWideString(const Text: RawUTF8): WideString; overload;  
Convert any UTF-8 encoded String into a generic WideString Text
```

```
procedure UTF8ToWideString(Text: PUTF8Char; Len: integer; var result:  
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 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

```
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 VariantToUTF8(const V: Variant): RawUTF8;  
Convert any Variant into UTF-8 encoded String
```

- will use an internal SynUnicode temporary conversion

```
procedure VarRecToUTF8(const V: TVarRec; var result: RawUTF8);
```

*Convert an open array (const Args: array of const) argument to an UTF-8 encoded text*

- note that cardinal values should be type-casted to Int64() (otherwise the integer mapped value will be transmitted, therefore wrongly)

**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

**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: integer);

*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: integer): 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

**procedure** WriteStringToStream(S: TStream; const Text: RawUTF8);

*Write an UTF-8 text into a TStream*  
 - format is Length(Integer):Text, i.e. the one used by ReadStringFromStream

**procedure** YearToPChar(Y: Word; P: PUTF8Char);

*Add the 4 digits of integer Y to P^*

**Variables implemented in the SynCommons unit:**

**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 SQLite3i18n.pas unit, if the ENHANCEDRTL conditional is not defined

**ConvertHexToBin: array[byte] of byte;**

*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

**CurrentAnsiConvert: 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 instantiated during the whole program life time

**DefaultHasher: THasher;**

*The default hasher used by TDynArrayHashed()*

- is set to kr32() function above
- should be set to faster and more accurate crc32() function if available (this is what SQLite3Commons unit does in its initialization block)

**ExeVersion: record**

*Global information about the current executable and computer*

- call ExeVersionRetrieve before using it

**GarbageCollector: TObjectList;**

*A global "Garbage collector", for some classes instances which must live for all the main executable process*

- used to avoid any memory leak with e.g. 'class var RecordProps'

**i18nDateText: function(Iso: TTimeLog): 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 current i18n settings
- used by TSQLTable.ExpandAsString() method, i.e. TSQLTableToGrid.DrawCell()

**IsWow64: boolean;**

*Is set to TRUE if the current process is running under WOW64*

- WOW64 is the x86 emulator that allows 32-bit Windows-based applications to run seamlessly on 64-bit Windows

**LoadResStringTranslate: procedure(var Text: string) = nil;**

*These procedure type must be defined if a default system.pas is used*

- SQLite3i18n 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)

**NormToLower: TNormTable;**

*The NormToLower[] array is defined in our Enhanced RTL: define it now if it was not installed*

**NormToUpper: TNormTable;**

*The NormToUpper[] array is defined in our Enhanced RTL: define it now if it was not installed*

**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

**OSVersion: TWindowsVersion;**

*The current Operating System version, as retrieved for the current process*

**OSVersionInfo: TOSVersionInfoEx;**

*The current Operating System information, as retrieved for the current process*

**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 IpMinimumApplicationAddress field is replaced by a more optimistic/realistic value (\$100000 instead of default \$10000)

**TSynLogTestLog: TSynLogClass = TSynLog;**

*The kind of .log file generated by TSynTestsLogged*

**TwoDigitLookupW: packed[0..99] of word absolute TwoDigitLookup;**

*Fast lookup table for converting any decimal number from 0 to 99 into their ASCII equivalence*

**WinAnsiConvert: TSynAnsiFixedWidth;**

*Global TSynAnsiConvert instance to handle WinAnsi encoding (code page 1252)*

- this instance is global and instantiated during the whole program life time

### 1.4.7.3. SynCrtSock unit

*Purpose:* Classes implementing HTTP/1.1 client and server protocol

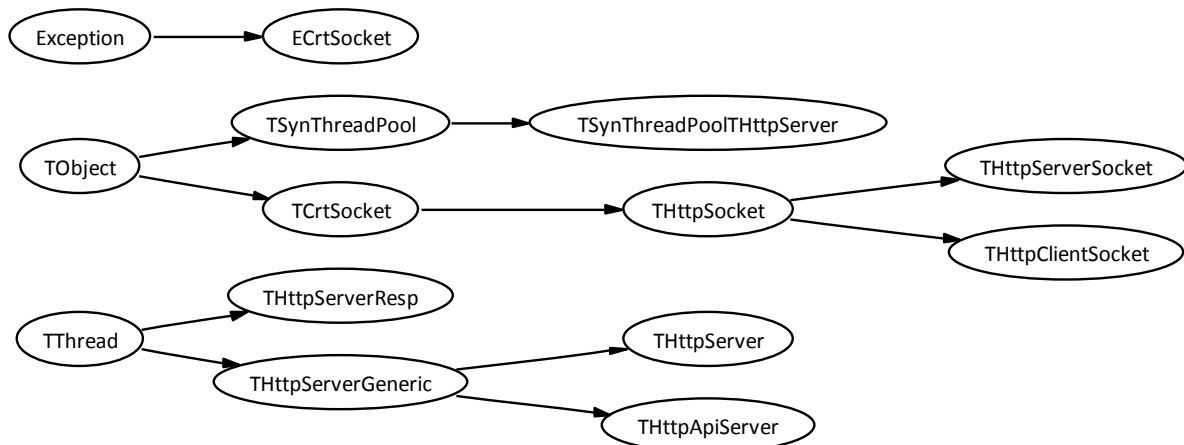
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**The SynCrtSock unit is quoted in the following items:**

| SWRS #       | Description       | Page |
|--------------|-------------------|------|
| DI-2.1.1.2.4 | HTTP/1.1 protocol | 830  |

**Units used in the SynCrtSock unit:**

| Unit Name  | Description                                                                                                                                                                   | Page |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| SynWinSock | Low level access to network Sockets for the Win32 platform<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.6 | 559  |



*SynCrtSock class hierarchy*

### Objects implemented in the *SynCrtSock* unit:

| Objects                   | Description                                                                 | Page |
|---------------------------|-----------------------------------------------------------------------------|------|
| ECrtSocket                | Exception thrown by the classes of this unit                                | 370  |
| TCrtSocket                | Fast low-level Socket implementation                                        | 371  |
| THttpApiServer            | HTTP server using fast http.sys kernel-mode server                          | 379  |
| THttpClientSocket         | REST and HTTP/1.1 compatible client class                                   | 376  |
| THttpServer               | Main HTTP server Thread                                                     | 380  |
| THttpServerGeneric        | Generic HTTP server                                                         | 378  |
| THttpServerResp           | HTTP response Thread                                                        | 377  |
| THttpServerSocket         | HTTP/1.1 server class used by THttpServer main server Thread                | 375  |
| THttpSocket               | Parent of THttpClientSocket and THttpServerSocket classes                   | 374  |
| THttpSocketCompressRec    | Used to maintain a list of known compression algorithms                     | 374  |
| TSynThreadPool            | A simple Thread Pool, used for fast handling HTTP requests                  | 377  |
| TSynThreadPoolTHttpServer | A simple Thread Pool, used for fast handling HTTP requests of a THttpServer | 377  |

**ECrtSocket** = **class**(Exception)

*Exception thrown by the classes of this unit*

## TCrtSocket = class(TObject)

*Fast low-level Socket implementation*

- direct access to the OS (Windows, Linux) network layer
- use **Open** constructor to create a client to be connected to a server
- use **Bind** constructor to initialize a server
- use direct access to low level Windows or Linux network layer
- use **SockIn** and **SockOut** (after **CreateSock\***) to read or write data as with standard Delphi text files (see **SendEmail** implementation)
- if app is multi-threaded, use faster **SockSend()** instead of **SockOut^** for direct write access to the socket; but **SockIn^** is much faster than **SockRecv()** thanks to its internal buffer, even on multi-threaded app (at least under Windows, it may be up to 10 times faster)
- but you can decide whatever to use none, one or both **SockIn/SockOut**
- our classes are much faster than the Indy or Synapse implementation

**BytesIn:** cardinal;  
*Total bytes received*

**BytesOut:** cardinal;  
*Total bytes sent*

**Port:** AnsiString;  
*Initialized after Open() with port number*

**Server:** AnsiString;  
*Initialized after Open() with Server name*

**Sock:** TSocket;  
*Initialized after Open() with socket*

**SockIn:** ^TextFile;  
*After CreateSockIn, use Readln(SockIn,s) to read a line from the opened socket*

**SockOut:** ^TextFile;  
*After CreateSockOut, use Writeln(SockOut,s) to send a line to the opened socket*

**TimeOut:** cardinal;  
*If higher than 0, read loop will wait for incoming data till TimeOut milliseconds (default value is 10000) - used also in SockSend()*

**constructor** Bind(const aPort: AnsiString; aLayer: TCrtSocketLayer=cslTCP);  
*Bind to aPort*

**constructor** Open(const aServer, aPort: AnsiString; aLayer: TCrtSocketLayer=cslTCP);  
*Connect to aServer:aPort*

**destructor** Destroy; **override**;  
*Close the opened socket, and corresponding SockIn/SockOut*



**function** SockCanRead(aTimeOut: cardinal): integer;

*Wait till some data is pending in the receiving queue within TimeOut milliseconds*

- returns 1 if there is some data to be read
- returns 0 if there is no data to be read
- returns <0 on any socket error

**function** SockCanWrite(aTimeOut: cardinal): integer;

*Wait till some data can be sent within TimeOut milliseconds*

- returns >0 if data can be written
- returns 0 if data can not be written
- returns <0 on any socket error

**function** SockConnected: boolean;

*Check the connection status of the socket*

**function** SockInRead(Content: PAnsiChar; Length: integer): 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
- can be used also without SockIn: it will call directly SockRecv() in such case

**function** SockReceiveString(aTimeOut : integer = 300): RawByteString;

*Returns the socket input stream as a string*

- specify the Max time to wait until some data is available for reading
- if the TimeOut parameter is 0, wait until something is available

**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; Length: integer): boolean;

*Fill the Buffer with Length bytes*

- use TimeOut milliseconds wait for incoming data
- bypass the SockIn^ buffers
- return false on any error, true on success

**procedure** CreateSockIn(LineBreak: TTextLineBreakStyle=tlbsCRLF);

*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;

*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: AnsiString; doBind: boolean; aSock:
integer=-1; aLayer: TCrtSocketLayer=cslTCP);
```

*Raise an ECrtSocket exception on error (called by above constructors)*

```
procedure Snd(P: pointer; Len: integer);
```

*Append P^ data into SndBuf (used by SockSend(), e.g.)*

- call SockSendFlush to send it through the network via SndLow()

```
procedure SndLow(P: pointer; Len: integer);
```

*Direct send data through network*

- raise a ECrtSocket exception on any error

- bypass the SndBuf 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; 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 SockRecvLn(out Line: RawByteString; CROnly: boolean=false); overload;
```

*Call readln(SockIn^,Line) 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

- 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 SockSend(const Values: array of const); overload;
```

*Simulate writeln() with direct use of Send(Sock, ..)*

- usefull on multi-treaded environnement (as in THttpServer.Process)

- no temp buffer is used

- handle RawByteString, ShortString, Char, Integer parameters

- raise ECrtSocket exception on socket error

```
procedure SockSend(const Line: RawByteString=''); overload;
```

*Simulate writeln() with a single line*

```
procedure SockSendFlush;
```

*Flush all pending data to be sent*

```
procedure Write(const Data: RawByteString);
```

*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 CR/CRLF is appened to the block

**THttpSocketCompressRec = record**

*Used to maintain a list of known compression algorithms*

**Func: THttpSocketCompress;**

*The function handling compression and decompression*

**Name: AnsiString;**

*The compression name, as in ACCEPT-ENCODING: header (gzip,deflate,synlz)*

**THttpSocket = class(TCrtSocket)**

*Parent of THttpClientSocket and THttpServerSocket classes*

- contain properties for implementing the HTTP/1.1 protocol
- 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: RawByteString;**

*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.

**ConnectionClose: boolean;**

*Same as HeaderValue('Connection')='close', but retrieved during Request*

**Content: RawByteString;**

*Will contain the data retrieved from the server, after the Request*

**ContentLength: integer;**

*Same as HeaderValue('Content-Length'), but retrieved during Request*

- is overridden with real Content length during HTTP body retrieval

**ContentType: RawByteString;**

*Same as HeaderValue('Content-Type'), but retrieved during Request*

**Headers: array of RawByteString;**

*Will contain the header lines after a Request - use HeaderValue() to get one*

**TCPPrefix: RawByteString;**

*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

**function HeaderAdd(const aValue: RawByteString): integer;**

*Add an header entry, returning the just entered entry index in Headers[]s*

**function HeaderGetText: RawByteString; virtual;**

*Get all Header values at once, as CRLF delimited text*

```
function HeaderValue(aName: RawByteString): RawByteString;  
    HeaderValue('Content-Type')='text/html', e.g.
```

```
function RegisterCompress(aFunction: THttpSocketCompress): 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  
    - the first registered algorithm will be the preferred one for compression
```

```
procedure HeaderSetText(const aText: RawByteString);  
    Set all Header values at once, from CRLF delimited text
```

```
THttpServerSocket = class(THttpSocket)  
    HTTP/1.1 server class used by THttpServer main server Thread
```

```
    KeepAliveClient: boolean;  
    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")
```

```
    Method: RawByteString;  
    Contains the method ('GET','POST'.. e.g.) after GetRequest()
```

```
    URL: RawByteString;  
    Contains the URL ('/' e.g.) after GetRequest()
```

```
constructor Create(aServer: THttpServer);  
    Create the socket according to a server  
    - will register the THttpSocketCompress functions from the server
```

```
function GetRequest(withBody: boolean=true): boolean;  
    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 ContentLength<>0)  
    - return false if the socket was not connected any more, or if any exception occurred during the  
    process
```

```
function HeaderGetText: RawByteString; override;  
    Get all Header values at once, as CRLF delimited text  
    - this overridden version will add the 'RemoteIP: 1.2.3.4' header
```

```
procedure InitRequest(aClientSock: TSocket);  
    Main object function called after aClientSock := Accept + Create:  
    - get initialize the socket with the supplied accepted socket  
    - caller will then use the GetRequest method below to get the request
```

## **THttpClientSocket = class(THttpSocket)**

*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

**UserAgent: RawByteString;**

*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

**function Delete(const url: RawByteString; KeepAlive: cardinal=0; const header: RawByteString=''): integer;**

*After an Open(server,port), return 200,202,204 if OK, http status error otherwise*

**function Get(const url: RawByteString; KeepAlive: cardinal=0; const header: RawByteString=''): integer;**

*After an Open(server,port), return 200 if OK, http status error otherwise - get the page data in Content*

**function Head(const url: RawByteString; KeepAlive: cardinal=0; const header: RawByteString=''): 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: RawByteString; KeepAlive: cardinal=0; const header: RawByteString=''): integer;**

*After an Open(server,port), return 200,201,204 if OK, http status error otherwise*

**function Put(const url, Data, DataType: RawByteString; KeepAlive: cardinal=0; const header: RawByteString=''): integer;**

*After an Open(server,port), return 200,201,204 if OK, http status error otherwise*

**function Request(const url, method: RawByteString; KeepAlive: cardinal; const header, Data, DataType: RawByteString; retry: boolean): integer;**

*Low-level HTTP/1.1 request*

- call by all REST methods above
- 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

## **THttpRequest = class(TThread)**

*HTTP response Thread*

- Execute procedure get the request and calculate the answer
- you don't have to overload the protected THttpRequest Execute method: override THttpRequest.Request() function or, if you need a lower-level access (change the protocol, e.g.) THttpRequest.Process() method itself

**constructor** Create(aServerSock: THttpRequestSocket; aServer: THttpRequest); overload;

- Initialize the response thread for the corresponding incoming socket*
- this version will handle KeepAlive, for such an incoming request

**constructor** Create(aSock: TSocket; aServer: THttpRequest); overload;

- Initialize the response thread for the corresponding incoming socket*
- this version will get the request directly from an incoming socket

**destructor** Destroy; override;

- Release used socket and memory*

## **TSynThreadPool = object(TObject)**

*A simple Thread Pool, used for fast handling HTTP requests*

- will handle multi-connection with less overhead than creating a thread for each incoming request
- this Thread Pool is implemented over I/O Completion Ports, which is a faster method than keeping a TThread list, and resume them on request: I/O completion just has the thread running while there is pending tasks, with no pause/resume

**function** Initialize(WorkerFunc: TThreadFunc; NumberOfThreads: Integer=32): Boolean;

- Initialize a thread pool with the supplied number of threads*
- WorkerFunc should be e.g. private THttpRequestWorkerFunction for handling a THttpRequest request
  - up to 64 threads can be associated to a Thread Pool

**function** Shutdown: Boolean;

- Shut down the Thread pool, releasing all associated threads*

## **TSynThreadPoolTHttpRequest = object(TSynThreadPool)**

*A simple Thread Pool, used for fast handling HTTP requests of a THttpRequest*

- will create a THttpRequest response thread, if the incoming request is identified as HTTP/1.1 keep alive

**function** Initialize(NumberOfThreads: Integer=32): Boolean; reintroduce;

- Initialize a thread pool with the supplied number of threads*
- will use private THttpRequestWorkerFunction as WorkerFunc
  - up to 64 threads can be associated to a Thread Pool

**function** Push(aServer: THttpServer; aClientSock: TSocket): Boolean;

*Add an incoming HTTP request to the Thread Pool*

**THttpServerGeneric = class(TThread)**

*Generic HTTP server*

*Used for DI-2.1.1.2.4 (page 830).*

**function** Request(const InURL, InMethod, InHeaders, InContent, InContentType: RawByteString; out OutContent, OutContentType, OutCustomHeader: RawByteString): 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 will handle Content-Type/Location
- if OutContentType is HTTP\_RESP\_STATICFILE (i.e. '!STATICFILE'), then OutContent is the UTF-8 file name of a file which must be sent to the client via http.sys (much faster than manual buffering/sending)
- default implementation is to call the OnRequest event (if existing)
- warning: this process must be thread-safe (can be called by several threads simultaneously)

*Used for DI-2.1.1.2.4 (page 830).*

**procedure** RegisterCompress(aFunction: THttpSocketCompress); **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
- the first registered algorithm will be the preferred one for compression

**property** OnHttpThreadStart: TNotifyThreadEvent **read** fOnHttpThreadStart **write** fOnHttpThreadStart;

*Event handler called when the Thread is just initiated*

- called in the thread context at first place in THttpServerGeneric.Execute

**property** OnHttpThreadTerminate: TNotifyEvent **read** fOnHttpThreadTerminate **write** fOnHttpThreadTerminate;

*Event handler called when the Thread is terminating, in the 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:

```
procedure TMyServer.OnHttpThreadTerminate(Sender: TObject);  
begin // TSQLDBConnectionPropertiesThreadSafe  
    fMyConnectionProps.EndCurrentThread;  
end;
```

**property** OnRequest: TOnHttpRequest read fOnRequest write fOnRequest;

*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 830).*

**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.

*Used for DI-2.1.1.2.4 (page 830).*

**constructor** Create(CreateSuspended: Boolean);

*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 TRUE

- if you will call AddUrl() methods later, set CreateSuspended to FALSE, then call explicitly the Resume method, after all AddUrl() calls, in order to start the server

*Used for DI-2.1.1.2.4 (page 830).*

**destructor** Destroy; override;

*Release all associated memory and handles*

**function** AddUrl(const aRoot, aPort: RawByteString; Https: boolean=false; const aDomainName: RawByteString='\*'): 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 been called with CreateSuspended = TRUE and then call the Resume method after all Url have been added



```
class function AddUrlAuthorize(const aRoot, aPort: RawByteString; Https: boolean=false; const aDomainName: RawByteString='*'; 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

```
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 TSQLite3HttpServer.Create() constructor)
- maximum value is 256 - higher should not be worth it

```
procedure RegisterCompress(aFunction: THttpSocketCompress); override;
```

*Will register a compression algorithm*

- overridden method which will handle any cloned instances

```
THttpServer = class(THttpServerGeneric)
```

*Main HTTP server Thread*

- bind to a port and listen to incoming requests
- assign this requests to THttpServerResp threads
- 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
- don't forget to use Free procedure when you are finished

*Used for DI-2.1.1.2.4 (page 830).*

```
ServerConnectionCount: cardinal;
```

*Will contain the total number of connection to the server*

- it's the global count since the server started

```
ServerKeepAliveTimeOut: cardinal;
```

*Time, in milliseconds, for the HTTP.1/1 connections to be kept alive; default is 3000 ms*



**Sock: TCrtSocket;**

*Contains the main server 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

**TCPPrefix: RawByteString;**

*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

**constructor Create(const aPort: AnsiString ; ServerThreadPoolCount: integer=32);**

*Create a Server Thread, binded and listening on a port*

- this constructor will raise a EHttpServer exception if binding failed
- 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 64)

*Used for DI-2.1.1.2.4 (page 830).*

**destructor Destroy; override;**

*Release all memory and handlers*

#### Types implemented in the SynCrtSock unit:

**PPointer = ^Pointer;**

*Not defined in Delphi 5 or older*

**PPtrInt = ^PtrInt;**

*FPC 64 compatibility pointer type*

**PtrInt = integer;**

*FPC 64 compatibility integer type*

**RawByteString = AnsiString;**

*Define RawByteString, as it does exist in Delphi 2009 and up*

- to be used for byte storage into an AnsiString

**TCrtSocketLayer = ( cs1TCP, cs1UDP, cs1UNIX );**

*The available available network transport layer*

- either TCP/IP, UDP/IP or Unix sockets

**THttpSocketCompress = function(var Data: RawByteString; Compress: boolean): RawByteString;**

*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

- type is a generic AnsiString, which should be in practice a RawByteString or a RawByteString

**TNotifyThreadEvent = procedure(Sender: TThread) of object;**

*Event prototype used e.g. by THttpServerGeneric.OnHttpThreadStart*

**TOnHttpRequest = function( const InURL, InMethod, InHeaders, InContent, InContentType: RawByteString; out OutContent, OutContentType, OutCustomHeader: RawByteString): cardinal of object;**

*Event handler used by THttpServerGeneric.OnRequest property*

- InURL/InMethod/InHeaders/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 will handle Content-Type/Location
- if OutContentType is HTTP\_RESP\_STATICFILE (i.e. '!STATICFILE'), then OutContent is the UTF-8 file name of a file which must be sent to the client via http.sys (much faster than manual buffering/sending)

#### Constants implemented in the *SynCrtSock* unit:

**HTTP\_RESP\_STATICFILE = '!STATICFILE';**

*Used by THttpApiServer.Request for http.sys to send a static file*

- the OutCustomHeader should contain the proper 'Content-type: ....' corresponding to the file

#### Functions or procedures implemented in the *SynCrtSock* unit:

| Functions or procedures | Description                                                                                                    | Page |
|-------------------------|----------------------------------------------------------------------------------------------------------------|------|
| Base64Decode            | Base64 decoding of a string                                                                                    | 383  |
| Base64Encode            | Base64 encoding of a string                                                                                    | 383  |
| GetRemoteMacAddress     | Remotely get the MAC address of a computer, from its IP Address                                                | 383  |
| HttpGet                 | Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method                                 | 383  |
| HttpPost                | Send some data to a remote web server, using the HTTP/1.1 protocol and POST method                             | 383  |
| Open                    | Create a TCrtSocket, returning nil on error (usefull to easily catch socket error exception ECrtSocket)        | 383  |
| OpenHttp                | Create a THttpClientSocket, returning nil on error (usefull to easily catch socket error exception ECrtSocket) | 383  |
| ResolveName             | Retrieve the IP adress from a computer name                                                                    | 383  |
| SendEmail               | Send an email using the SMTP protocol                                                                          | 383  |
| StatusCodeToReason      | Retrieve the HTTP reason text from a code                                                                      | 383  |

**function** Base64Decode(const s: RawByteString): RawByteString;

*Base64 decoding of a string*

**function** Base64Encode(const s: RawByteString): RawByteString;

*Base64 encoding of a string*

**function** GetRemoteMacAddress(const IP: AnsiString): RawByteString;

*Remotly 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** HttpGet(const server, port: AnsiString; const url: RawByteString): RawByteString;

*Retrieve the content of a web page, using the HTTP/1.1 protocol and GET method*

**function** HttpPost(const server, port: AnsiString; const url, Data, DataType: RawByteString): boolean;

*Send some data to a remote web server, using the HTTP/1.1 protocol and POST method*

**function** Open(const aServer, aPort: AnsiString): TCrtSocket;

*Create a TCrtSocket, returning nil on error (usefull to easily catch socket error exception ECrtSocket)*

**function** OpenHttp(const aServer, aPort: AnsiString): THttpClientSocket;

*Create a THttpClientSocket, returning nil on error (usefull to easily catch socket error exception ECrtSocket)*

**function** ResolveName(const Name: AnsiString): AnsiString;

*Retrieve the IP adress from a computer name*

**function** SendEmail(const Server: AnsiString; const From, CSVDest, Subject, Text: RawByteString; const Headers: RawByteString=''; const User: RawByteString=''; const Pass: RawByteString=''; const Port: AnsiString='25'): boolean;

*Send an email using the SMTP protocol*

- retry true on success

**function** StatusCodeToReason(Code: integer): RawByteString;

*Retrieve the HTTP reason text from a code*

- e.g. StatusCodeToReason(200)='OK'

#### 1.4.7.4. SynCrypto unit

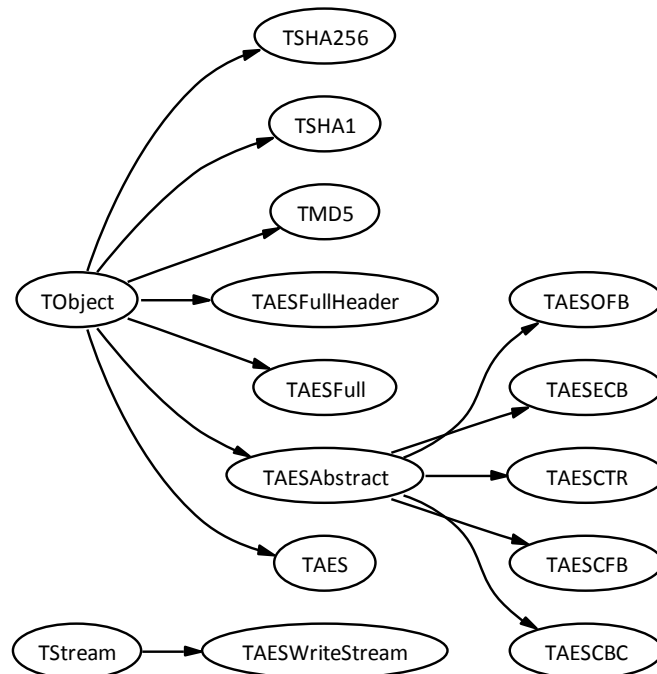
*Purpose:* Fast cryptographic routines (hashing and cypher)

- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms
- optimized for speed (tuned assembler and VIA PADLOCK optional support)
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**Units used in the SynCrypto unit:**

| Unit Name | Description | Page |
|-----------|-------------|------|
|-----------|-------------|------|

| Unit Name         | Description                                                                                                                                                                  | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SynCrypto class hierarchy*

#### Objects implemented in the *SynCrypto* unit:

| Objects         | Description                                                           | Page |
|-----------------|-----------------------------------------------------------------------|------|
| TAES            | Handle AES cypher/uncypher                                            | 385  |
| TAESAbstract    | Handle AES cypher/uncypher with chaining                              | 386  |
| TAESCBC         | Handle AES cypher/uncypher with Cipher-block chaining (CBC)           | 387  |
| TAESCFB         | Handle AES cypher/uncypher with Cipher feedback (CFB)                 | 387  |
| TAESCTR         | Handle AES cypher/uncypher with Counter mode (CTR)                    | 387  |
| TAEECB          | Handle AES cypher/uncypher without chaining (ECB)                     | 386  |
| TAESFull        | AES and XOR encryption object for easy direct memory or stream access | 389  |
| TAESFullHeader  | Internal header for storing our AES data with salt and CRC            | 388  |
| TAESOFB         | Handle AES cypher/uncypher with Output feedback (OFB)                 | 387  |
| TAESWriteStream | AES encryption stream                                                 | 389  |
| TMD5            | Handle MD5 hashing                                                    | 388  |

| Objects | Description           | Page |
|---------|-----------------------|------|
| TSHA1   | Handle SHA1 hashing   | 387  |
| TSHA256 | Handle SHA256 hashing | 388  |

## TAES = **object**(TObject)

*Handle AES cypher/uncypher*

- this is the default Electronic codebook (ECB) mode

**Initialized:** boolean;

*True if the context was initialized*

**function** DecryptInit(**const** Key; KeySize: cardinal): boolean;

*Initialize AES contexts for uncypher*

**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 class

- KeySize is in bits, i.e. 128,192,256

**procedure** Decrypt(**const** BI: TAESBlock; **var** B0: TAESBlock); overload;

*Decrypt an AES data block into another data block*

**procedure** Decrypt(**var** B: TAESBlock); overload;

*Decrypt an AES 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** 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*

- only used with Padlock

```
procedure Encrypt(const BI: TAESBlock; var B0: TAESBlock); overload;
```

*Encrypt an AES data block into another data block*

```
procedure Encrypt(var B: TAESBlock); overload;
```

*Encrypt an AES data block*

```
TAESAbstract = class(TObject)
```

*Handle AES cypher/uncypher with chaining*

- use any of the inherited implementation, accorsponding to the chaining mode required -  
 TAESECB, TAESCBC, TAESCFB, TAESOFB and TAESCTR classes to handle in ECB, CBC, CFB, OFB and  
 CTR mode (including PKCS7 padding)

```
constructor Create(const aKey; aKeySize: cardinal; const aIV: TAESBlock);  
virtual;
```

*Initialize AES contexts for cypher*

- first method to call before using this class  
 - KeySize is in bits, i.e. 128,192,256  
 - IV is the Initialization Vector

```
function DecryptPKCS7(const Input: RawByteString): RawByteString;
```

*Decrypt a memory buffer using a PKCS7 padding pattern*

- PKCS7 is described in RFC 5652 - it will trim up to 16 bytes from the input buffer

```
function EncryptPKCS7(const Input: RawByteString): RawByteString;
```

*Encrypt a memory buffer using a PKCS7 padding pattern*

- PKCS7 is described in RFC 5652 - it will add up to 16 bytes to the input buffer

```
procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); virtual;
```

*Perform the AES un-cypher in the corresponding mode*

- this abstract method will set CV from AES.Context, and fln/fOut from BufIn/BufOut

```
procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); virtual;
```

*Perform the AES cypher in the corresponding mode*

- this abstract method will set CV from AES.Context, and fln/fOut from BufIn/BufOut

```
property Key: TAESKey read fKey;
```

*Associated Key data*

```
property KeySize: cardinal read fKeySize;
```

*Associated Key Size, in bits (i.e. 128,192,256)*

```
TAESECB = class(TAESAbstract)
```

*Handle AES cypher/uncypher without chaining (ECB)*

- this mode is known to be less secure than the others  
 - IV value set on constructor is used to code the trailing bytes of the buffer (by a simple XOR)

```
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(TAESAbstract)
```

```
    Handle AES cypher/uncypher with Cipher-block chaining (CBC)
```

```
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
```

```
TAESCFB = class(TAESAbstract)
```

```
    Handle AES cypher/uncypher with Cipher feedback (CFB)
```

```
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(TAESAbstract)
```

```
    Handle AES cypher/uncypher with Output feedback (OFB)
```

```
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 cypher in the OFB mode
```

```
TAESCTR = class(TAESAbstract)
```

```
    Handle AES cypher/uncypher with Counter mode (CTR)
```

```
procedure Decrypt(BufIn, BufOut: pointer; Count: cardinal); override;  
    Perform the AES un-cypher in the CTR mode
```

```
procedure Encrypt(BufIn, BufOut: pointer; Count: cardinal); override;  
    Perform the AES cypher in the CTR mode
```

```
TSHA1 = object(TObject)
```

```
    Handle SHA1 hashing
```

```
procedure Final(out Digest: TSHA1Digest);  
    Finalize and compute the resulting SHA1 hash Digest of all data affected to Update() method
```

**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;

*Used by Update and Final initialize SHA1 context for hashing*

**procedure** Update(Buffer: pointer; Len: integer);

*Update the SHA1 context with some data*

TSHA256 = **object**(TObject)

*Handle SHA256 hashing*

**procedure** Final(**out** Digest: TSHA256Digest);

*Finalize and compute the resulting SHA256 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;

*Used by Update and Final initialize SHA256 context for hashing*

**procedure** Update(Buffer: pointer; Len: integer);

*Update the SHA256 context with some data*

TMD5 = **object**(TObject)

*Handle MD5 hashing*

**function** Final: TMD5Digest;

*Finalize and compute the resulting MD5 hash Digest of all data affected to Update() method*

**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; Len: cardinal);

*Update the MD5 context with some data*

TAESFullHeader = **object**(TObject)

*Internal header for storing our AES data with salt and CRC*

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*

**TAESFull = object(TObject)**

*AES and XOR encryption object for easy direct memory or stream access*

- calls internaly TAES objet 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 uncryptd 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 closing the outStream (some data may still be in the internal buffers)

#### Types implemented in the SynCrypto unit:

TAESBlock = **packed array**[0..AESBlockSize-1] of byte;

*128 bits memory block for AES data cypher/uncypher*

TAESKey = **packed array**[0..AESKeySize-1] of byte;

*256 bits memory block for maximum AES key storage*

TMD5Digest = **array**[0..15] of Byte;

*128 bits memory block for MD5 hash digest storage*

TSHA1Digest = **packed array**[0..19] of byte;

*160 bits memory block for SHA1 hash digest storage*

TSHA256Digest = **packed array**[0..31] of byte;

*256 bits memory block for SHA256 hash digest storage*

#### Constants implemented in the SynCrypto unit:

AESBlockSize = 16;

*Standard AES block size during cypher/uncypher*

AESContextSize = 278+sizeof(pointer);

*Hide all AES Context complex code*

AESKeySize = 256 div 8;

*Maximum AES key size*

SHAContextSize = 108;

*Hide all SHA Context complex code*

#### Functions or procedures implemented in the SynCrypto unit:

| Functions or procedures | Description | Page |
|-------------------------|-------------|------|
|-------------------------|-------------|------|

| Functions or procedures | Description                                                                   | Page |
|-------------------------|-------------------------------------------------------------------------------|------|
| Adler32Asm              | Fast Adler32 implementation                                                   | 392  |
| Adler32Pas              | Simple Adler32 implementation                                                 | 392  |
| Adler32SelfTest         | Self test of Adler32 routines                                                 | 392  |
| AES                     | Direct Encrypt/Decrypt of data using the TAES class                           | 392  |
| AES                     | Direct Encrypt/Decrypt of data using the TAES class                           | 392  |
| AES                     | Direct Encrypt/Decrypt of data using the TAES class                           | 392  |
| AES                     | Direct Encrypt/Decrypt of data using the TAES class                           | 392  |
| AESFull                 | AES and XOR encryption using the TAESFull format                              | 393  |
| AESFull                 | AES and XOR encryption using the TAESFull format                              | 393  |
| AESFullKeyOK            | AES and XOR decryption check using the TAESFull format                        | 393  |
| AESSelfTest             | Self test of AES routines                                                     | 393  |
| AESSHA256               | AES encryption using the TAES format with a supplied SHA256 password          | 393  |
| AESSHA256               | AES encryption using the TAES format with a supplied SHA256 password          | 393  |
| AESSHA256               | AES encryption using the TAES format with a supplied SHA256 password          | 393  |
| AESSHA256Full           | AES encryption using the TAESFull format with a supplied SHA256 password      | 393  |
| bswap160                | Little endian fast conversion                                                 | 393  |
| bswap256                | Little endian fast conversion                                                 | 393  |
| bswap32                 | Little endian fast conversion                                                 | 394  |
| htdigest                | Compute the HTDigest for a user and a realm, according to a supplied password | 394  |
| MD5                     | Direct MD5 hash calculation of some data (string-encoded)                     | 394  |
| MD5Buf                  | Direct MD5 hash calculation of some data                                      | 394  |
| MD5DigestsEqual         | Compare two supplied MD5 digests                                              | 394  |
| MD5DigestToString       | Compute the hexadecimal representation of a MD5 digest                        | 394  |
| MD5SelfTest             | Self test of MD5 routines                                                     | 394  |
| SHA1                    | Direct SHA1 hash calculation of some data (string-encoded)                    | 394  |

| Functions or procedures | Description                                                       | Page |
|-------------------------|-------------------------------------------------------------------|------|
| SHA1DigestToString      | Compute the hexadecimal representation of a SHA1 digest           | 394  |
| SHA1SelfTest            | Self test of SHA1 routines                                        | 394  |
| SHA256                  | Direct SHA256 hash calculation of some data (string-encoded)      | 394  |
| SHA256DigestToString    | Compute the hexadecimal representation of a SHA256 digest         | 394  |
| SHA256SelfTest          | Self test of SHA256 routines                                      | 394  |
| SHA256Weak              | Direct SHA256 hash calculation of some data (string-encoded)      | 394  |
| XorBlock                |                                                                   | 395  |
| XorConst                | Fast XOR Cypher changing by Count value                           | 395  |
| XorOffset               | Fast and simple XOR Cypher using Index (=Position in Dest Stream) | 395  |

**function** Adler32Asm(Adler: cardinal; p: pointer; Count: Integer): cardinal;

*Fast Adler32 implementation*

- 16-bytes-chunck 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*

**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** AES(const Key; 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 crypted by AES, but with XOR

**function** AES(const Key; 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 crypted 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

```
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 begining of buff contains true AESFull encrypted data with this Key
- if not KeySize in [128,192,256] -> use fast and efficient Xor Cypher

```
function AESSelfTest: boolean;
```

*Self test of AES routines*

```
procedure AESSHA256(bIn, bOut: pointer; Len: integer; const Password:
RawByteString; Encrypt: boolean); overload;
```

*AES encryption using the TAES format with a supplied SHA256 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 SHA256 password*

- last bytes (not part of 16 bytes blocks) are not crypted by AES, but with XOR

```
procedure AESSHA256(Buffer: pointer; Len: integer; const Password: RawByteString;
Encrypt: boolean); overload;
```

*AES encryption using the TAES format with a supplied SHA256 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 SHA256 password*

- outStream will be larger/smaller than Len: this is a full AES version with a trimming  
TAESFullHeader at the beginning

```
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 bswap32(a: cardinal): cardinal;
```

*Little endian fast conversion*

- 32 bits = 1 integer
- use fast bswap asm in x86/x64 mode

```
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 MD5DigestsEqual(const A, B: TMD5Digest): Boolean;
```

*Compare two supplied MD5 digests*

```
function MD5DigestToString(const D: TMD5Digest): RawUTF8;
```

*Compute the hexadecimal representation of a MD5 digest*

```
function MD5SelfTest: boolean;
```

*Self test of MD5 routines*

```
function SHA1(const s: RawByteString): RawUTF8;
```

*Direct SHA1 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 SHA1 digest*

```
function SHA1SelfTest: boolean;
```

*Self test of SHA1 routines*

```
function SHA256(const s: RawByteString): RawUTF8;
```

*Direct SHA256 hash calculation of some data (string-encoded)*

- result is returned in hexadecimal format

```
function SHA256DigestToString(const D: TSHA256Digest): RawUTF8;
```

*Compute the hexadecimal representation of a SHA256 digest*

```
function SHA256SelfTest: boolean;
```

*Self test of SHA256 routines*

```
procedure SHA256Weak(const s: RawByteString; out Digest: TSHA256Digest);  
overload;
```

*Direct SHA256 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.

**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** 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: pByte; 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

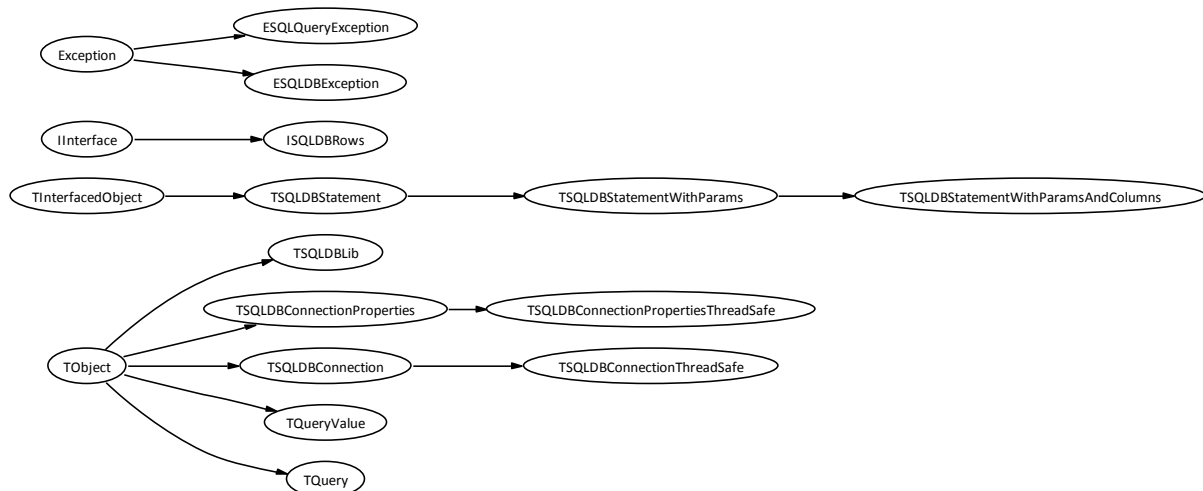
#### 1.4.7.5. SynDB unit

*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 1.17

**Units used in the SynDB unit:**

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SynDB class hierarchy*

**Objects implemented in the SynDB unit:**

| Objects | Description | Page |
|---------|-------------|------|
|---------|-------------|------|

| Objects                              | Description                                                                             | Page |
|--------------------------------------|-----------------------------------------------------------------------------------------|------|
| ESQLDBException                      | Generic Exception type, as used by the SynDB unit                                       | 396  |
| ESQLQueryException                   | Generic Exception type raised by the TQuery class                                       | 422  |
| ISQLDBRows                           | Generic interface to access a SQL query result rows                                     | 398  |
| TQuery                               | Class mapping VCL DB TQuery for direct database process                                 | 424  |
| TQueryValue                          | Pseudo-class handling a TQuery bound parameter or column value                          | 422  |
| TSQLDBColumnDefine                   | Used to define a field/column layout in a table schema                                  | 396  |
| TSQLDBColumnProperty                 | Used to define a field/column layout                                                    | 397  |
| TSQLDBConnection                     | Abstract connection created from TSQLDBConnectionProperties                             | 406  |
| TSQLDBConnectionProperties           | Abstract class used to set Database-related properties                                  | 401  |
| TSQLDBConnectionPropertiesThreadSafe | Connection properties which will implement an internal Thread-Safe connection pool      | 416  |
| TSQLDBConnectionThreadSafe           | Abstract connection created from TSQLDBConnectionProperties                             | 416  |
| TSQLDBLib                            | Access to a native library                                                              | 421  |
| TSQLDBParam                          | A structure used to store a standard binding parameter                                  | 417  |
| TSQLDBStatement                      | Generic abstract class to implement a prepared SQL query                                | 407  |
| TSQLDBStatementWithParams            | Generic abstract class handling prepared statements with binding                        | 418  |
| TSQLDBStatementWithParamsAndColumns  | Generic abstract class handling prepared statements with binding and column description | 421  |

**ESQLDBException = class(Exception)**

*Generic Exception type, as used by the SynDB unit*

**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;**

*Should be TRUE if the 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

**ColumnType: TSQldbFieldType;**

*The Column type,*

- 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

**TSQldbColumnProperty = packed record**

*Used to define a field/column layout*

- for TSQldbConnectionProperties.SQLCreate to describe the table

- for TOleDBStatement.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

**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

**ColumnUnique: boolean;**

*Set if the Column shall have unique value (add the corresponding constraint)*

**ColumnValueDBCharSet:** integer;

*For SQLT\_STR/SQLT\_CLOB: used to store the ftUTF8 column char set encoding*  
 - for SynDBOracle, equals to the OCI char set

**ColumnValueDBForm:** byte;

*For SynDBOracle: used to store the ftUTF8 column encoding, i.e. for SQLT\_CLOB, equals either to SQLCS\_NCHAR or SQLCS\_IMPLICIT*

**ColumnValueDBSize:** cardinal;

*For TSQLDBOracleStatement: used to store one value 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 and SQLT\_BLOB  
 - 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

**ColumnValueInlined:** boolean;

*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  
 - for TSQLDBODBCStatement: FALSE if bigger than 255 WideChar (ftUTF8) or 255 bytes (ftBlob)

**ColumnValueState:** TSQLDBStatementGetCol;

*For SynDBODBC: state of the latest SQLGetData() call*

**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(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;

*Return a CoLumn as a blob value of the current Row, first Col is 0*

**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** 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** 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(**const** ColName: RawUTF8): TTimeLog; overload;  
*Return a coLumn date and time value of the current Row, from a supplied coLumn name*

**function** ColumnTimeStamp(Col: integer): TTimeLog; overload;  
*Return a coLumn date and time value of the current Row, first Col is 0*

**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(const ColName: RawUTF8): Variant; overload;

*Return a Column as a variant, from a supplied column name*

**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** FetchAllAsJSON(Expanded: boolean; ReturnedRowCount: PPtrInt=nil): RawUTF8;

- 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 SQLite3 unit

**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** Step(SeekFirst: boolean=false): boolean;

*After a prepared statement has been prepared returning a TSQLDBRows 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:

```
var Customer: Variant;
begin
  with Props.Execute( 'select * from Sales.Customer where AccountNumber like ?',
    ['AW000001%'],@Customer) do
    while Step do // loop through all matching data rows
      assert(Copy(Customer.AccountNumber,1,8)='AW000001');
    end;
```

```
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:

```
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']));  
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

```
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

```
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*

```
function AdaptSQLLimitForEngineList(var SQL: RawUTF8; LimitRowCount,  
AfterSelectPos, WhereClausePos, LimitPos: integer): boolean; virtual;
```

*Adapt the LIMIT # clause in the SQL SELECT statement to a syntax matching the underlying DBMS*

- e.g. TSQLRestServerStaticExternal.AdaptSQLForEngineList() calls this to let TSQLRestServer.URI by-pass virtual table mechanism
- integer parameters state how the SQL statement has been analysed

```
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 ): ISQLDBRows;
```

*Execute a SQL query, returning a statement interface instance to retrieve the result rows*  
 - will call NewThreadSafeStatement method to retrieve a thread-safe statement instance, then run the corresponding Execute() method  
 - 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']));  
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  
    while Step do  
      writeln(R.FirstName, ' ', DateToStr(R.BirthDate));  
end;
```

```
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 (or 0 if the DB driver do not give access to 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]'

```
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 NewConnection: TSQLDBConnection; virtual; abstract;
```

*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 aSQL: RawUTF8; ExpectResults: Boolean): TSQLDBStatement; overload;
```

*Create a new thread-safe statement from an internal cache (if any)*

- will call ThreadSafeConnection.NewStatementPrepared
- this method should return nil in case of error, or a prepared statement instance in case of success

```
function NewThreadSafeStatementPrepared(SQLFormat: PUTF8Char; const Args: array of const; ExpectResults: Boolean): TSQLDBStatement; overload;
```

*Create a new thread-safe statement from an internal cache (if any)*

- this method will call the 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

```
function SQLAddColumn(const aTableName: RawUTF8; const aField: TSQLDBColumnProperty): RawUTF8; virtual;
```

*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 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 SQLAddIndex(const aTableName: RawUTF8; const aFieldNames: array of RawUTF8; aUnique: boolean; const aIndexName: RawUTF8=''): RawUTF8; virtual;
```

*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])'

```
function SQLCreate(const aTableName: RawUTF8; const aFields: TSQLDBColumnPropertyDynArray): RawUTF8; virtual;
```

*Used to create a Table*

- should return the SQL "CREATE" statement needed to create a table with the specified field/column names and types
- a "ID Int64 PRIMARY KEY" column is always added at first position, and will expect the ORM to create an 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** SQLIso8601ToDate(**const** Iso8601: RawUTF8): RawUTF8; **virtual**;

*Convert an ISO-8601 encoded time and date into a date appropriate to be pasted in the SQL request*

- this default implementation will return the quoted ISO-8601 value, i.e. '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** SQLSelectAll(**const** aTableName: RawUTF8; **const** aFields: TSQLDBColumnDefineDynArray; aExcludeTypes: TSQLDBFieldTypes): RawUTF8; **virtual**;

*Used to compute a SELECT statement for the given fields*

- 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** 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

**procedure** GetFieldDefinitions(**const** aTableName: RawUTF8; **var** 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; **var** 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** GetTableNames(**var** Tables: TRawUTF8DynArray); **virtual**;

*Get all table names*

- this default implementation will use protected SQLGetTableNames virtual method to retrieve the table names



**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

**property** BatchSendingAbilities: TSQLDBStatementCRUDs **read** fBatchSendingAbilities;

*The abilities of the database for batch sending*  
- e.g. Oracle will handle array binds, or MS SQL bulk insert

**property** DatabaseName: RawUTF8 **read** fDatabaseName;

*The associated database name, as specified at creation*

**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** 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** ForeignKeysData: RawByteString **read** GetForeignKeysData **write** SetForeignKeysData;

*Can be used to store the fForeignKeys[] data in an external BLOB*  
- since GetForeignKeys is somewhat slow, could save a lot of time

**property** MainConnection: TSQLDBConnection **read** GetMainConnection;

*Return a shared connection, corresponding to the given*  
- call the ThreadSafeConnection method instead e.g. for multi-thread access, or NewThreadSafeStatement for direct retrieval of a new statement

**property** PassWord: RawUTF8 **read** fPassWord;

*The associated User Password, as specified at creation*

**property** ServerName: RawUTF8 **read** fServerName;

*The associated server name, as specified at creation*

**property** UserID: RawUTF8 **read** fUserID;

*The associated User Identifier, as specified at creation*

**property** VariantStringAsWideString: 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-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 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

**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): TSQLDBStatement; **virtual**;

*Initialize a new SQL query statement for the given connection*

- this default implementation will call the NewStatement method
- but children may override this method to handle statement caching
- this method should return nil in case of error, or a prepared statement instance in case of success

**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**; **abstract**;

*Connect to the specified database*

- should raise an Exception on error

**procedure** Disconnect; **virtual**; **abstract**;

*Stop connection to the specified database*  
- should raise an Exception on error

**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** InfoMessage: string **read** fInfoMessage;

*Some information message, as retrieved during execution*

**property** InTransaction: boolean **read** GetInTransaction;

*TRUE if StartTransaction has been called*  
- check if TransactionCount>0

**property** LastErrorMessage: string **read** fErrorMessage;

*Some information message, as retrieved during execution*

**property** Properties: TSQLDBConnectionProperties **read** fProperties;

*The associated database properties*

**property** ServerTimeStamp: TTimeLog **read** GetServerTimeStamp;

*The current Date and Time, as retrieved from the server*  
- this property will return the timestamp in TTimeLog / Iso8601 / Int64 after correction from the Server returned time-stamp (if any)  
- default implementation will return the executable time, i.e. Iso8601Now

**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** 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** 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(Col: integer): double; overload; **virtual**; **abstract**;  
*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** 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)

**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:
TSQLDBColumnPropertyDynArray): RawUTF8; virtual;
```

*Compute the SQL INSERT statement corresponding to this columns row*

- and populate the Fields[] array with columns information (type and name)
- the SQL statement is prepared with bound parameters, e.g.  
insert into TableName (Col1,Col2) values (?,N)

- used e.g. to convert some data on the fly from one database to another

```
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; 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 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 Iso8601/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 Iso8601/Int64 time stamp from a TDateTime or text

```
function ColumnToVarData(Col: Integer; var Value: TVarData; var Temp:
RawByteString): TSQLDBFieldType; virtual;
```

*Return a Column as a TVarData value, first Col is 0*

- TVarData returned types are varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0]); ftDate is returned as varString, as in TSQLDBStatement.ColumnsToJSON
- so this Value should not be used typecasted to a Variant
- the specified Temp variable will be used for temporary storage of varString/varAny values
- this default implementation will call corresponding Column\*() method

```
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(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; **virtual; abstract;**

*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, 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 ""\uFFFF0base64encodedbinary"" 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 SQLite3 unit

**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;

- 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 ""\uFFFF0base64encodedbinary"" format and contains true BLOB data
- similar to corresponding TSQLRequest.Execute method in SQLite3 unit
- returns the number of row data returned (excluding field names)
- warning: TSQLRestServerStaticExternal.EngineRetrieve in SQLite3DB expects the Expanded=true format to return '[{...}]'#10

**function** ParamToVariant(Param: Integer; var Value: Variant; CheckIsOutParameter: boolean=true): TSQldbFieldType; **virtual**;

- Retrieve the parameter content, after SQL execution*
- the leftmost SQL parameter has an index of 1
- to be used e.g. with stored procedures
- 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** 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 SQLite3DB unit):

```
var Query: TSQldbStatement;
begin
  Query := Props.NewThreadSafeStatementPrepared('select AccountNumber from Sales.Customer
where AccountNumber like ?', ['AW000001%'],true);
  if Query<>nil then
    try
      assert(SameTextU(Query.ColumnName(0),'AccountNumber'));
      while Query.Step do // loop through all matching data rows
        assert(Copy(Query.ColumnUTF8(0),1,8)='AW000001');
      finally
        Query.Free;
      end;
    end;
  end;
```

**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(Param: Integer; Value: Int64; IO: TSQLDBParamInOutType=paramIn);  
overload; virtual; abstract;
```

*Bind an integer value to a parameter*

- the leftmost SQL parameter has an index of 1

```
procedure Bind(const Params: TVarDataDynArray; IO:  
TSQLDBParamInOutType=paramIn); overload; virtual;
```

*Bind an array of TVarData values*

- TVarData handled types are varNull, varInt64, varDouble, varDate, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- so this Param should not be used typecasted from a Variant
- this default implementation will call corresponding Bind\*() method

```
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
- this default implementation will call corresponding Bind\*() method

```
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
- values are stored as in SQL (i.e. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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
- values are stored as in SQL (i.e. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind



```
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', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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
- values are stored as in SQL (i.e. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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 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: TSQldbColumnPropertyDynArray; 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); virtual;  
abstract;
```

*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; 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 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 ""\uFFFF0base64encodedbinary" format and contains true BLOB data

```
procedure Execute(SQLFormat: PUTF8Char; 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 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 ExecutePrepared; virtual; abstract;
```

*Execute a prepared SQL statement*

- parameters marked as ? should have been already bound with Bind\*() functions
- should raise an Exception on any error

```
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 if it was not already connected

**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

**property** Connection: TSQLDBConnection **read** fConnection;

*The associated database connection*

**property** CurrentRow: Integer **read** fCurrentRow;

*The current row after Execute call, corresponding to Column\*() methods*

- contains 0 in case of no (more) available data, or a number >=1

**property** SQL: RawUTF8 **read** fSQL;

*The prepared SQL statement, as supplied to Prepare() method*

**property** SQLWithInlinedParams: RawUTF8 **read** GetSQLWithInlinedParams;

*The prepared SQL statement, with all '?' changed into the supplied parameter values*

**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

**property** UpdateCount: Integer **read** GetUpdateCount;

*Gets a number of updates made by latest executed statement*

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 be still be used on the background, 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:

```
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, SQLite3DB unit will call this method for every terminating thread created for TSQLRestServerNamedPipeResponse or TSQLite3HttpServer multi-thread process

**TSQLDBParam = packed record**

*A structure used to store a standard binding parameter*

- you can use your own internal representation of parameters (TOleDBStatement use its own TOleDBStatementParam 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)

**VArray: TRawUTF8DynArray;**

*Storage used for array bind values*

- number of items in array is stored in VInt64
- values are stored as in SQL (i.e. number, 'quoted string', 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

- values are stored as in SQL (i.e. number, 'quoted string', null)

- 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

- by now, only SynDBOracle unit implements an array bind

**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

- values are stored as in SQL (i.e. number, 'quoted string', null)

- this default implementation will call BindArray() after conversion into RawUTF8 items, stored in TSQLDBParam.VArray

- by now, only SynDBOracle unit implements an array bind

```
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 in SQL (i.e. number, 'quoted string', null)
- this default implementation will raise an exception if the engine does not support array binding
- by now, only SynDBOracle unit implements an array bind

```
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', null)
- by now, only SynDBOracle unit implements an array bind

```
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
- values are stored as in SQL (i.e. number, 'quoted string', null)
- 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
- by now, only SynDBOracle unit implements an array bind

```
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. number, 'quoted string', null)
- 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
- by now, only SynDBOracle unit implements an array bind

```
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 BindNull(Param: Integer; IO: TSQLDBParamInOutType=paramIn); override;
```

*Bind a NULL value to a parameter*

- the leftmost SQL parameter has an index of 1
- raise an Exception 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

```
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** 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

**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) - this implementation will store fColumns[Col].ColumnValueDBSize if ColumnValueInlined=true

**property** Columns: TSQLDBColumnPropertyDynArray **read** fColumns;

*Direct access to the columns description*

- gives more details than the default ColumnType() function

**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*

```
ESQLQueryException = class(Exception)
```

*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 PQueryValue
- 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 CLea + 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;

*How to use this parameter on 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 := '123420020100000430015';
  Q.Open;
  Q.First; // optional
  while not Q.Eof do begin
    assert(Q.FieldByName('id_detail').AsString='123420020100000430015');
    Q.Next;
  end;
  Q.Close; // optional
finally
  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 QueryToDataSet() 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** 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

```
procedure First;
```

*After a successful Open, will get the first row of results*

```
procedure Next;
```

*After successful Open and First, go to 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** 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*

#### 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

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

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*

TSQLDBDefinition =

( dUnknown, dDefault, dOracle, dMSSQL, dJet, dMySQL, dSQLite );

*The known database definitions*

- will be used e.g. for TSQLDBConnectionProperties.SQLFieldCreate(), or for ODBC definition

```
TSQLDBFieldType =  
( ftUnknown, ftNull, ftInt64, ftDouble, ftCurrency, ftDate, ftUTF8, ftBlob );
```

*The handled field/parameter/column types by this unit*

- this will map low-level database-level access types, not high-level Delphi types as TSQLFieldType defined in SQLite3Commons
- for instance, it 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](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

```
TSQLDBFieldTypeDefinition = array[ftNull..ftBlob] of RawUTF8;
```

*An array of RawUTF8, for each existing column type*

- used e.g. by SQLCreate method
- the RowID definition will expect the ORM to create a unique identifier, 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>
- 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 a unique identifier, 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>

```
TSQLDBFieldTypeDynArray = array of TSQLDBFieldType;
```

*Used to specify an array of field/parameter/column types*

```
TSQLDBFieldTypes = set of TSQLDBFieldType;
```

*Used to specify a set of field/parameter/column types*

```
TSQLDBParamDynArray = array of TSQLDBParam;
```

*Dynamic array used to store standard binding parameters*

- used e.g. by TSQLDBStatementWithParams (and its inherited TSQLDBOracleStatement)

```
TSQLDBParamInOutType = ( 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)

```
TSQLDBStatementCRUD = ( cCreate, cRead, cUpdate, cDelete );
```

*Identify a CRUD mode of a statement*

```
TSQLDBStatementCRUDs = set of TSQLDBStatementCRUD;
```

*Identify the CRUD modes of a statement*

- used e.g. for batch send abilities of a DB engine

```
TSQLDBStatementGetCol = ( colNone, colNull, colWrongType, colTmpUsed,  
colTmpUsedTruncated );
```

*Possible column retrieval patterns*

- used by TSQLDBColumnProperty.ColumnValueState

### Constants implemented in the *SynDB* unit:

```
FIXEDLENGTH_SQLDBFIELDTYPE = [ftInt64, ftDouble, ftCurrency, ftDate];
```

*TSQLDBFieldKind* kind of columns which have a fixed width

### Functions or procedures implemented in the *SynDB* unit:

| Functions or procedures | Description                                                     | Page |
|-------------------------|-----------------------------------------------------------------|------|
| LogTruncatedColumn      | Function helper logging some column truncation information text | 428  |
| TrimLeftSchema          | Retrieve a table name without any left schema                   | 428  |

```
procedure LogTruncatedColumn(const Col: TSQLDBColumnProperty);
```

*Function helper logging some column truncation information text*

```
function TrimLeftSchema(const TableName: RawUTF8): RawUTF8;
```

*Retrieve a table name without any left schema*  
- e.g. TrimLeftSchema('SCHEMA.TABLENAME')='TABLENAME'

### 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 SQLite3Commons  
- since not all exceptions are handled specifically by this unit, you may better use a common TSynLog class for the whole application or module

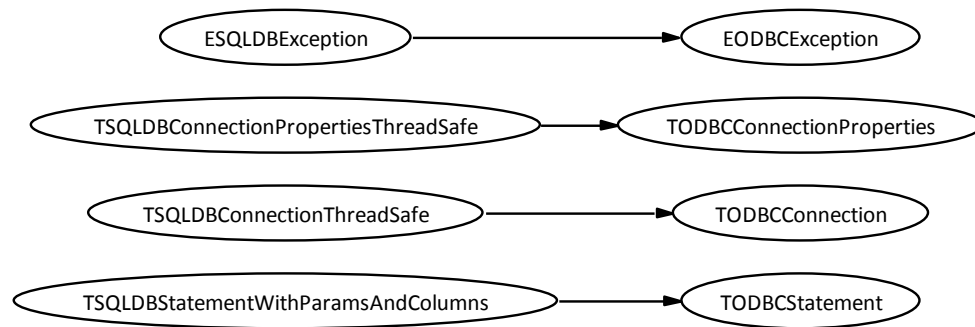
#### 1.4.7.6. SynDBODBC unit

*Purpose:* ODBC 3.x library direct access classes to be used with our SynDB architecture  
- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

### Units used in the *SynDBODBC* unit:

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |
| <i>SynDB</i>      | Abstract database direct access classes<br>- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17               | 395  |





*SynDBODBC class hierarchy*

### Objects implemented in the *SynDBODBC* unit:

| Objects                   | Description                                           | Page |
|---------------------------|-------------------------------------------------------|------|
| EODBCException            | Generic Exception type, generated for ODBC connection | 429  |
| TODBCConnection           | Implements a direct connection to the ODBC library    | 430  |
| TODBCConnectionProperties | Will implement properties shared by the ODBC library  | 429  |
| TODBCStatement            | Implements a statement using a ODBC connection        | 431  |

**EODBCException** = **class**(ESQLErrorException)

*Generic Exception type, generated for ODBC connection*

**TODBCConnectionProperties** = **class**(TSQLErrorConnectionPropertiesThreadSafe)

*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

**function** NewConnection: TSQLErrorConnection; **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; **var** Fields: TSQLErrorColumnDefineDynArray); **override**;

*Retrieve the column/field layout of a specified table*

- will retrieve the corresponding metadata from ODBC library if SQL direct access was not defined

**procedure** GetForeignKeys; **override**;

*Initialize fForeignKeys content with all foreign keys of this DB*  
- used by GetForeignKey method

**procedure** GetTableNames(**var** Tables: TRawUTF8DynArray); **override**;

*Get all table names*  
- will retrieve the corresponding metadata from ODBC library if SQL direct access was not defined

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*

**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 TSQLDBConnection 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 if it was not a blob, 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 exception 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 a prepared SQL statement*

- parameters marked as ? should have been already bound with Bind\*() functions
- raise an ESQLErrorException 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 ESQLErrorException on any error

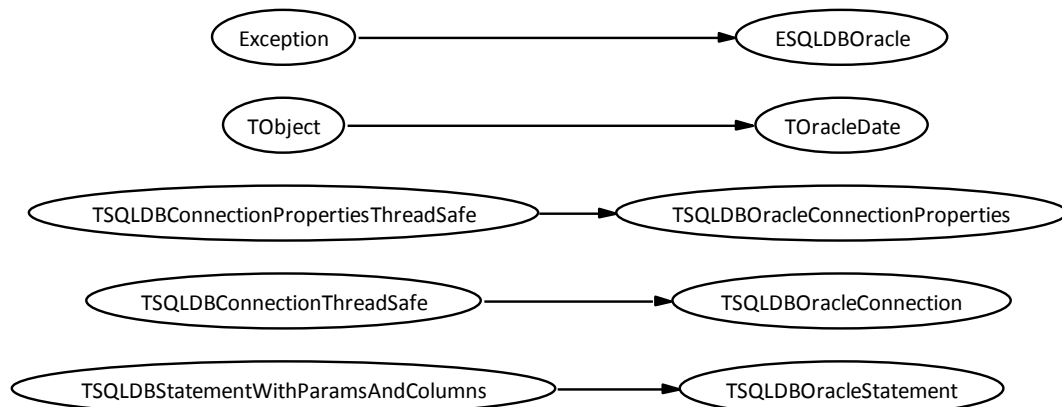
#### 1.4.7.7. SynDBOracle unit

*Purpose:* Oracle DB direct access classes (via OCI)

- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**Units used in the SynDBOracle unit:**

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |
| <i>SynDB</i>      | Abstract database direct access classes<br>- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17               | 395  |



*SynDBOracle class hierarchy*

### Objects implemented in the *SynDBOracle* unit:

| Objects                          | Description                                                                    | Page |
|----------------------------------|--------------------------------------------------------------------------------|------|
| ESQldbOracle                     | Exception type associated to the native Oracle Client Interface (OCI)          | 433  |
| TOracleDate                      | Memory structure used to store a date and time in native Oracle format         | 433  |
| TSQldbOracleConnection           | Implements a direct connection to the native Oracle Client Interface (OCI)     | 435  |
| TSQldbOracleConnectionProperties | Will implement properties shared by native Oracle Client Interface connections | 434  |
| TSQldbOracleStatement            | Implements a statement via the native Oracle Client Interface (OCI)            | 436  |

**ESQldbOracle = class(Exception)**

*Exception type associated to the native Oracle Client Interface (OCI)*

**TOracleDate = 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*

**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*  
- inherited from TSQldbConnectionPropertiesThreadSafe so that the oci.dll library is not initialized with OCI\_THREADED: this could make the process faster on multi-core CPU and a multi-threaded server application

```
constructor Create(const aServerName, aDatabaseName, aUserID, aPassWord:  
RawUTF8); override;
```

*Initialize the connection properties*  
- aDatabaseName is not used for Oracle: only aServerName is to be set  
- this overridden method will force the code page to be zero: you shall better call the CreateWithCodePage constructor instead of this generic method

```
constructor CreateWithCodePage(const aServerName, aUserID, aPassWord: RawUTF8;  
aCodePage: integer); virtual;
```

*Initialize the OCI connection properties*  
- we don't need a database name parameter for Oracle connection  
- you may specify the TNSName in aServerName, or a connection string like '//host[:port]/[service\_name]', e.g. '//sales-server:1523/sales'  
- since the OCI client will make conversion when returning column data, to avoid any truncate when retrieving VARCHAR2 or CHAR fields into the internal fixed-sized buffer, you may specify here the exact database code page, as existing on the server (e.g. CODEPAGE\_US=1252 for default WinAnsi western encoding) - if aCodePage is set to 0, either the global NLS\_LANG environnement variable is used, either the thread setting (GetACP)

```
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. '11.2.0.1 at oci.dll'*

**property** CodePage: cardinal read fCodePage;

*The code page used for the connection*

- e.g. 1252 for default CODEPAGE\_US
- connection is opened globally as UTF-8, to match the internal encoding of our units; but CHAR / NVARCHAR2 fields will use this code page encoding to avoid any column truncation when retrieved from the server

**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** StatementCacheSize: integer read fStatementCacheSize write fStatementCacheSize;

*The number of prepared statements cached by OCI on the Client side*

- is set to 30 by default

**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*

- 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 Oracle database server*

- should raise an Exception on error
- the connection will be globally opened with UTF-8 encoding; for CHAR / NVARCHAR2 fields, the TSQldbOracleConnectionProperties.CodePage encoding will be used instead, to avoid any truncation during data retrieval
- BlobPrefetchSize 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)

**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 TOLDBConnection type, will raise an exception

**destructor Destroy; override;**

*Release all associated memory and OCI 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 if it 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 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** ColumnToVarData(Col: Integer; var Value: TVarData; var Temp: RawByteString): TSQLDBFieldType; **override;**

*Return a Column as a TVarData value, first Col is 0*

- TVarData returned types are varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0]); ftDate is returned as varString, as in TSQLDBStatement.ColumnsToJSON
- so this Value should not be used typecasted to a Variant
- the specified Temp variable will be used for temporary storage of varString/varAny 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 SQLite3DB (i.e. mORMot external DB access)

**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 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

**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 ""\uFFFF0base64encodedbinary" 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 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

**Types implemented in the SynDBOracle unit:**

TOracleDateArray = array[0..(maxInt div sizeof(TOracleDate))-1] of TOracleDate;

*Wrapper to an array of TOracleDate items*

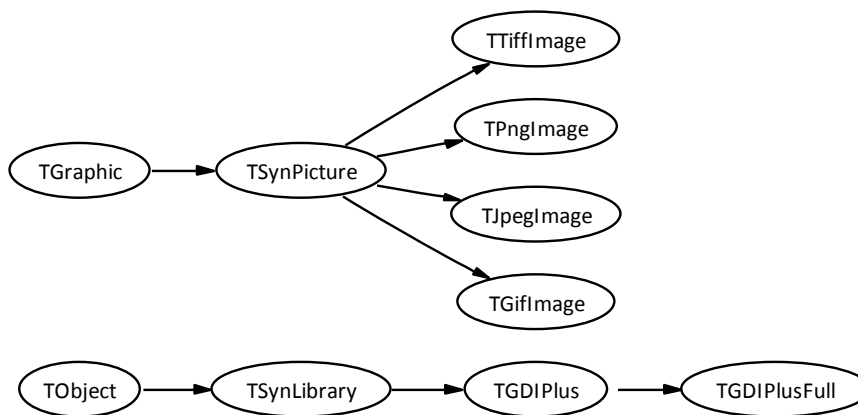
#### 1.4.7.8. SynGdiPlus unit

*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 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 1.17

**The SynGdiPlus unit is quoted in the following items:**

| SWRS #   | Description                                                                                | Page |
|----------|--------------------------------------------------------------------------------------------|------|
| DI-2.3.2 | A reporting feature, with full preview and export as PDF or TXT files, shall be integrated | 835  |



*SynGdiPlus class hierarchy*

**Objects implemented in the SynGdiPlus unit:**

| Objects         | Description                                                | Page |
|-----------------|------------------------------------------------------------|------|
| TGdipBitmapData | Data as retrieved by GdipFull.BitmapLockBits               | 439  |
| TGDIPlus        | Handle picture related GDI+ library calls                  | 439  |
| TGDIPlusFull    | Handle most GDI+ library calls                             | 441  |
| TGdipPointF     | GDI+ floating point coordinates for a point                | 439  |
| TGdipRect       | GDI+ integer coordinates rectangles                        | 439  |
| TGdipRectF      | GDI+ floating point coordinates rectangles                 | 439  |
| TGifImage       | Sub class to handle .GIF file extension                    | 441  |
| TJpegImage      | Sub class to handle .JPG file extension                    | 441  |
| TPngImage       | Sub class to handle .PNG file extension                    | 441  |
| TSynLibrary     | An object wrapper to load dynamically a library            | 439  |
| TSynPicture     | GIF, PNG, TIFF and JPG pictures support using GDI+ library | 440  |
| TTiffImage      | Sub class to handle .TIF file extension                    | 441  |

**TGdipRect = packed record**

*GDI+ integer coordinates rectangles*

- use width and height instead of right and bottom

**TGdipRectF = packed record**

*GDI+ floating point coordinates rectangles*

- use width and height instead of right and bottom

**TGdipPointF = packed record**

*GDI+ floating point coordinates for a point*

**TGdipBitmapData = packed record**

*Data as retrieved by GdipFull.BitmapLockBits*

**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 835).*

**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 TGDIPPlus implementation uses GDI drawing only
- use a TGDIPPlusFull 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

*Used for DI-2.3.2 (page 835).*

**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 TGDIPPlusFull overridden method for true GDI+ AntiAliased drawing

*Used for DI-2.3.2 (page 835).*

**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\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms536393(VS.85).aspx) for all available image formats

**function** GetImageFormat: TGDIPPictureType;

*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: TGDIPPictureType; 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 gptTIF format, use ord(TGDIPPEncoderValue) to define the parameter; by default, will use ord(evCompressionLZW) to save the TIFF picture with LZW - for gptTIF, 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*

**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

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 GDIPLUS.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; out hGlobal: THandle): IStream;

*Internal method used for GDI32 metafile loading*

**procedure** DrawAntiAliased(Source: TMetafile; Dest: HDC; R: TRect; aSmoothing: TSmoothingMode=smAntiAlias; aTextRendering: TTextRenderingHint=trhClearTypeGridFit); **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 GdiConvertToEmfPlus() 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*

TGDIPPEncoderValue =  
 ( evColorTypeCMYK, evColorTypeYCK, evCompressionLZW, evCompressionCCITT3,  
 evCompressionCCITT4, evCompressionRle, evCompressionNone, evScanMethodInterlaced,  
 evScanMethodNonInterlaced, evVersionGif87, evVersionGif89, evRenderProgressive,  
 evRenderNonProgressive, evTransformRotate90, evTransformRotate180,

```
evTransformRotate270, evTransformFlipHorizontal, evTransformFlipVertical,
evMultiFrame, evLastFrame, evFlush, evFrameDimensionTime,
evFrameDimensionResolution, evFrameDimensionPage );
```

*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*

```
TGdipStatus =
( stOk, stGenericError, stInvalidParameter, stOutOfMemory, stObjectBusy,
stInsufficientBuffer, stNotImplemented, stWin32Error, stWrongState, stAborted,
stFileNotFound, stValueOverflow, stAccessDenied, stUnknownImageFormat,
stFontFamilyNotFound, stFontStyleNotFound, stNotTrueTypeFont,
stUnsupportedGdiplusVersion, stGdiplusNotInitialized, stPropertyNotFound,
stPropertyNotSupported );
```

*GDI+ error codes*

```
TLockModeOption = ( lmRead, lmWrite, lmUserInputBuf );
```

*GDI+ lock mode for Gdiplus.BitmapLockBits*

```
TLockModeOptions = set of TLockModeOption;
```

*GDI+ lock mode settings for Gdiplus.BitmapLockBits*

```
TSmoothingMode = ( smDefault, smHighSpeed, smHighQuality, smNone, smAntiAlias );
```

*GDI+ line drawing smoothing types*

```
TTextRenderingHint =
( 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 | Description                                                                  | Page |
|-------------------------|------------------------------------------------------------------------------|------|
| DrawEmfGdip             | Draw the specified GDI TMetaFile (emf) using the GDI-plus antialiased engine | 444  |
| GdipTest                | Test function                                                                | 444  |



| Functions or procedures | Description                                                               | Page |
|-------------------------|---------------------------------------------------------------------------|------|
| LoadFrom                | Helper function to create a bitmap from any EMF content                   | 444  |
| LoadFrom                | Helper function to create a bitmap from any GIF/PNG/JPG/TIFF/EMF/WMF file | 444  |
| LoadFromRawByteString   | Helper to load a specified graphic from GIF/PNG/JPG/TIFF format content   | 444  |
| PictureName             | Retrieve a ready to be displayed name of the supplied Graphic Class       | 444  |
| SaveAs                  | Helper to save a specified graphic into GIF/PNG/JPG/TIFF format           | 445  |
| SaveAs                  | Helper to save a specified graphic into GIF/PNG/JPG/TIFF format           | 445  |
| SaveAsRawByteString     | Helper to save a specified graphic into GIF/PNG/JPG/TIFF format           | 445  |

```
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 TGDIPPlusFull.ForceUseDrawString property)

```
procedure GdipTest(const JpegFile: TFileName);
```

*Test function*

```
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)

```
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)

```
function LoadFromRawByteString(const Picture: AnsiString): TBitmap;
```

*Helper to load a specified graphic from GIF/PNG/JPG/TIFF format content*

```
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



```
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 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

```
procedure SaveAsRawByteString(Graphic: TPersistent; out Data: AnsiString; 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 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

#### 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

#### 1.4.7.9. *SynLZ* unit

*Purpose:* SynLZ Compression routines

- licensed under a MPL/GPL/LGPL tri-license; version 1.17

#### Types implemented in the *SynLZ* unit:

```
RawByteString = AnsiString;
```

*Define RawByteString, as it does exist in Delphi 2009 and up*

- to be used for byte storage into an AnsiString

#### Functions or procedures implemented in the *SynLZ* unit:

| Functions or procedures | Description                                                                | Page |
|-------------------------|----------------------------------------------------------------------------|------|
| CompressSynLZ           | Compress a data content using the SynLZ algorithm                          | 446  |
| SynLZcompress1asm       | Optimized asm version of the 1st compression method                        | 446  |
| SynLZcompress1pas       | 1st compression method uses hashing with a 32bits control word             | 446  |
| SynLZcompress2          | 2nd compression method optimizes pattern copy -> a bit smaller, but slower | 446  |
| SynLZcompressdestlen    |                                                                            | 446  |

| Functions or procedures | Description                                                                | Page |
|-------------------------|----------------------------------------------------------------------------|------|
| SynLZdecompress1asm     | Optimized asm version of the 1st compression method                        | 446  |
| SynLZdecompress1pas     | 1st compression method uses hashing with a 32bits control word             | 446  |
| SynLZdecompress2        | 2nd compression method optimizes pattern copy -> a bit smaller, but slower | 446  |
| SynLZdecompressdestlen  | Get uncompressed size from lz-compressed buffer (to reserve memory, e.g.)  | 446  |

**function** CompressSynLZ(**var** Data: RawByteString; Compress: boolean): RawByteString;

*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** SynLZcompress1asm(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;

*Optimized asm version of the 1st compression method*

**function** SynLZcompress1pas(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;

*1st compression method uses hashing with a 32bits control word*

**function** SynLZcompress2(src: PAnsiChar; size: integer; dst: PAnsiChar): integer;

*2nd compression method optimizes pattern copy -> a bit smaller, but slower*

**function** SynLZcompressdestlen(in\_len: integer): integer;

*Disable Range checking in our code disable Stack checking in our code expect extended syntax  
disable stack frame generation disable overflow checking in our code expect short circuit boolean  
disable Var-String Checking Typed @ operator enumerators stored as byte by default Open string  
params get maximum possible (worse) compressed size for out\_p*

**function** SynLZdecompress1asm(src: PAnsiChar; size: integer; dst: PAnsiChar): Integer;

*Optimized asm version of the 1st compression method*

**function** SynLZdecompress1pas(src: PAnsiChar; size: integer; dst: PAnsiChar): Integer;

*1st compression method uses hashing with a 32bits control word*

**function** SynLZdecompress2(src: PAnsiChar; size: integer; dst: PAnsiChar): Integer;

*2nd compression method optimizes pattern copy -> a bit smaller, but slower*

**function** SynLZdecompressdestlen(in\_p: PAnsiChar): integer;

*Get uncompressed size from lz-compressed buffer (to reserve memory, e.g.)*

#### 1.4.7.10. SynLZO 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:

| Functions or procedures  | Description                                                                         | Page |
|--------------------------|-------------------------------------------------------------------------------------|------|
| CompressSynLZO           | (de)compress a data content using the SynLZO algorithm                              | 447  |
| lzopas_compress          | Compress in_p(in_len) into out_p                                                    | 447  |
| lzopas_compressdestlen   | Get maximum possible (worse) compressed size for out_p                              | 447  |
| lzopas_decompress        | Uncompress in_p(in_len) into out_p (must be allocated before call), returns out_len | 447  |
| lzopas_decompressdestlen | Get uncompressed size from lzo-compressed buffer (to reserve memory, e.g.)          | 447  |

**function** CompressSynLZO(**var** Data: AnsiString; Compress: boolean): AnsiString;

*(de)compress a data content using the SynLZO algorithm*

- as expected by THttpSocket.RegisterCompress
- will return 'synlzo' 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** lzopas\_compress(in\_p: PAnsiChar; in\_len: integer; out\_p: PAnsiChar): integer;

*Compress in\_p(in\_len) into out\_p*

- out\_p must be at least lzopas\_compressdestlen(in\_len) bytes long
- returns compressed size in out\_p

**function** lzopas\_compressdestlen(in\_len: integer): integer;

*Get maximum possible (worse) compressed size for out\_p*

**function** lzopas\_decompress(in\_p: PAnsiChar; in\_len: integer; out\_p: PAnsiChar): Integer;

*Uncompress in\_p(in\_len) into out\_p (must be allocated before call), returns out\_len*

- may write up to out\_len+3 bytes in out\_p
- the decompression mode is "fast-unsafe" -> CRC/Adler32 in\_p data before call

**function** lzopas\_decompressdestlen(in\_p: PAnsiChar): integer;

*Get uncompressed size from lzo-compressed buffer (to reserve memory, e.g.)*

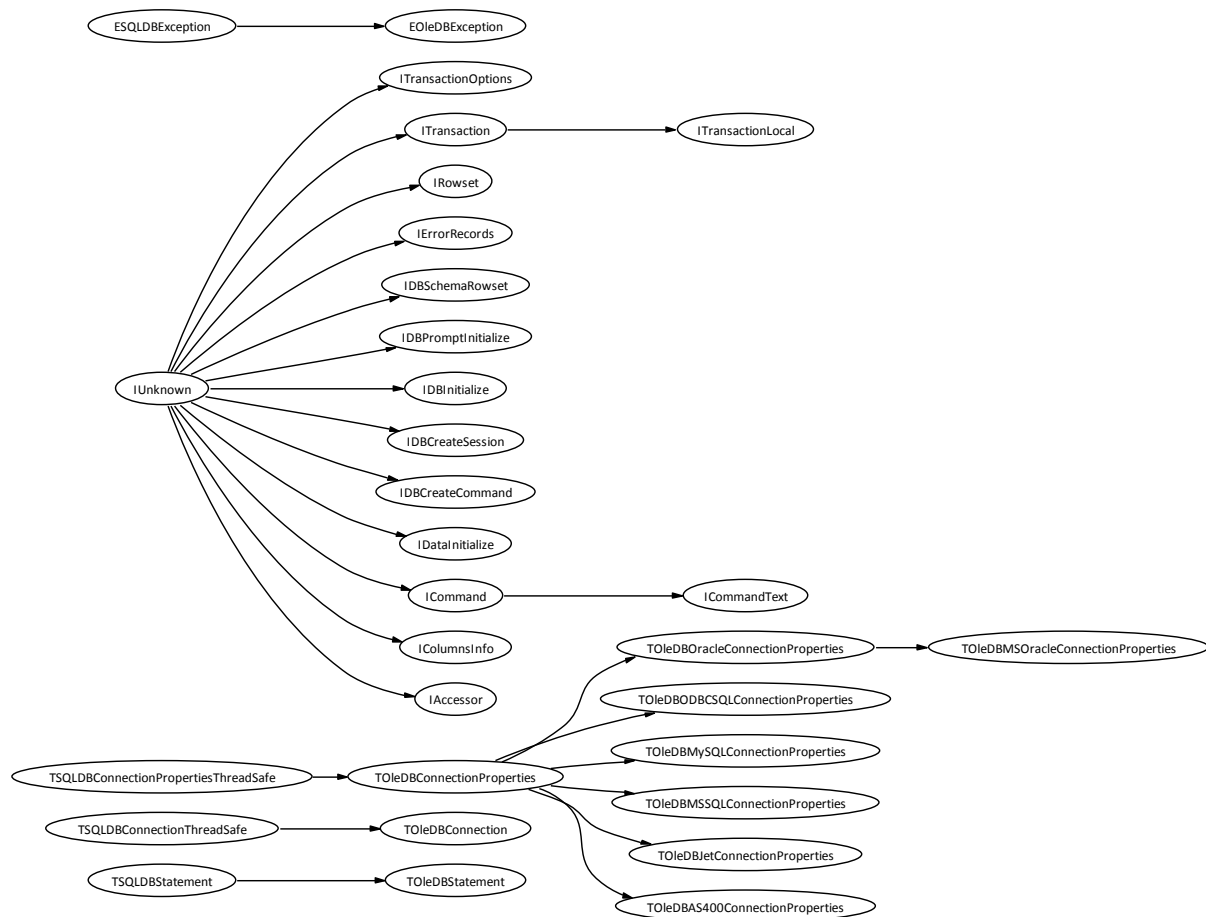
#### 1.4.7.11. SynOleDB unit

*Purpose:* Fast OleDB direct access classes

- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

### Units used in the *SynOleDB* unit:

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |
| <i>SynDB</i>      | Abstract database direct access classes<br>- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17               | 395  |



*SynOleDB class hierarchy*

### Objects implemented in the *SynOleDB* unit:

| Objects                | Description                                                             | Page |
|------------------------|-------------------------------------------------------------------------|------|
| <i>EOleDbException</i> | Generic Exception type, generated for OleDb connection                  | 451  |
| <i>IAccessor</i>       | Provides methods for accessor management, to access OleDb data          | 451  |
| <i>IColumnsInfo</i>    | Expose information about columns of an OleDb rowset or prepared command | 451  |
| <i>ICommand</i>        | Provide methods to execute commands                                     | 450  |

| Objects                            | Description                                                                                          | Page |
|------------------------------------|------------------------------------------------------------------------------------------------------|------|
| ICommandText                       | Methods to access the ICommand text to be executed                                                   | 450  |
| IDataInitialize                    | Create an OleDB data source object using a connection string                                         | 450  |
| IDBCreateCommand                   | Used on an OleDB session to obtain a new command                                                     | 451  |
| IDBCreateSession                   | Obtain a new session to a given OleDB data source                                                    | 450  |
| IDBInitialize                      | Initialize and uninitialize OleDB data source objects and enumerators                                | 450  |
| IDBPromptInitialize                | Allows the display of the data link dialog boxes programmatically                                    | 451  |
| IDBSchemaRowset                    | Used to retrieve the database metadata (e.g. tables and fields layout)                               | 451  |
| IErrorRecords                      | Interface used to retrieve enhanced custom error information                                         | 450  |
| IRowset                            | Provides methods for fetching rows sequentially, getting the data from those rows, and managing rows | 450  |
| ITransaction                       | Commit, abort, and obtain status information about OleDB transactions                                | 450  |
| ITransactionLocal                  | Optional interface on OleDB sessions, used to start, commit, and abort transactions on the session   | 450  |
| ITransactionOptions                | Gets and sets a suite of options associated with an OleDB transaction                                | 450  |
| TOleDBAS400ConnectionProperties    | OleDB connection properties to IBM AS/400                                                            | 452  |
| TOleDBConnection                   | Implements an OleDB connection                                                                       | 453  |
| TOleDBConnectionProperties         | Will implement properties shared by OleDB connections                                                | 451  |
| TOleDBJetConnectionProperties      | OleDB connection properties to Jet/MSAccess .mdb files                                               | 452  |
| TOleDBMSOracleConnectionProperties | OleDB connection properties to an Oracle database using Microsoft's Provider                         | 452  |
| TOleDBMSSQLConnectionProperties    | OleDB connection properties to Microsoft SQL Server                                                  | 452  |
| TOleDBMySQLConnectionProperties    | OleDB connection properties to MySQL Server                                                          | 452  |
| TOleDBODBCSQLConnectionProperties  | OleDB connection properties via Microsoft Provider for ODBC                                          | 453  |
| TOleDBOracleConnectionProperties   | OleDB connection properties to an Oracle database using Oracle's Provider                            | 452  |
| TOleDBStatement                    | Implements an OleDB SQL query statement                                                              | 455  |
| TOleDBStatementParam               | Used to store properties and value about one TOleDBStatement Param                                   | 454  |

**IDBInitialize = interface(IUnknown)**

*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: UINT; rghRows: PCardinalArray; rgRefCounts: PCardinalArray; 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: Integer; cRows: Integer; **out** pcRowsObtained: UINT; **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: PCardinalArray; rgRowOptions, rgRefCounts, 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** 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; var 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
- if GetForeignKey is TRUE, will retrieve ColumnForeignKey\* properties, but will be much slower

**procedure** GetTableNames(var 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](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\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms675851(v=VS.85).aspx)

`ToleDBMSSQLConnectionProperties = class(ToleDBConnectionProperties)`

*OleDB connection properties to Microsoft SQL Server*

- this will use the native OleDB provider supplied by Microsoft see [http://msdn.microsoft.com/en-us/library/ms677227\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms677227(v=VS.85).aspx)
- is `aUserID=""` at `Create`, it will use Windows Integrated Security for the connection

`ToleDBMySQLConnectionProperties = class(ToleDBConnectionProperties)`

*OleDB connection properties to MySQL Server*

`ToleDBJetConnectionProperties = class(ToleDBConnectionProperties)`

*OleDB connection properties to Jet/MSAccess .mdb files*

- the server name should be the .mdb file name

`ToleDBAS400ConnectionProperties = class(ToleDBConnectionProperties)`

*OleDB connection properties to IBM AS/400*



**TOLeDBODBCSQLConnectionProperties = class(TOLeDBConnectionProperties)**

*OleDB connection properties via Microsoft Provider for ODBC*

- this will use the ODBC provider supplied by Microsoft see [http://msdn.microsoft.com/en-us/library/ms675326\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms675326(v=VS.85).aspx)
- an ODBC Driver should be specified at creation
- you should better use direct connection classes, like TOLeDBMSSQLConnectionProperties or TOLeDBOracleConnectionProperties

**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*

**TOLeDBConnection = class(TSQLDBConnectionThreadSafe)**

*Implements an OleDB connection*

- will retrieve the remote DataBase 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 [http://msdn.microsoft.com/en-us/library/ms716985\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms716985(v=vs.85).aspx)

**property OleDbProperties: TOleDbConnectionProperties read fOleDbProperties;**

*The associated OleDb database properties*

**TOleDbStatementParam = packed record**

*Used to store properties and value about one TOleDbStatement Param*

- we don't use a Variant, not the standard TSQLEDBParam 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

**VBlob: RawByteString;**

*Storage used for BLOB (ftBlob) values*

- will be referred as DBTYPE\_BYREF when sent as OleDb parameters, to avoid unnecessary memory copy

**VFill:**

**array[ sizeof(TSQLEDBFieldType)+sizeof(TSQLEDBParamInOutType)..SizeOf(Int64)-1] of byte;**

*So that VInt64 will be 8 bytes aligned*

**VInOut: TSQLEDBParamInOutType;**

*Define if parameter can be retrieved after a stored procedure execution*

**VInt64: Int64;**

*Storage used for ftInt64, ftDouble, ftDate and ftCurrency value*

**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: TSQLEDBFieldType;**

*The column/parameter Value type*

**TOLeDBStatement = 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 if it was not a Blob, 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 that 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 TBlobData 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

```
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 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); override;
```

*Bind a NULL value to a parameter*

- the leftmost SQL parameter has an index of 1
- 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); 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 ""\uFFFF0base64encodedbinary" 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);
```

*Retrieve column information from a supplied IRowSet*

- is used e.g. by TOleDBStatement.Execute or to retrieve metadata columns
- raise an exception on 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

## 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>

```
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, stPermissionDenied,
stIntegrityViolation, stBadStatus, stDefault );
```

*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*

- see [http://msdn.microsoft.com/en-us/library/ms722617\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms722617(VS.85).aspx)

#### Functions or procedures implemented in the *SynOleDB* unit:

| Functions or procedures | Description                                                               | Page |
|-------------------------|---------------------------------------------------------------------------|------|
| CoInit                  | This global procedure should be called for each thread needing to use OLE | 459  |
| CoUninit                | This global procedure should be called at thread termination              | 459  |
| IsJetFile               | Check from the file beginning if sounds like a valid Jet / MSAccess file  | 459  |

#### **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*

#### 1.4.7.12. SynPdf unit

*Purpose:* PDF file generation

- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

The *SynPdf* unit is quoted in the following items:

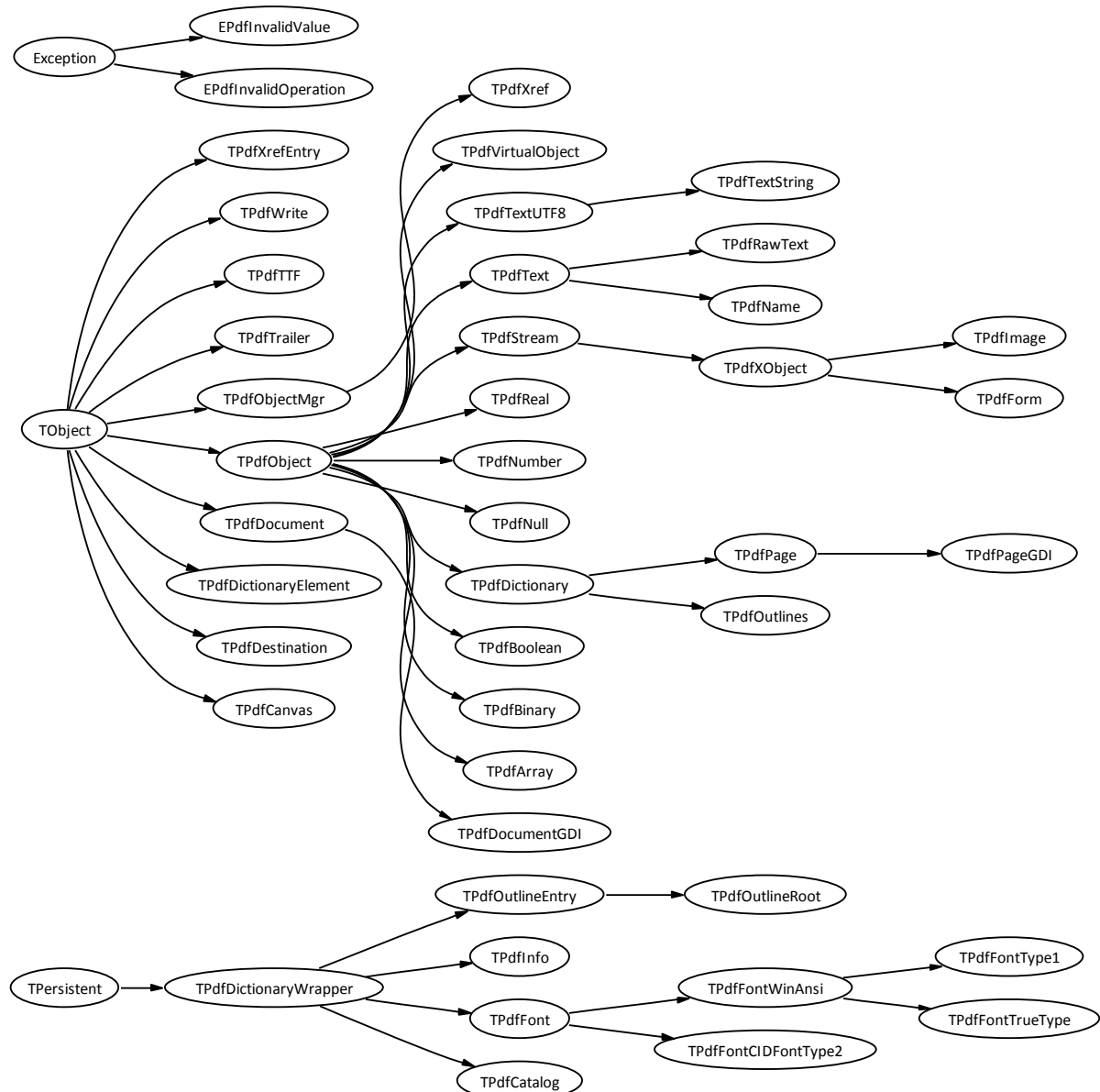
| SWRS #   | Description                                                                                | Page |
|----------|--------------------------------------------------------------------------------------------|------|
| DI-2.3.2 | A reporting feature, with full preview and export as PDF or TXT files, shall be integrated | 835  |

#### Units used in the *SynPdf* unit:

| Unit Name | Description | Page |
|-----------|-------------|------|
|-----------|-------------|------|



| Unit Name         | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                      | 229  |
| <i>SynGdiPlus</i> | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynZip</i>     | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                | 560  |





*SynPdf class hierarchy*

**Objects implemented in the *SynPdf* unit:**

| Objects               | Description                                                                                                                                                                                                                                  | Page |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| EPdfInvalidOperation  | PDF exception, raised when an invalid operation is triggered                                                                                                                                                                                 | 463  |
| EPdfInvalidValue      | PDF exception, raised when an invalid value is given to a constructor                                                                                                                                                                        | 463  |
| TCmapFmt4             | Header for the 'cmap' Format 4 table                                                                                                                                                                                                         | 463  |
| TCmapHEAD             | The 'head' table contains global information about the font                                                                                                                                                                                  | 463  |
| TCmapHeader           | The 'cmap' table begins with an index containing the table version number followed by the number of encoding tables. The encoding subtables follow.                                                                                          | 463  |
| TCmapHHEA             | 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 | 463  |
| TPdfArray             | Used to store an array of PDF objects                                                                                                                                                                                                        | 467  |
| TPdfBinary            | Used to handle object which are not defined in this library                                                                                                                                                                                  | 471  |
| TPdfBoolean           | A PDF object, storing a boolean value                                                                                                                                                                                                        | 466  |
| TPdfBox               | A PDF coordinates box                                                                                                                                                                                                                        | 463  |
| TPdfCanvas            | Access to the PDF Canvas, used to draw on the page                                                                                                                                                                                           | 478  |
| TPdfDestination       | A destination defines a particular view of a document, consisting of the following:                                                                                                                                                          | 489  |
| TPdfDictionary        | A PDF Dictionary is used to manage Key / Value pairs                                                                                                                                                                                         | 468  |
| TPdfDictionaryElement | PDF dictionary element definition                                                                                                                                                                                                            | 468  |
| TPdfDictionaryWrapper | Common ancestor to all dictionary wrapper classes                                                                                                                                                                                            | 485  |
| TPdfDocument          | The main class of the PDF engine, processing the whole PDF document                                                                                                                                                                          | 472  |
| TPdfDocumentGDI       | Class handling PDF document creation using GDI commands                                                                                                                                                                                      | 491  |
| TPdfFont              | A generic PDF font object                                                                                                                                                                                                                    | 486  |
| TPdfFontCIDFontType2  | An embedded Composite CIDFontType2                                                                                                                                                                                                           | 487  |
| TPdfFontTrueType      | Handle TrueType Font                                                                                                                                                                                                                         | 488  |
| TPdfFontType1         | An embedded WinAnsi-Encoded standard Type 1 font                                                                                                                                                                                             | 487  |
| TPdfFontWinAnsi       | A generic PDF font object, handling at least WinAnsi encoding                                                                                                                                                                                | 486  |
| TPdfForm              | Handle any form XObject                                                                                                                                                                                                                      | 492  |

| Objects           | Description                                                                  | Page |
|-------------------|------------------------------------------------------------------------------|------|
| TPdfImage         | Generic image object                                                         | 492  |
| TPdfInfo          | A dictionary wrapper class for the PDF document information fields           | 485  |
| TPdfName          | A PDF object, storing a PDF name                                             | 467  |
| TPdfNull          | A PDF object, storing a NULL value                                           | 466  |
| TPdfNumber        | A PDF object, storing a numerical (integer) value                            | 466  |
| TPdfObject        | Master class for most PDF objects declaration                                | 466  |
| TPdfObjectMgr     | Object manager is a virtual class to manage instance of indirect PDF objects | 466  |
| TPdfOutlineEntry  | An Outline entry in the PDF document                                         | 490  |
| TPdfOutlineRoot   | Root entry for all Outlines of the PDF document                              | 491  |
| TPdfOutlines      | Generic PDF Outlines entries, stored as a PDF dictionary                     | 472  |
| TPdfPage          | A PDF page                                                                   | 477  |
| TPdfPageGDI       | A PDF page, with its corresponding Meta File and Canvas                      | 491  |
| TPdfRawText       | A PDF object, storing a raw PDF content                                      | 467  |
| TPdfReal          | A PDF object, storing a numerical (floating point) value                     | 466  |
| TPdfRect          | A PDF coordinates rectangle                                                  | 463  |
| TPdfStream        | A temporary memory stream, to be stored into the PDF content                 | 470  |
| TPdfText          | A PDF object, storing a textual value                                        | 466  |
| TPdfTextString    | A PDF object, storing a textual value                                        | 467  |
| TPdfTextUTF8      | A PDF object, storing a textual value                                        | 467  |
| TPdfTrailer       | The Trailer of the PDF File                                                  | 471  |
| TPdfTTF           | Handle Unicode glyph description for a True Type Font                        | 487  |
| TPdfVirtualObject | A virtual PDF object, with an associated PDF Object Number                   | 466  |
| TPdfWrite         | Buffered writer class, specialized for PDF encoding                          | 463  |
| TPdfXObject       | Any object stored to the PDF file                                            | 472  |
| TPdfXref          | Store the XRef list of the PDF file                                          | 472  |
| TPdfXrefEntry     | Store one entry in the XRef list of the PDF file                             | 471  |
| TScriptAnalysis   | An Uniscribe script analysis                                                 | 493  |
| TScriptItem       | A Uniscribe script item, after analysis of a unicode text                    | 493  |

| Objects           | Description                                                                                                         | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------|------|
| TScriptProperties | Contains information about Uniscribe special processing for each script                                             | 493  |
| TScriptState      | An UniScribe script state                                                                                           | 492  |
| TScriptVisAttr    | Contains the visual (glyph) attributes that identify clusters and justification points, as generated by ScriptShape | 493  |

#### 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*

#### TPdfWrite = **class**(TObject)

*Buffered writer class, specialized for PDF encoding*

**constructor** Create(DestStream: TStream; CodePage: integer);

*Create the buffered writer, for a specified destination stream*

**function** Add(Value: Extended): TPdfWrite; overload;

*Add a floating point numerical value to the buffer*  
- up to 2 decimals are written

**function** Add(const Text: RawByteString): TPdfWrite; overload;

*Direct raw write of some data*  
- no conversion is made

**function** Add(Text: PAnsiChar; Len: integer): 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: TColorRef): TPdfWrite;

*Add a PDF color, from its TColorRef RGB value*

**function** AddEscape(Text: PAnsiChar): TPdfWrite;

*Add some WinAnsi text as PDF text*  
- used by TPdfText object

**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): 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** 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: Integer): TPdfWrite; overload;

*Add an integer numerical value to the buffer*

- add a trailing space

**function** AddWithSpace(Value: Extended): 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: Extended; 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** 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

**constructor** CreateFmt(Fmt: PAnsiChar; **const** Args: array of Integer);

*Simple creator, replacing every % in Fmt by the corresponding Args[]*

**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*

```
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*

**property** Attributes: TPdfDictionary **read** FAttributes;

*Retrieve the associated attributes, e.g. the stream Length*

**property** Filter: TPdfArray **read** FFilter;

*Retrieve the associated filter*

**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 write FByteOffset;

*The position (in bytes) in the PDF file content stream*

**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** 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*

**TPdfDocument = class(TObject)**

*The main class of the PDF engine, processing the whole PDF document*

*Used for DI-2.3.2 (page 835).*

**constructor** Create(AUseOutlines: Boolean=false; ACodePage: integer=0; APDFA1: boolean=false); **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

**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): TPdfDictionary; overload;

*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** CreateLink(**const** ARect: TPdfRect; **const** aBookmarkName: RawUTF8): 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** CreateOrGetImage(B: TBitmap; DrawAt: 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

**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: TPdfImageHash; 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 algorithms, 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 dtXYZ destination with the corresponding TopPosition Y value is defined

- the associated bookmark name must be unique, otherwise an exception is raised

**procedure** NewDoc;

*Create a new document*

- this method is called first, by the Create constructor

- you can call it multiple time 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*

**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** DefaultPageWidth: cardinal **read** FDefaultPageWidth **write** SetDefaultPageWidth;

*The default page width, used for new every page creation (i.e. AddPage method call)*

**property** DefaultPaperSize: TPDFPaperSize **read** FDefaultPaperSize **write** SetDefaultPaperSize;

*The default page size, used for every new page creation (i.e. AddPage method call)*  
- a write to this property this will reset the default paper 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 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** 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** 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 descendant

**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** 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*

- usefull 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 835).*

**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(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** SetFont(const AName: RawUTF8; ASize: Single; AStyle: TFontStyles; 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** 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** 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** 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 specified XObject*

**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 work with embedded fonts by now, not true type fonts (because it use 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; Scale: Single=1.0; XOff: single=0.0;  
YOff: single=0.0; UseSetTextJustification: boolean=true; KerningHScaleBottom:  
single=99.0; KerningHScaleTop: single=101.0);
```

*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

*Used for DI-2.3.2 (page 835).*

```
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 font and 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, Tl, 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: Byte);

*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, Trise, 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 ETO\_GLYPH\_INDEX layout, i.e. refers to an array as returned from the GetCharacterPlacement: all glyph indexes are 16-bit values

**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** 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** 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*

**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 descendent will handle also Unicode chars, for all WideChar which are outside the WinAnsi selection

**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

- WidthArray[0]=Ascent, WidthArray[1]=Descent, WidthArray[2..]=Width(#32..)

**procedure** SetData(Value: TPdfDictionary); **override**;

*Set internally the font data*

**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#ottttables>

- 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 usefull 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: TFontStyles; **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 unicode char 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 unicode character*

- 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: TFontStyles **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);

*Create the PDF document instance, with a VCL Canvas property*

**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

**property** KerningHScaleBottom: Single **read** fKerningHScaleBottom **write** fKerningHScaleBottom;

*The % limit below which Font Kerning is transformed into PDF Horizontal Scaling commands*

- 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*

- set to 101.0 by default

**property** UseSetTextJustification: Boolean **read** fUseSetTextJustification **write** fUseSetTextJustification;

*UseSetTextJustification is to be set to true to ensure better rendering if the Canvas used SetTextJustification() API call to justify text*

- set to true by default

**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*

- usefull 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); **reintroduce**;

*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; aJpegFile: 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 EOpenError 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*

**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 internaly 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*

**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*

```
TPdfAnnotationSubType = ( asTextNotes, asLink );
```

*The annotation types determines the valid annotation subtype of TPdfDoc*

```
TPdfColor = -$FFFFFFF-1..$FFFFFFF;
```

*The available PDF color range*

```
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*

```
TPdfGDIComment = ( pgcOutline, pgcBookmark, pgcLink );
```

*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 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

```
TPdfImageHash = array[0..3] of cardinal;
```

*Array used to store a TPdfImage hash*

- uses 4 hash codes, created with 4 diverse algorithms, in order to avoid false positives

```
TPdfObjectType = ( otDirectObject, otIndirectObject, otVirtualObject );
```

*Allowed types for PDF objects (i.e. TPdfObject)*

```
TPdfPageLayout = ( plSinglePage, plOneColumn, plTwoColumnLeft, plTwoColumnRight );
```

*The page layout to be used when the document is opened*

```
TPdfPageMode = ( pmUseNone, pmUseOutlines, pmUseThumbs, pmFullScreen );
```

*Page mode determines how the document should appear when opened*

```
TPDFPaperSize =
```

```
( psA4, psA5, psA3, psLetter, psLegal, psUserDefined );
```

*Available known paper size (psA4 is the default on TPdfDocument creation)*

```
TPdfViewerPreference = ( vpHideToolbar, vpHideMenubar, vpHideWindowUI, vpFitWindow, vpCenterWindow );
```

*Viewer preferences specifying how the reader User Interface must start*

```
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, fNeedsCaretInfo, 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
- fNeedsCaretInfo: 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 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 embedd these fonts in the PDF document, and save some KB, just use the

EmbeddedTTFFIgnore property of TPdfDocument/TPdfDocumentGDI:

```
PdfDocument.EmbeddedTTFFIgnore.Text := MSWINDOWS_DEFAULT_FONTS;
```

- note that this is usefull 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*

```
PDF_IN_USE_ENTRY = 'n';
```

*Used for an used xref entry*

```
PDF_MAX_GENERATION_NUM = 65535;
```

*Used e.g. for the root xref entry*

```
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:

| Functions or procedures | Description                                                                                                                             | Page |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|------|
| CurrentPrinterPaperSize | Retrieve the paper size used by the current selected printer                                                                            | 497  |
| GDICommentBookmark      | Append a EMR_GDICOMMENT message for handling PDF bookmarks                                                                              | 498  |
| GDICommentLink          | Append a EMR_GDICOMMENT message for creating a Link into a specified bookmark                                                           | 498  |
| GDICommentOutline       | Append a EMR_GDICOMMENT message for handling PDF outline                                                                                | 498  |
| L2R                     | Reverse char orders for every hebrew and arabic words                                                                                   | 498  |
| PdfBox                  | Wrapper to create a temporary PDF box                                                                                                   | 498  |
| PdfRect                 | Wrapper to create a temporary PDF coordinates rectangle                                                                                 | 498  |
| PdfRect                 | Wrapper to create a temporary PDF coordinates rectangle                                                                                 | 498  |
| RawUTF8ToPDFString      | Convert a specified UTF-8 content into a PDFString value                                                                                | 498  |
| ScriptGetProperties     | Uniscribe function to retrieve information about the current scripts                                                                    | 498  |
| ScriptItemize           | Uniscribe function to break a Unicode string into individually shapeable items                                                          | 499  |
| ScriptLayout            | Uniscribe function to convert an array of run embedding levels to a map of visual-to-logical position and/or logical-to-visual position | 499  |
| ScriptShape             | Uniscribe function to generate glyphs and visual attributes for an Unicode run                                                          | 499  |
| _DateTimeToPdfDate      | Convert a date, into PDF string format, i.e. as 'D:20100414113241'                                                                      | 499  |
| _GetCharCount           | Return the number of glyphs in the supplied text                                                                                        | 500  |
| _HasMultiByteString     | This function returns TRUE if the supplied text contain any MBCS character                                                              | 500  |
| _PdfDateToDateTime      | Decode PDF date, encoded as 'D:20100414113241'                                                                                          | 500  |

```
function CurrentPrinterPaperSize: TPDFPaperSize;
```

*Retrieve the paper size used by the current selected printer*

**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);

*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 chars in the supplied buffer

**function** PdfBox(Left, Top, Width, Height: Single): TPdfBox;

*Wrapper to create a temporary PDF box*

**function** PdfRect(Left, Top, Right, Bottom: Single): TPdfRect; overload;

*Wrapper to create a temporary PDF coordinates rectangle*

**function** PdfRect(**const** Box: TPdfBox): TPdfRect; overload;

*Wrapper to create a temporary PDF coordinates rectangle*

**function** RawUTF8ToPDFString(**const** Value: RawUTF8): PDFString;

*Convert a specified UTF-8 content into a PDFString value*

**function** ScriptGetProperties(**out** ppSp: PScriptPropertiesArray; **out** piNumScripts: Integer): HRESULT; **stdcall**; **external** Usp10;

*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.

```
function ScriptItemize( const pwcInChars: PWideChar; cInChars: Integer;  
cMaxItems: Integer; const psControl: pointer; const psState: pointer; pItems:  
PScriptItem; var pcItems: Integer): HRESULT; stdcall; external Usp10;
```

*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\_CONTROL structure indicating the type of itemization to perform. Alternatively, the application can set this parameter to NULL if no SCRIPT\_CONTROL 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.
- pItem: 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 _DateTimeToPdfDate(ADate: TDateTime): TPdfDate;
```

*Convert a date, into PDF string format, i.e. as 'D:20100414113241'*



**function** \_GetCharCount(Text: PAnsiChar): integer;

*Return the number of glyphs in the supplied text*  
- work with MBCS strings

**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'*

#### 1.4.7.13. SynSelfTests unit

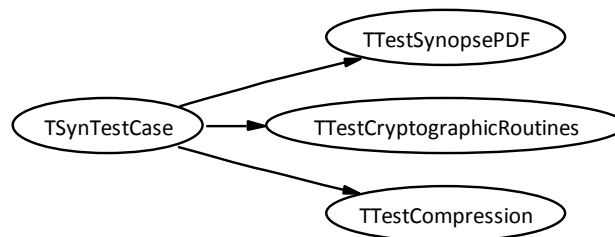
*Purpose:* Automated tests for common units of the Synopse Framework

- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

#### Units used in the SynSelfTests unit:

| Unit Name         | Description                                                                                                                                                                                                                                                                                                             | Page |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                               | 229  |
| <i>SynCrtSock</i> | Classes implementing HTTP/1.1 client and server protocol<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                     | 369  |
| <i>SynCrypto</i>  | Fast cryptographic routines (hashing and cypher)<br>- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms<br>- optimized for speed (tuned assembler and VIA PADLOCK optional support)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 383  |
| <i>SynDB</i>      | Abstract database direct access classes<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                             | 395  |
| <i>SynLZ</i>      | SynLZ Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                                                 | 445  |
| <i>SynLZO</i>     | Fast LZO Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.13                                                                                                                                                                                                                              | 447  |
| <i>SynPdf</i>     | PDF file generation<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                 | 459  |
| <i>SynZip</i>     | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                         | 560  |





*SynSelfTests class hierarchy*

### Objects implemented in the *SynSelfTests* unit:

| Objects                    | Description                                                                                              | Page |
|----------------------------|----------------------------------------------------------------------------------------------------------|------|
| TTestCompression           | This test case will test most functions, classes and types defined and implemented in the SynZip unit    | 501  |
| TTestCryptographicRoutines | This test case will test most functions, classes and types defined and implemented in the SynCrypto unit | 501  |
| TTestSynopsisPDF           | This test case will test most functions, classes and types defined and implemented in the SynPDF unit    | 502  |

**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 LZ77 deflate/inflate functions*

**procedure** ZipFormat;

*.zip archive handling*

**procedure** \_SynLZ;

*SynLZ internal format*

**procedure** \_SynLZO;

*SynLZO internal format*

**TTestCryptographicRoutines = class(TSynTestCase)**

*This test case will test most functions, classes and types defined and implemented in the SynCrypto unit*

**procedure** Adler32;

*Adler32 hashing functions*

**procedure** Base64;

*Base-64 encoding/decoding functions*

**procedure \_AES256;**  
*AES encryption/decryption functions*

**procedure \_MD5;**  
*MD5 hashing functions*

**procedure \_SHA1;**  
*SHA-1 hashing functions*

**procedure \_SHA256;**  
*SHA-256 hashing functions*

**TTestSynopsePDF = class(TSynTestCase)**  
*This test case will test most functions, classes and types defined and implemented in the SynPDF unit*

**procedure \_TPdfDocument;**  
*Create a PDF document, using the PDF Canvas property*  
- test font handling, especially standard font substitution

**procedure \_TPdfDocumentGDI;**  
*Create a PDF document, using a EMF content*  
- validates the EMF/TMetaFile enumeration, and its conversion into the PDF content  
- 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)

#### 1.4.7.14. SynSQLite3 unit

*Purpose:* SQLite3 embedded Database engine direct access

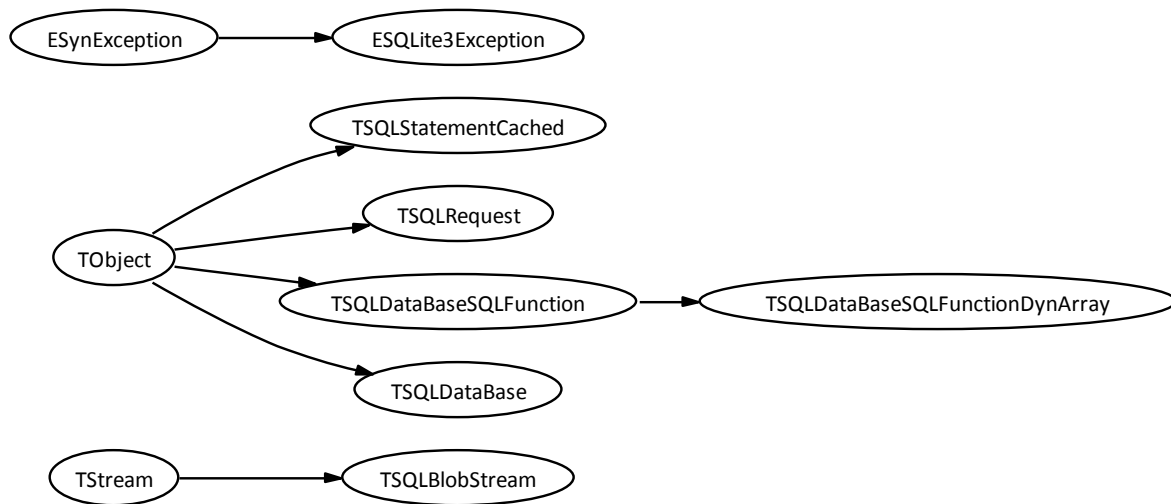
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**The SynSQLite3 unit is quoted in the following items:**

| SWRS #   | Description                                                  | Page |
|----------|--------------------------------------------------------------|------|
| DI-2.2.1 | The <i>SQLite3</i> engine shall be embedded to the framework | 832  |

**Units used in the SynSQLite3 unit:**

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SynSQLite3 class hierarchy*

#### Objects implemented in the *SynSQLite3* unit:

| Objects                              | Description                                                                                                                                                 | Page |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| ESQLite3Exception                    | Custom SQLite3 dedicated Exception type                                                                                                                     | 515  |
| TFTSMATCHINFO                        | Map the matchinfo function returned BLOB value                                                                                                              | 504  |
| TSQLErrorBlobStream                  | Used to read or write a BLOB Incrementaly                                                                                                                   | 527  |
| TSQLErrorDataBase                    | Simple wrapper for direct SQLite3 database manipulation                                                                                                     | 522  |
| TSQLErrorDataBaseSQLFunction         | Those classes can be used to define custom SQL functions inside a TSQLErrorDataBase                                                                         | 521  |
| TSQLErrorDataBaseSQLFunctionDynArray | To be used to define custom SQL functions for dynamic arrays BLOB search                                                                                    | 521  |
| TSQLErrorIndexConstraint             | Records WHERE clause constraints of the form "column OP expr"                                                                                               | 504  |
| TSQLErrorIndexConstraintUsage        | Define what information is to be passed to xFilter() for a given WHERE clause constraint of the form "column OP expr"                                       | 505  |
| TSQLErrorIndexInfo                   | 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 | 505  |
| TSQLErrorIndexOrderBy                | ORDER BY clause, one item per column                                                                                                                        | 505  |
| TSQLErrorModule                      | Defines a module object used to implement a virtual table.                                                                                                  | 507  |
| TSQLError3VTab                       | Virtual Table Instance Object                                                                                                                               | 506  |
| TSQLError3VTabCursor                 | Virtual Table Cursor Object                                                                                                                                 | 507  |
| TSQLErrorRequest                     | Wrapper to a SQLite3 request                                                                                                                                | 515  |

| Objects             | Description                           | Page |
|---------------------|---------------------------------------|------|
| TSQLStatementCache  | Used to retrieve a prepared statement | 520  |
| TSQLStatementCached | Handle a cache of prepared statements | 520  |

#### TFTSMatchInfo = 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](http://www.sqlite.org/fts3.html#appendix_a)

#### TSQLite3IndexConstraint = 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: boolean;

*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 constraints with usable set to false

#### **TSQLite3IndexOrderBy = record**

*ORDER BY clause, one item per column*

**desc:** boolean;

*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:** boolean;

*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*

**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

**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

**nRef: Integer;**

*No longer used*

**pInstance: TObject;**

*This will be used to store a Delphi class instance handling the Virtual Table*

**pModule: PSQLite3Module;**

*The module for this virtual table*

**zErrMsg: PAnsiChar;**

*Error message from sqlite3\_mprintf()*

- 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.

**pInstance: TObject;**

*This will be used to store a Delphi class instance handling the cursor*

**pVtab: PSQLite3VTab;**

*Virtual table of this cursor*

**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.

**iVersion: Integer;**

*Defines the particular edition of the module table structure*

- 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

**xBegin: function(var pVTab: TSQLite3VTab): Integer; cdecl;**

*Begins a transaction on a virtual 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.

**xBestIndex: function(var pVTab: TSQLite3VTab; var pInfo: TSQLite3IndexInfo): Integer; cdecl;**

*Used to determine the best way to access the virtual 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 again 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: PAnsiChar): 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: PUTF8CharArray; var ppVTab: PSQLite3VTab; var pzErr: PAnsiChar): 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;
```

*Begins 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: TSQLiteFunctionFunc; 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](http://www.sqlite.org/lang_savepoint.html)

- iSavepoint parameter indicates the unique name of the SAVEPOINT

```
xSync: 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, then 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.

**ESQLite3Exception = class(ESynException)**

*Custom SQLite3 dedicated Exception type*

**DB: TSQLite3DB;**

*The DB which raised this exception*

**ErrorCode: integer;**

*The corresponding error code*

**constructor** Create(const aMessage: string; aErrorCode: integer); reintroduce; overload;

*Create the exception, getting the message from caller*

**constructor** Create(aDB: TSQLite3DB; aErrorCode: integer); reintroduce; overload;

*Create the exception, getting the message from DB*

**TSQLRequest = object(TObject)**

*Wrapper to a SQLite3 request*

*Used for DI-2.2.1 (page 832).*

**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 832).*

**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 832).*

```
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 TSQLiteTableJSON)  
 { "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 832).*

```
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 internally 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 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 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;
```

*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** 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): integer;

*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

**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: 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** 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

**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** 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;

*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** 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 internally
- raise an ESQLite3Exception on any error

*Used for DI-2.2.1 (page 832).*

**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 832).*

**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 832).*

**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 832).*

**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*

- internaly call Prepare() then Step then PrepareNext until end of aSQL
- Close is always called internaly
- 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);

*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 ""\uFFFF0base64encodedbinary" 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*

**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:** TSQLStatementCachedDynArray;

*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*

**function** Prepare(**const** GenericSQL: RawUTF8): PSQLRequest;

*Add or retrieve a generic SQL (with ? parameters) statement from cache*

```
procedure Init(aDB: TSQLite3DB);
```

*Intialize the cache*

```
procedure ReleaseAllDBStatements;
```

*Used internaly to release all prepared statements from Cache[]*

```
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\_v2() 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\_v2() 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(TObject)**

*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 832).*

**constructor** Create(const aFileName: TFileName; const aPassword: RawUTF8='');

*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)
- SYSTEMNOCASE collation is added (our custom fast UTF-8 case insensitive compare, 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
- initialize a TRTLCriticalSection 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*

- don't use the experimental SQLite Online Backup API
- database is closed, VACCUUMed, copied, then reopened: it's very fast

**function** Blob(const DBName, TableName, ColumnName: RawUTF8; RowID: Int64=-1; 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 DBName; in other words, the same BLOB that would be selected by:  

```
SELECT ColumnName FROM DBName.TableName WHERE rowid = RowID;
```
- use after a TSQLRequest.BindZero() to reserve Blob memory
- if RowID=-1, then the last inserted RowID is used
- will raise an ESQLite3Exception on any error

**function** DBOpen: integer;

*(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

**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** 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** 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 ""\uFFFF0base64encodedbinary""

**function** ExecuteNoException(const aSQL: RawUTF8): boolean; overload;

*Execute one SQL statements in aSQL UTF-8 encoded string*

- can be prepared with TransactionBegin()
- raise no Exception on error, but returns FALSE in such case

**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 TRTLCriticalSection: 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 TRTLCriticalSection: 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



**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** DBClose;

*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

**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: Int64); 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; out ID: RawUTF8); 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** ExecuteNoException(const aSQL: RawUTF8; out ID: RawUTF8); overload;

*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

**procedure** ExecuteNoException(const aSQL: RawUTF8; out ID: Int64); overload;

*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

**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(**const** aSQL: RawUTF8);

*Enter the TRTLCriticalSection: called before any DB access*

- provide the SQL statement about to be executed: handle proper caching
- is the SQL statement is void, assume a SELECT statement (no cache flush)

**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(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: 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 TRTLCriticalSection: called after any DB access*

**procedure** UnLockJSON(const aJSONResult: RawUTF8; aResultCount: PtrInt);

*Leave the TRTLCriticalSection: 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** 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** 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** 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** Log: TSynLog **read** fLog;

*Access to the Log instance associated with this SQLite3 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

**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 incrementally*

- 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 832).*

**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:*

*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

```
property Handle: TSQLite3Blob read fBlob;
```

*Read-only access to the BLOB object handle*

### Types implemented in the *SynSQLite3* unit:

```
TOnSQLStoredProc = procedure(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 optionnaly 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:

| <b>const</b>               | <b>zTab</b>  | <b>zCol</b> |
|----------------------------|--------------|-------------|
| SQLITE_CREATE_INDEX        | Index Name   | Table Name  |
| SQLITE_CREATE_TABLE        | Table Name   | nil         |
| SQLITE_CREATE_TEMP_INDEX   | Index Name   | Table Name  |
| SQLITE_CREATE_TEMP_TABLE   | Table Name   | nil         |
| SQLITE_CREATE_TEMP_TRIGGER | Trigger Name | Table Name  |
| SQLITE_CREATE_TEMP_VIEW    | View Name    | nil         |
| SQLITE_CREATE_TRIGGER      | Trigger Name | Table Name  |
| SQLITE_CREATE_VIEW         | View Name    | nil         |
| SQLITE_DELETE              | Table Name   | nil         |
| SQLITE_DROP_INDEX          | Index Name   | Table Name  |
| SQLITE_DROP_TABLE          | Table Name   | nil         |
| SQLITE_DROP_TEMP_INDEX     | Index Name   | Table Name  |
| SQLITE_DROP_TEMP_TABLE     | Table Name   | nil         |
| SQLITE_DROP_TEMP_TRIGGER   | Trigger Name | Table Name  |
| SQLITE_DROP_TEMP_VIEW      | View Name    | nil         |
| SQLITE_DROP_TRIGGER        | Trigger Name | Table Name  |
| SQLITE_DROP_VIEW           | View Name    | nil         |
| SQLITE_INSERT              | Table Name   | nil         |

|                      |               |                |
|----------------------|---------------|----------------|
| SQLITE_PRAGMA        | Pragma Name   | 1st arg or nil |
| SQLITE_READ          | Table Name    | Column Name    |
| SQLITE_SELECT        | nil           | nil            |
| SQLITE_TRANSACTION   | Operation     | nil            |
| SQLITE_UPDATE        | Table Name    | Column Name    |
| SQLITE_ATTACH        | Filename      | nil            |
| SQLITE_DETACH        | Database Name | nil            |
| SQLITE_ALTER_TABLE   | Database Name | Table Name     |
| SQLITE_REINDEX       | Index Name    | nil            |
| SQLITE_ANALYZE       | Table Name    | nil            |
| SQLITE_CREATE_VTABLE | Table Name    | Module Name    |
| SQLITE_DROP_VTABLE   | Table Name    | Module Name    |
| SQLITE_FUNCTION      | nil           | Function Name  |
| SQLITE_SAVEPOINT     | Operation     | Savepoint Name |

- 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.

**TSQLEasyHandler = 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.

**TSQLEasyFunc = 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

**TSQLEasyCallback = 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.

```
TSQLErrorBaseTransactionBehaviour = ( tbDeferred, tbImmediate, tbExclusive );
```

*TSQLErrorBase.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 tbExclusive 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.

```
TSQLErrorDestroyPtr = 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

```
TSQLErrorFunctionFinal = procedure(Context: TSQLErrorFunctionContext); cdecl;
```

*SQLite3 user final aggregate callback prototype*

```
TSQLErrorFunctionFunc = procedure(Context: TSQLErrorFunctionContext; argc: integer; var  
argv: TSQLErrorValueArray); 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

```
TSQLError3Blob = type PtrUInt;
```

*Internally store the SQLite3 blob handle*

*Used for DI-2.2.1 (page 832).*

```
TSQLError3DB = type PtrUInt;
```

*Internally store the SQLite3 database handle*

*Used for DI-2.2.1 (page 832).*

```
TSQLError3FunctionContext = type PtrUInt;
```

*Internal 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 TSQLError3FunctionContext type
- A pointer to an sqlite3\_context object is always first parameter to application-defined SQL functions, i.e. a TSQLErrorFunctionFunc prototype

*Used for DI-2.2.1 (page 832).*



**TSQLite3Statement = type PtrUInt;**

*Internally 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 832).*

**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 832).*

**TSQLite3ValueArray = array[0..63] of TSQLite3Value;**

*Internally store any array of SQLite3 value*

*Used for DI-2.2.1 (page 832).*

**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(TraceArg: Pointer; Trace: PUTF8Char); cdecl;**

*Callback function registered by `sqlite3_trace()`*

- this procedure will be invoked at various times when an SQL statement is being run by `sqlite3_step()`

**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:**

`SQL EncryptTableSize = $4000;`

*Encryption XOR mask table size (in bytes)*

- must be a power of 2
- bigger encryption table makes stronger encryption, but use more memory
- it's faster when the mask table can stay in the CPU L1 cache
- default size is therefore 16KB

`SQLITE_ABORT = 4;`

*Sqlite\_exec() return code: Callback routine requested an abort*

`SQLITE_ANY = 5;`

*Sqlite3\_create\_function\_v2 don't care about text encoding*

`SQLITE_AUTH = 23;`

*Sqlite\_exec() return code: Authorization denied*

`SQLITE_BLOB = 4;`

*Internal SQLite3 type as Blob*

`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\_FLOAT = 2;  
*Internal SQLite3 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 SQLite3 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\_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\_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 readonly 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

`SQLITE_TEXT = 3;`

*Internal SQLite3 type as Text*

`SQLITE_TOOBIG = 18;`

*Sqlite\_exec() return code: Too much data for one row of a table*

`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

`SQLITE_UTF16 = 4;`

*Text is UTF-16 encoded, using the system native byte order*

`SQLITE_UTF16BE = 3;`

*Text is UTF-16 BE encoded*

`SQLITE_UTF16LE = 2;`

*Text is UTF-16 LE encoded*

`SQLITE_UTF16_ALIGNED = 8;`

*Used by sqlite3\_create\_collation() only*

`SQLITE_UTF8 = 1;`

*Text is UTF-8 encoded*

`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:

| Functions or procedures           | Description                                                                     | Page |
|-----------------------------------|---------------------------------------------------------------------------------|------|
| ChangeSQLEncryptTable<br>PassWord | Use this procedure to change the password for an existing SQLite3 database file | 538  |

| Functions or procedures      | Description                                                                                             | Page |
|------------------------------|---------------------------------------------------------------------------------------------------------|------|
| CreateSQLEncryptTable        | You can use this simple (and strong enough) procedure for easy SQL encryption                           | 539  |
| ErrorWrongNumberOfArgs       | Wrapper around sqlite3_result_error() to be called if wrong number of arguments                         | 539  |
| IsSQLite3File                | Check from the file beginning if sounds like a valid SQLite3 file                                       | 539  |
| IsSQLite3FileEncrypted       | Check if sounds like an encrypted SQLite3 file                                                          | 539  |
| sqlite3InternalFree          | An internal function which calls Freemem(p)                                                             | 539  |
| sqlite3InternalFreeObject    | An internal function which calls TObject(p).Free                                                        | 539  |
| sqlite3_aggregate_context    | Implementations of aggregate SQL functions use this routine to allocate memory for storing their state. | 540  |
| sqlite3_bind_blob            | Bind a Blob Value to a parameter of a prepared statement                                                | 540  |
| sqlite3_bind_double          | Bind a floating point Value to a parameter of a prepared statement                                      | 540  |
| sqlite3_bind_Int             | Bind a 32 bits Integer Value to a parameter of a prepared statement                                     | 541  |
| sqlite3_bind_Int64           | Bind a 64 bits Integer Value to a parameter of a prepared statement                                     | 541  |
| sqlite3_bind_null            | Bind a NULL Value to a parameter of a prepared statement                                                | 541  |
| sqlite3_bind_parameter_count | Number Of SQL Parameters for a prepared statement                                                       | 541  |
| sqlite3_bind_text            | Bind a Text Value to a parameter of a prepared statement                                                | 541  |
| sqlite3_bind_zeroblob        | Bind a ZeroBlob buffer to a parameter                                                                   | 542  |
| sqlite3_blob_bytes           | Return The Size Of An Open BLOB                                                                         | 542  |
| sqlite3_blob_close           | Close A BLOB Handle                                                                                     | 542  |
| sqlite3_blob_open            | Open a BLOB For Incremental I/O                                                                         | 542  |
| sqlite3_blob_read            | Read Data From a BLOB Incrementally                                                                     | 542  |
| sqlite3_blob_write           | Write Data To a BLOB Incrementally                                                                      | 542  |
| sqlite3_busy_handler         | Register A Callback To Handle SQLITE_BUSY Errors                                                        | 542  |
| sqlite3_busy_timeout         | Set A Busy Timeout                                                                                      | 543  |
| sqlite3_changes              | Count The Number Of Rows Modified                                                                       | 543  |
| sqlite3_check                | Test the result state of a sqlite3_*( ) function                                                        | 543  |
| sqlite3_clear_bindings       | Reset All Bindings On A Prepared Statement                                                              | 543  |

| Functions or procedures    | Description                                                                                                                                                                          | Page |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| sqlite3_close              | Destructor for the sqlite3 object, which handle is DB                                                                                                                                | 543  |
| sqlite3_column_blob        | Converts the Col column in the current row of prepared statement S into a BLOB and then returns a pointer to the converted value                                                     | 543  |
| sqlite3_column_bytes       | Number of bytes for a BLOB or UTF-8 string result                                                                                                                                    | 544  |
| sqlite3_column_count       | Number of columns in the result set for the statement                                                                                                                                | 544  |
| sqlite3_column_decltype    | Returns a zero-terminated UTF-8 string containing the declared datatype of a result column                                                                                           | 544  |
| sqlite3_column_double      | Converts the Col column in the current row prepared statement S into a floating point value and returns a copy of that value                                                         | 544  |
| sqlite3_column_int         | Converts the Col column in the current row prepared statement S into a 32 bit integer value and returns a copy of that value                                                         | 544  |
| sqlite3_column_int64       | Converts the Col column in the current row prepared statement S into a 64 bit integer value and returns a copy of that value                                                         | 544  |
| sqlite3_column_name        | Returns the name of a result column as a zero-terminated UTF-8 string                                                                                                                | 544  |
| sqlite3_column_text        | 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                                             | 544  |
| sqlite3_column_type        | Datatype code for the initial data type of a result column                                                                                                                           | 545  |
| sqlite3_column_value       | Get the value handle of the Col column in the current row of prepared statement S                                                                                                    | 545  |
| sqlite3_commit_hook        | Register Commit Notification Callbacks                                                                                                                                               | 545  |
| sqlite3_context_db_handle  | Returns a copy of the pointer to the database connection (the 1st parameter) of the sqlite3_create_function_v2() routine that originally registered the application defined function | 545  |
| sqlite3_create_collation   | Define New Collating Sequences                                                                                                                                                       | 545  |
| sqlite3_create_function_v2 | Function creation routine used to add SQL functions or aggregates or to redefine the behavior of existing SQL functions or aggregates                                                | 546  |
| sqlite3_create_module_v2   | Used to register a new virtual table module name                                                                                                                                     | 546  |
| sqlite3_declare_vtab       | Declare the Schema of a virtual table                                                                                                                                                | 547  |
| sqlite3_errmsg             | Returns English-language text that describes an error, using UTF-8 encoding (which, with English text, is the same as Ansi).                                                         | 547  |
| sqlite3_exec               | One-Step Query Execution Interface                                                                                                                                                   | 547  |
| sqlite3_finalize           | Delete a previously prepared statement                                                                                                                                               | 547  |

| Functions or procedures   | Description                                                                                                                          | Page |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------|------|
| sqlite3_free              | Releases memory previously returned by sqlite3_malloc() or sqlite3_realloc()                                                         | 547  |
| sqlite3_free_table        | Release memory allocated for a sqlite3_get_table() result                                                                            | 547  |
| sqlite3_get_table         | Convenience Routines For Running Queries                                                                                             | 547  |
| sqlite3_initialize        | Initialize the SQLite3 database code                                                                                                 | 547  |
| sqlite3_last_insert_rowid | Returns the rowid of the most recent successful INSERT into the database                                                             | 548  |
| sqlite3_libversion        | Return the version of the SQLite database engine, in ascii format                                                                    | 548  |
| sqlite3_malloc            | Returns a pointer to a block of memory at least N bytes in length                                                                    | 548  |
| sqlite3_memory_highwater  | Returns the maximum value of sqlite3_memory_used() since the high-water mark was last reset                                          | 548  |
| sqlite3_memory_used       | Returns the number of bytes of memory currently outstanding (malloced but not freed)                                                 | 548  |
| sqlite3_next_stmt         | Find the next prepared statement                                                                                                     | 548  |
| sqlite3_open              | Open a SQLite3 database filename, creating a DB handle                                                                               | 548  |
| sqlite3_prepare_v2        | Compile a SQL query into byte-code                                                                                                   | 549  |
| sqlite3_realloc           | Attempts to resize a prior memory allocation                                                                                         | 549  |
| sqlite3_reset             | Reset a prepared statement object back to its initial state, ready to be re-Prepared                                                 | 549  |
| sqlite3_result_blob       | Sets the result from an application-defined function to be the BLOB                                                                  | 549  |
| sqlite3_result_double     | Sets the result from an application-defined function to be a floating point value specified by its 2nd argument                      | 549  |
| sqlite3_result_error      | Cause the implemented SQL function to throw an exception                                                                             | 549  |
| sqlite3_result_int64      | Sets the return value of the application-defined function to be the 64-bit signed integer value given in the 2nd argument            | 550  |
| sqlite3_result_null       | Sets the return value of the application-defined function to be NULL                                                                 | 550  |
| sqlite3_result_text       | Sets the return value of the application-defined function to be a text string which is represented as UTF-8                          | 550  |
| sqlite3_result_value      | Sets the result of the application-defined function to be a copy the unprotected sqlite3_value object specified by the 2nd parameter | 550  |
| sqlite3_rollback_hook     | Register Rollback Notification Callbacks                                                                                             | 550  |
| sqlite3_set_authorizer    | Registers an authorizer callback to a specified DB connection                                                                        | 550  |

| Functions or procedures    | Description                                                                                                                                                                                | Page |
|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| sqlite3_shutdown           | Shutdown the SQLite3 database core                                                                                                                                                         | 551  |
| sqlite3_step               | Evaluate An SQL Statement, returning a result status:                                                                                                                                      | 551  |
| sqlite3_stmt_readonly      | Returns true (non-zero) if and only if the prepared statement X makes no direct changes to the content of the database file                                                                | 551  |
| sqlite3_total_changes      | Total Number Of Rows Modified                                                                                                                                                              | 551  |
| sqlite3_trace              | Register callback function that can be used for tracing the execution of SQL statements                                                                                                    | 552  |
| sqlite3_update_hook        | Register Data Change Notification Callbacks                                                                                                                                                | 552  |
| sqlite3_user_data          | Returns a copy of the pointer that was the pUserData parameter (the 5th parameter) of the sqlite3_create_function_v2() routine that originally registered the application defined function | 552  |
| sqlite3_value_blob         | Converts a sqlite3_value object, specified by its handle, into a blob memory, and returns a copy of that value                                                                             | 552  |
| sqlite3_value_bytes        | Number of bytes for a sqlite3_value object, specified by its handle                                                                                                                        | 552  |
| sqlite3_value_double       | Converts a sqlite3_value object, specified by its handle, into a floating point value and returns a copy of that value                                                                     | 552  |
| sqlite3_value_int64        | Converts a sqlite3_value object, specified by its handle, into an integer value and returns a copy of that value                                                                           | 553  |
| sqlite3_value_numeric_type | Attempts to apply numeric affinity to the value                                                                                                                                            | 553  |
| sqlite3_value_text         | Converts a sqlite3_value object, specified by its handle, into an UTF-8 encoded string, and returns a copy of that value                                                                   | 553  |
| sqlite3_value_type         | Datatype code for a sqlite3_value object, specified by its handle                                                                                                                          | 553  |

**procedure** ChangeSQLEncryptTablePassword(**const** FileName: TFileName; **const** OldPassword, NewPassword: RawUTF8);

*Use this procedure to change the password for an existing SQLite3 database file*

- conversion is done in-place, therefore this procedure can handle very big files
- the OldPassword must be correct, otherwise the resulting file will be corrupted
- any password can be "" to mark no encryption
- you may use instead SynCrypto unit for more secure SHA-256 and AES-256 algos
- please note that this encryption is compatible only with SQLite3 files using the default page size of 1024

**procedure** CreateSQLEncryptTable(**const** Password: RawUTF8);

*You can use this simple (and strong enough) procedure for easy SQL encryption*

- usage is global for ALL SQLite3 databases access
- specify an ascii or binary password
- a buffer is allocated and initialized in SQLEncryptTable
- call with Password="" to end up encryption
- you may use instead SynCrypto unit for more secure SHA-256 and AES-256 algos
- please note that this encryption is compatible only with SQLite3 files using the default page size of 1024

**procedure** ErrorWrongNumberOfArgs(Context: TSQLite3FunctionContext);

*Wrapper around sqlite3\_result\_error() to be called if wrong number of arguments*

**function** IsSQLite3File(**const** FileName: TFileName): boolean;

*Check from the file beginning if sounds like a valid SQLite3 file*

- since encryption starts only with the 2nd page, this function will return true if a database file is encrypted or not

**function** IsSQLite3FileEncrypted(**const** FileName: TFileName): boolean;

*Check if sounds like an encrypted SQLite3 file*

- this will check the 2nd file page beginning to be a valid B-TREE page
- in some cases, may return false negatives (depending on the password used)

**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

**function** sqlite3\_aggregate\_context(Context: TSQLite3FunctionContext; nBytes: integer): pointer; cdecl; external;

*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 sqlite3\_aggregate\_context(C,N) routine returns a NULL pointer if N is less than or equal to zero or if a memory allocate error occurs.

- 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.

**function** sqlite3\_bind\_blob(S: TSQLite3Statement; Param: integer; Buf: pointer; Buf\_bytes: integer; DestroyPtr: TSQLDestroyPtr=SQLITE\_TRANSIENT): integer; cdecl; external;

*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()

**function** sqlite3\_bind\_double(S: TSQLite3Statement; Param: integer; Value: double): integer; cdecl; external;

*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



**function** sqlite3\_bind\_Int(S: TSQLite3Statement; Param: integer; Value: integer): integer; cdecl; external;

*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

**function** sqlite3\_bind\_Int64(S: TSQLite3Statement; Param: integer; Value: Int64): integer; cdecl; external;

*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

**function** sqlite3\_bind\_null(S: TSQLite3Statement; Param: integer): integer; cdecl; external;

*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)

**function** sqlite3\_bind\_parameter\_count(S: TSQLite3Statement): integer; cdecl; external;

*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 are used, there may be gaps in the list.

**function** sqlite3\_bind\_text(S: TSQLite3Statement; Param: integer; Text: PUTF8Char; Text\_bytes: integer=-1; DestroyPtr: TSQLDestroyPtr=SQLITE\_TRANSIENT): integer; cdecl; external;

*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()

```
function sqlite3_bind_zeroblob(S: TSQlite3Statement; Param: integer; Size: integer): integer; cdecl; external;
```

*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.
- 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

```
function sqlite3_blob_bytes(Blob: TSQlite3Blob): Integer; cdecl; external;
```

*Return The Size Of An Open BLOB*

```
function sqlite3_blob_close(Blob: TSQlite3Blob): Integer; cdecl; external;
```

*Close A BLOB Handle*

```
function sqlite3_blob_open(DB: TSQlite3DB; DBName, TableName, ColumnName: PUTF8Char; RowID: Int64; Flags: Integer; var Blob: TSQlite3Blob): Integer; cdecl; external;
```

*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;

```
function sqlite3_blob_read(Blob: TSQlite3Blob; const Data; Count, Offset: Integer): Integer; cdecl; external;
```

*Read Data From a BLOB Incrementally*

```
function sqlite3_blob_write(Blob: TSQlite3Blob; const Data; Count, Offset: Integer): Integer; cdecl; external;
```

*Write Data To a BLOB Incrementally*

```
function sqlite3_busy_handler(DB: TSQlite3DB; CallbackPtr: TSQlBusyHandler; user: Pointer): integer; cdecl; external;
```

*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 NULL, then SQLITE\_BUSY or SQLITE\_IOERR\_BLOCKED is returned immediately upon encountering the lock. If the busy callback is not NULL, then the callback might be invoked with two arguments.
- The default busy callback is NULL.

```
function sqlite3_busy_timeout(DB: TSQLite3DB; Milliseconds: integer): integer;  
cdecl; external;
```

*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.

```
function sqlite3_changes(DB: TSQLite3DB): Integer; cdecl; external;
```

*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.

```
function sqlite3_check(DB: TSQLite3DB; aResult: integer): integer;
```

*Test the result state of a sqlite3\_\*(\*) function*

- raise a `ESQLite3Exception` if the result state is an error
- return the result state otherwise (`SQLITE_OK`, `SQLITE_ROW`, `SQLITE_DONE` e.g.)

```
function sqlite3_clear_bindings(S: TSQLite3Statement): integer; cdecl; external;
```

*Reset All Bindings On A Prepared Statement*

```
function sqlite3_close(DB: TSQLite3DB): integer; cdecl; external;
```

*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

```
function sqlite3_column_blob(S: TSQLite3Statement; Col: integer): PAnsiChar;  
cdecl; external;
```

*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

```
function sqlite3_column_bytes(S: TSQLite3Statement; Col: integer): integer;  
cdecl; external;
```

*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

```
function sqlite3_column_count(S: TSQLite3Statement): integer; cdecl; external;
```

*Number of columns in the result set for the statement*

```
function sqlite3_column_decltype(S: TSQLite3Statement; Col: integer): PAnsiChar;  
cdecl; external;
```

*Returns a zero-terminated UTF-8 string containing the declared datatype of a result column*

```
function sqlite3_column_double(S: TSQLite3Statement; Col: integer): double;  
cdecl; external;
```

*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

```
function sqlite3_column_int(S: TSQLite3Statement; Col: integer): integer; cdecl; external;
```

*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

```
function sqlite3_column_int64(S: TSQLite3Statement; Col: integer): int64; cdecl; external;
```

*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

```
function sqlite3_column_name(S: TSQLite3Statement; Col: integer): PUTF8Char;  
cdecl; external;
```

*Returns the name of a result column as a zero-terminated UTF-8 string*

```
function sqlite3_column_text(S: TSQLite3Statement; Col: integer): PUTF8Char;  
cdecl; external;
```

*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

**function** sqlite3\_column\_type(S: TSQLite3Statement; Col: integer): integer; cdecl; external;

*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

**function** sqlite3\_column\_value(S: TSQLite3Statement; Col: integer): TSQLite3Value; cdecl; external;

*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

**function** sqlite3\_commit\_hook(DB: TSQLite3DB; xCallback: TSQLCommitCallback; pArg: Pointer): Pointer; cdecl; external;

*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(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.

**function** sqlite3\_context\_db\_handle(Context: TSQLite3FunctionContext): TSQLite3DB; cdecl; external;

*Returns a copy of the pointer to the database connection (the 1st parameter) of the sqlite3\_create\_function\_v2() routine that originally registered the application defined function*

**function** sqlite3\_create\_collation(DB: TSQLite3DB; CollationName: PUTF8Char; StringEncoding: integer; CollateParam: pointer; cmp: TSQLCollateFunc): integer; cdecl; external;

*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

```
function sqlite3_create_function_v2(DB: TSQLite3DB; FunctionName: PUTF8Char;  
nArg, eTextRep: integer; pApp: pointer; xFunc, xStep: TSQLFunctionFunc; xFinal:  
TSQLFunctionFinal; xDestroy: TSQLDestroyPtr): Integer; cdecl; external;
```

*Function creation routine used to 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 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.
- If the tenth parameter is not NULL, 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 is passed a single argument which is a copy of the pointer which was the fifth parameter to `sqlite3_create_function_v2()`.
- 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.

```
function sqlite3_create_module_v2(DB: TSQLite3DB; const zName: PAnsiChar; var p:  
TSQLite3Module; pClientData: Pointer; xDestroy: TSQLDestroyPtr): Integer; cdecl;  
external;
```

*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



```
function sqlite3_declare_vtab(DB: TSQLite3DB; const zSQL: PAnsiChar): Integer;  
cdecl; external;
```

*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.

```
function sqlite3_errmsg(DB: TSQLite3DB): PAnsiChar; cdecl; external;
```

*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.

```
function sqlite3_exec(DB: TSQLite3DB; SQL: PUTF8Char; Callback, Args: pointer;  
Error: PUTF8Char): integer; cdecl; external;
```

*One-Step Query Execution Interface*

```
function sqlite3_finalize(S: TSQLite3Statement): integer; cdecl; external;
```

*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

```
procedure sqlite3_free(p: Pointer); cdecl; external;
```

*Releases memory previously returned by sqlite3\_malloc() or sqlite3\_realloc()*

- should call native free() function, i.e. FreeMem() in this unit

```
procedure sqlite3_free_table(Table: PPUTF8CharArray); cdecl; external;
```

*Release memory allocated for a sqlite3\_get\_table() result*

```
function sqlite3_get_table(DB: TSQLite3DB; SQL: PUTF8Char; var Table:  
PPUTF8CharArray; var ResultRow, ResultCol: integer; var Error: PUTF8Char):  
integer; cdecl; external;
```

*Convenience Routines For Running Queries*

- fill Table with all Row/Col for the SQL query
- use sqlite3\_free\_table() to release memory

```
function sqlite3_initialize: integer; cdecl; external;
```

*Initialize the SQLite3 database code*

- automatically called by the initialization block of this unit
- so sqlite3.c is compiled with SQLITE\_OMIT\_AUTOINIT defined

**function** sqlite3\_last\_insert\_rowid(DB: TSQLite3DB): Int64; cdecl; external;

*Returns the rowid of the most recent successful INSERT into the database*

**function** sqlite3\_libversion: PUTF8Char; cdecl; external;

*Return the version of the SQLite database engine, in ascii format*

- currently returns '3.7.14'

**function** sqlite3\_malloc(n: Integer): Pointer; cdecl; external;

*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

**function** sqlite3\_memory\_highwater(resetFlag: Integer): Int64; cdecl; external;

*Returns the maximum value of sqlite3\_memory\_used() since the high-water mark was last reset*

**function** sqlite3\_memory\_used: Int64; cdecl; external;

*Returns the number of bytes of memory currently outstanding (malloced but not freed)*

**function** sqlite3\_next\_stmt(DB: TSQLite3DB; S: TSQLite3Statement):

TSQLite3Statement; cdecl; external;

*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

**function** sqlite3\_open(filename: PUTF8Char; var DB: TSQLite3DB): integer; cdecl; external;

*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



```
function sqlite3_prepare_v2(DB: TSQLite3DB; SQL: PUTF8Char; SQL_bytes: integer;  
var S: TSQLite3Statement; var SQLtail: PUTF8Char): integer; cdecl; external;
```

*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

```
function sqlite3_realloc(pOld: Pointer; n: Integer): Pointer; cdecl; external;
```

*Attempts to resize a prior memory allocation*

- should call native realloc() function, i.e. ReallocMem() in this unit

```
function sqlite3_reset(S: TSQLite3Statement): integer; cdecl; external;
```

*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.

```
procedure sqlite3_result_blob(Context: TSQLite3FunctionContext; Value: Pointer;  
Value_bytes: Integer=0; DestroyPtr: TSQLDestroyPtr=SQLITE_TRANSIENT); cdecl;  
external;
```

*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

```
procedure sqlite3_result_double(Context: TSQLite3FunctionContext; Value: double);  
cdecl; external;
```

*Sets the result from an application-defined function to be a floating point value specified by its 2nd argument*

```
procedure sqlite3_result_error(Context: TSQLite3FunctionContext; Msg: PUTF8Char;  
MsgLen: integer=-1); cdecl; external;
```

*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

**procedure** sqlite3\_result\_int64(Context: TSQLite3FunctionContext; Value: Int64);  
**cdecl; external;**

*Sets the return value of the application-defined function to be the 64-bit signed integer value given in the 2nd argument*

**procedure** sqlite3\_result\_null(Context: TSQLite3FunctionContext); **cdecl; external;**

*Sets the return value of the application-defined function to be NULL*

**procedure** sqlite3\_result\_text(Context: TSQLite3FunctionContext; Value: PUTF8Char;  
 Value\_bytes: Integer=-1; DestroyPtr: TSQLDestroyPtr=SQLITE\_TRANSIENT); **cdecl;**  
**external;**

*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

**procedure** sqlite3\_result\_value(Context: TSQLite3FunctionContext; Value: TSQLite3Value); **cdecl; external;**

*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

**function** sqlite3\_rollback\_hook(DB: TSQLite3DB; xCallback: TSQLCommitCallback;  
 pArg: Pointer): Pointer; **cdecl; external;**

*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.

**function** sqlite3\_set\_authorizer(DB: TSQLite3DB; xAuth: TSQLAuthorizerCallback;  
 pUserData: Pointer): Integer; **cdecl; external;**

*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

**function** sqlite3\_shutdown: integer; cdecl; external;

*Shutdown the SQLite3 database core*

- automatically called by the finalization block of this unit

**function** sqlite3\_step(S: TSQLite3Statement): integer; cdecl; external;

*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 a 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 schame changed in a way that makes the statement no longer valid, as a fatal error.
- another specific error code is returned on fatal error

**function** sqlite3\_stmt\_readonly(S: TSQLite3Statement): boolean; cdecl; external;

*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.

**function** sqlite3\_total\_changes(DB: TSQLite3DB): Integer; cdecl; external;

*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.

**function** sqlite3\_trace(DB: TSQLite3DB; tCallback: TSQLiteTraceCallback; aUserData: Pointer): Pointer; cdecl; external;

*Register callback function that can be used for tracing the execution of SQL statements*

- The callback function registered by sqlite3\_trace() is invoked at various times when an SQL statement is being run by sqlite3\_step(). The sqlite3\_trace() callback is invoked with a UTF-8 rendering of the SQL statement text as the statement first begins executing. Additional sqlite3\_trace() callbacks might occur as each triggered subprogram is entered. The callbacks for triggers contain a UTF-8 SQL comment that identifies the trigger.

**function** sqlite3\_update\_hook(DB: TSQLite3DB; xCallback: TSQLiteUpdateCallback; pArg: pointer): pointer; cdecl; external;

*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.

**function** sqlite3\_user\_data(Context: TSQLite3FunctionContext): pointer; cdecl; external;

*Returns a copy of the pointer that was the pUserData parameter (the 5th parameter) of the sqlite3\_create\_function\_v2() 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

**function** sqlite3\_value\_blob(Value: TSQLite3Value): pointer; cdecl; external;

*Converts a sqlite3\_value object, specified by its handle, into a blob memory, and returns a copy of that value*

**function** sqlite3\_value\_bytes(Value: TSQLite3Value): integer; cdecl; external;

*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)

**function** sqlite3\_value\_double(Value: TSQLite3Value): double; cdecl; external;

*Converts a sqlite3\_value object, specified by its handle, into a floating point value and returns a copy of that value*

**function** sqlite3\_value\_int64(Value: TSQLite3Value): Int64; cdecl; external;

*Converts a sqlite3\_value object, specified by its handle, into an integer value and returns a copy of that value*

**function** sqlite3\_value\_numeric\_type(Value: TSQLite3Value): integer; cdecl; external;

*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

**function** sqlite3\_value\_text(Value: TSQLite3Value): PUTF8Char; cdecl; external;

*Converts a sqlite3\_value object, specified by its handle, into an UTF-8 encoded string, and returns a copy of that value*

**function** sqlite3\_value\_type(Value: TSQLite3Value): integer; cdecl; external;

*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

#### Variables implemented in the SynSQLite3 unit:

SQLEncryptTable: PByteArray = nil;

*In order to allow file encryption on disk, initialize this pointer with SQLEncryptTableSize bytes of XOR tables*

- you can use fixed or custom (SHA+AES) generated table
- using a fixed XOR table is very fast and provides strong enough encryption
- the first page (first 1024 bytes) is not encrypted, since its content (mostly zero) can be used to easily guess the beginning of the key
- if the key is not correct, a ESQLite3Exception will be raised with 'database disk image is malformed' (ErrorCode=SQLITE\_CORRUPT)
- this table is common to ALL files accessed by the database engine: you have maintain several XOR mask arrays, and set SQLEncryptTable before any sqlite3\*( ) call, to mix passwords or crypted and uncrypted databases (see ChangeSQLEncryptTablePassword() for multiple SQLEncryptTable use)
- please note that this encryption is compatible only with SQLite3 files using the default page size of 1024

SynSQLite3Log: TSynLogClass = TSynLog;

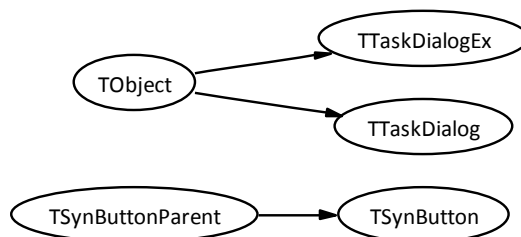
*The TSynLog class used for logging for all our SynSQLite3 related functions*

- you may override it with TSQLog, if available from SQLite3Commons
- since not all exceptions are handled specifically by this unit, you may better use a common TSynLog class for the whole application or module

#### 1.4.7.15. SynTaskDialog 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 1.17



*SynTaskDialog class hierarchy*

#### Objects implemented in the *SynTaskDialog* unit:

| Objects       | Description                                       | Page |
|---------------|---------------------------------------------------|------|
| TSynButton    | A generic Button to be used in the User Interface | 557  |
| TTaskDialog   | Implements a TaskDialog                           | 554  |
| TTaskDialogEx | A wrapper around the TTaskDialog.Execute method   | 556  |

**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,200);
  ShowMessage(IntToStr(Task.RadioRes)); // 200=Registry, 201=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
- 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)

**Content: string;**

*The dialog's primary content content text*

- any '\n' will be converted into a line feed

**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
- 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=tdiInformation;  
aFooterIcon: TTaskDialogFooterIcon=tdfiWarning; aRadioDef: integer=0; aWidth:  
integer=0; aParent: HWND=0; aNonNative: boolean=false; aEmulateClassicStyle:  
boolean = false): 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 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 @<http://synopse.info/forum/viewtopic.php?pid=2867#p2867>

**TTaskDialogEx = 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 fFlags 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*

**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

**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
```

```
TTaskDialogFlag =  
( tdfEnableHyperLinks, tdfUseHIconMain, tdfUseHIconFooter,  
tdfAllowDialogCancellation, tdfUseCommandLinks, tdfUseCommandLinksNoIcon,  
tdfExpandFooterArea, tdfExpandByDefault, tdfVerificationFlagChecked,  
tdfShowProgressBar, tdfShowMarqueeProgressBar, tdfCallbackTimer,  
tdfPositionRelativeToWindow, tdfRtlLayout, tdfNoDefaultRadioButton,  
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
```

```
TTaskDialogFlags = 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
```

```
TTaskDialogIcon =  
( tiBlank, tiWarning, tiQuestion, tiError, tiInformation, tiNotUsed, tiShield );  
The available main icons for the Task Dialog
```

### Functions or procedures implemented in the *SynTaskDialog* unit:

| Functions or procedures | Description                                       | Page |
|-------------------------|---------------------------------------------------|------|
| UnAmp                   | Return the text without the '&' characters within | 558  |

```
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

**TaskDialogIndirect: function(AConfig: pointer; Res: PInteger; ResRadio: PInteger; VerifyFlag: PBOOL): HRESULT; stdcall;**

*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)

#### 1.4.7.16. SynWinSock 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 1.6

**Objects implemented in the SynWinSock unit:**

| Objects    | Description | Page |
|------------|-------------|------|
| TIPv6_mreq |             | 559  |

**TIPv6\_mreq = record**

**ipv6mr\_interface: integer;**

*IPv6 multicast address.*

**padding: integer;**

*Interface index.*

**Constants implemented in the SynWinSock unit:**

**AI\_CANONNAME = \$2;**

*Socket address will be used in bind() call.*

**AI\_NUMERICHOST = \$4;**

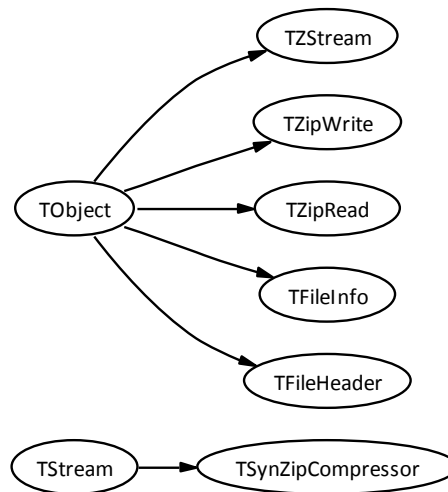
*Return canonical name in first ai\_canonname.*

**AI\_PASSIVE = \$1;**

#### 1.4.7.17. SynZip 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 1.17



*SynZip class hierarchy*

#### Objects implemented in the *SynZip* unit:

| Objects           | Description                                                       | Page |
|-------------------|-------------------------------------------------------------------|------|
| TFileHeader       | Directory file information structure, as used in .zip file format | 561  |
| TFileInfo         | Generic file information structure, as used in .zip file format   | 560  |
| TLastHeader       | Last header structure, as used in .zip file format                | 562  |
| TLocalFileHeader  | Internal file information structure, as used in .zip file format  | 562  |
| TSynZipCompressor | A simple TStream descendant for compressing data into a stream    | 562  |
| TZipEntry         | Stores an entry of a file inside a .zip archive                   | 563  |
| TZipRead          | Read-only access to a .zip archive file                           | 563  |
| TZipWrite         | Write-only access for creating a .zip archive file                | 564  |
| TZStream          | The internal memory structure as expected by the ZLib library     | 560  |

**TZStream = object(TObject)**

*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=stored 8=deflate 12=BZ2 14=LZMA*

zzipMethod: word;

0

zzipSize: 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;

02014b50 PK#1#2

```
function DataPosition: cardinal;  
  TLocalFileHeader
```

```
TLocalFileHeader = 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 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;  
  06054b50 PK#5#6
```

```
thisFiles: word;  
  0
```

```
totalFiles: word;  
  1
```

```
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** 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*

**property** SizeIn: cardinal **read** FStrm.total\_in;

*The number of byte written, i.e. the current uncompressed size*

**property** SizeOut: cardinal **read** FStrm.total\_out;

*The number of byte sent to the destination stream, i.e. the current compressed size*

**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*

**info:** PFileInfo;

*The information of this file, as stored in the .zip archive*

**storedName:** PAnsiChar;

*Name of the file inside the .zip archive*

- not ASCIIZ: length = info.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;

*The number of files inside a .zip archive*

**Entry:** array of TZipEntry;

*The files inside the .zip archive*

**constructor** Create(aFile: THandle; ZipStartOffset: cardinal=0; Size: cardinal=0); overload;

*Open a .zip archive file from its File Handle*

**constructor** Create(BufZip: pByteArray; Size: cardinal); overload;

*Open a .zip archive file directly from memory*

**constructor** Create(const aFileName: TFileName; ZipStartOffset: cardinal=0; Size: cardinal=0); overload;

*Open a .zip archive file as Read Only*

**constructor** Create(Instance: THandle; const ResName: string; ResType: PChar); overload;

*Open a .zip archive file directly from a resource*

**destructor** Destroy; override;

*Release associated memory*

**function** NameToIndex(const aName: TFileName): integer;

*Get the index of a file inside the .zip archive*

**function** UnZip(aIndex: integer; const DestDir: TFileName): boolean; overload;

*Uncompress a file stored inside the .zip archive into a destination directory*

**function** UnZip(aIndex: integer): RawByteString; overload;

*Uncompress a file stored inside the .zip archive into memory*

**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** intName: RawByteString; 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

**Handle: integer;**

*The associated file handle*

**constructor** Create(const aFileName: TFileName); overload;

*The file name, as stored in the .zip internal directory the corresponding file header initialize the .zip file*

- a new .zip file content is created

**constructor** CreateFrom(const aFileName: TFileName);

*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)



**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: 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** AddFromZip(**const** ZipEntry: TZipEntry);

*Add a file from an already compressed zip entry*

**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: RawByteString);

*Append a file content into the destination file*

- usefull to add the initial Setup.exe file, e.g.

#### Types implemented in the SynZip unit:

NativeUInt = cardinal;

*As available in newer Delphi versions*

PInteger = ^Integer;

*Delphi 5 doesn't have those base types defined :(*

RawByteString = AnsiString;

*Define RawByteString, as it does exist in Delphi 2009 and up*

- to be used for byte storage into an AnsiString

TSynZipCompressorFormat = ( szcfRaw, szcfZip, szcfGZ );

*The format used for storing data*

#### Functions or procedures implemented in the SynZip unit:

| Functions or procedures | Description                                              | Page |
|-------------------------|----------------------------------------------------------|------|
| Check                   | Low-level check of the code returned by the ZLib library | 566  |
| CompressDeflate         | (un)compress a data content using the Deflate algorithm  | 566  |
| CompressGZip            | (un)compress a data content using the gzip algorithm     | 566  |
| CompressMem             | In-memory ZLib DEFLATE compression                       | 566  |
| CompressStream          | ZLib DEFLATE compression from memory into a stream       | 566  |

| Functions or procedures | Description                                                                                | Page |
|-------------------------|--------------------------------------------------------------------------------------------|------|
| CompressString          | Compress some data, with a proprietary format (including CRC)                              | 566  |
| CRC32string             | Just hash aString with CRC32 algorithm                                                     | 566  |
| EventArchiveZip         | A TSynLogArchiveEvent handler which will compress older .log files into .zip archive files | 567  |
| GZRead                  | Uncompress a .gz file content                                                              | 567  |
| UnCompressMem           | In-memory ZLib INFLATE decompression                                                       | 567  |
| UnCompressStream        | ZLib INFLATE decompression from memory into a stream                                       | 567  |
| UncompressString        | Uncompress some data, with a proprietary format (including CRC)                            | 567  |

**function** Check(**const** Code: Integer; **const** ValidCodes: array of Integer): integer;  
*Low-level check of the code returned by the ZLib library*

**function** CompressDeflate(**var** Data: RawByteString; Compress: boolean): RawByteString;  
*(un)compress a data content using the Deflate algorithm*  
- as expected by THttpSocket.RegisterCompress  
- will use internally 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** CompressGZip(**var** Data: RawByteString; Compress: boolean): RawByteString;  
*(un)compress a data content using the gzip algorithm*  
- as expected by THttpSocket.RegisterCompress  
- will use internally 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; ZipFormat: Boolean=false) : integer;  
*In-memory ZLib DEFLATE compression*

**function** CompressStream(src: pointer; srcLen: integer; aStream: TStream; CompressionLevel: integer=6; ZipFormat: Boolean=false): cardinal;  
*ZLib DEFLATE compression from memory into a stream*

**function** CompressString(**const** data: RawByteString; failIfGrow: boolean = false; CompressionLevel: integer=6) : RawByteString;  
*Compress some data, with a proprietary format (including CRC)*

**function** CRC32string(**const** aString: RawByteString): cardinal;  
*Just hash aString with CRC32 algorithm*  
- crc32 is better than adler32 for short strings

**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** GZRead(gz: PAnsiChar; gzLen: integer): RawByteString;

*Uncompress a .gz file content*  
- return "" if the .gz content is invalid (e.g. bad crc)

**function** UnCompressMem(src, dst: pointer; srcLen, dstLen: integer) : integer;

*In-memory ZLib INFLATE decompression*

**function** UnCompressStream(src: pointer; srcLen: integer; aStream: TStream; checkCRC: PCardinal): cardinal;

*ZLib INFLATE decompression from memory into a stream*  
- return the number of bytes written into the stream  
- if checkCRC is not nil, it will contain thecrc32 (if aStream is nil, it will fast calculate the crc of the the uncompressed memory block)

**function** UncompressString(const data: RawByteString) : RawByteString;

*Uncompress some data, with a proprietary format (including CRC)*  
- return "" in case of a decompression failure

#### 1.4.7.18. SQLite3 unit

*Purpose:* SQLite3 embedded Database engine used as the kernel of mORMot

- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

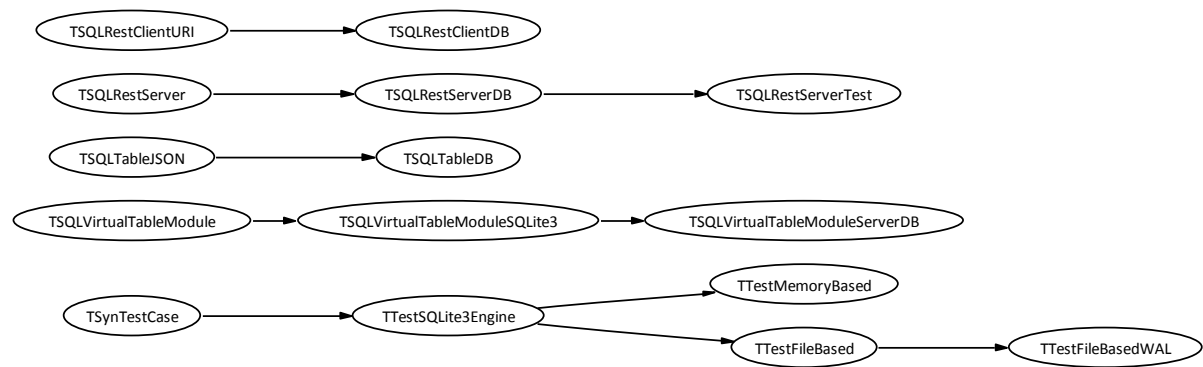
The **SQLite3** unit is quoted in the following items:

| SWRS #   | Description                                                                                                       | Page |
|----------|-------------------------------------------------------------------------------------------------------------------|------|
| DI-2.2.1 | The <i>SQLite3</i> engine shall be embedded to the framework                                                      | 832  |
| DI-2.2.2 | The framework libraries, including all its <i>SQLite3</i> related features, shall be tested using Unitary testing | 833  |

Units used in the **SQLite3** unit:

| Unit Name             | Description                                                                                                                                                               | Page |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                     | 575  |
| <i>SynCommons</i>     | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |

| Unit Name         | Description                                                                                                                                                                     | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynSQLite3</i> | SQLite3 embedded Database engine direct access<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17       | 502  |
| <i>SynZip</i>     | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 560  |



*SQLite3 class hierarchy*

#### Objects implemented in the *SQLite3* unit:

| Objects                        | Description                                                                                                                                                          | Page |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| TSQLRestClientDB               | REST client with direct access to a SQLite3 database                                                                                                                 | 571  |
| TSQLRestServerDB               | REST server with direct access to a SQLite3 database                                                                                                                 | 569  |
| TSQLRestServerTest             | This class defined two published methods of type TSQLRestServerCallBack in order to test the Server-Side ModelRoot/TableName/ID/MethodName RESTful model             | 572  |
| TSQLTableDB                    | Execute a SQL statement in the local SQLite3 database engine, and get result in memory                                                                               | 569  |
| TSQLVirtualTableModuleServerDB | Define a Virtual Table module for a TSQLRestServerDB SQLite3 engine                                                                                                  | 573  |
| TSQLVirtualTableModuleSQLite3  | Define a Virtual Table module for a stand-alone SQLite3 engine                                                                                                       | 572  |
| TTestFileBased                 | This test case will test most functions, classes and types defined and implemented in the SQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach   | 573  |
| TTestFileBasedWAL              | This test case will test most functions, classes and types defined and implemented in the SQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach   | 574  |
| TTestMemoryBased               | This test case will test most functions, classes and types defined and implemented in the SQLite3 unit, i.e. the SQLite3 engine itself, with a memory-based approach | 574  |

| Objects            | Description                                                                                                                                      | Page |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|------|
| TTestSQLite3Engine | A parent test case which will test most functions, classes and types defined and implemented in the SQLite3 unit, i.e. the SQLite3 engine itself | 573  |

#### **TSQLEngineDB = class(TSQLEngineJSON)**

*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 TSQLEngineJSON internally: faster than sqlite3\_get\_table() (less memory allocation/fragmentation) and allows efficient caching

**constructor** Create(aDB: TSQLEngine; **const** Tables: array of TClass; **const** aSQL: RawUTF8; Expand: boolean);

*Execute a SQL statement, and init TSQLEngine 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 TSQLEngineJSON 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

#### **TSQLEngineServerDB = class(TSQLEngineServer)**

*REST server with direct access to a SQLite3 database*

- caching is handled at TSQLEngine level
- SQL statements for record retrieval from ID are prepared for speed

*Used for DI-2.2.1 (page 832).*

**constructor** Create(aModel: TSQLEngine; aDB: TSQLEngine; aHandleUserAuthentication: boolean=false); overload; **virtual**;

*Initialize a REST server with a database*

- any needed TSQLEngineVirtualTable class should have been already registered via the RegisterVirtualTableModule() method

**constructor** Create(aModel: TSQLEngine; **const** aDBFileName: TFileName; aHandleUserAuthentication: boolean=false; **const** aPassword: RawUTF8=''); **reintroduce**; overload;

*Initialize a REST server with a database, by specifying its filename*

- TSQLEngineServerDB will initialize a owned TSQLEngine, 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

**destructor** Destroy; **override**;

*Close database and free used memory*

**function** Backup(Dest: TStream): boolean;

*Backup of the opened Database into an external stream (e.g. a file, compressed or not)*  
 - this method doesn't use the experimental SQLite Online Backup API (which restart the backup process on any write: so no good performance could be achieved on a working database: this method uses a faster lock + copy approach)  
 - database is closed, VACCUUMed, copied, then reopened

**function** BackupGZ(const DestFileName: TFileName; CompressionLevel: integer=2): boolean;

*Backup of the opened Database into a .gz compressed file*  
 - 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** EngineExecuteAll(const aSQL: RawUTF8): boolean; **override;**

*Overriden methods for direct sqlite3 database engine call*

**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

**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

**procedure** Commit(SessionID: cardinal=1); **override;**

*End a transaction (implements REST END Member)*  
 - write all pending SQL statements to the disk

**procedure** CreateMissingTables(user\_version: cardinal=0);

*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** FlushInternalIDBCache; **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. TSQLRestServerStaticExternal classes defined in SQLite3DB) could flush the database content without proper notification  
 - this overriden implementation will call TSQLDataBase.CacheFlush method

**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*

**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 832).*

**constructor** Create(aClientModel, aServerModel: TSQLModel; **const** aDBFileName: TFileName; aServerClass: TSQLRestServerDBClass; aHandleUserAuthentication: boolean=false; **const** aPassword: RawUTF8=''); **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*

**TSQLRestServerTest = class**(TSQLRestServerDB)

*This class defined two published methods of type TSQLRestServerCallBack in order to test the Server-Side ModelRoot/TableName/ID/MethodName RESTful model*

**function** DataAsHex(**var** aParams: TSQLRestServerCallBackParams): Integer;

*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

**function** Sum(**var** aParams: TSQLRestServerCallBackParams): Integer;

*Method used to test the Server-Side ModelRoot/Sum or ModelRoot/People/Sum Requests*

- 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

**TSQLVirtualTableModuleSQLite3 = class**(TSQLVirtualTableModule)

*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): TFileName; **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 ':memory:' (so 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 ':memory:'

**property** DB: TSQLDataBase read fDB write SetDB;

*The associated SQLite3 database connection*

- setting the DB property will initialize the virtual table module for this DB connection, calling the sqlite3\_create\_module\_v2() function



```
TSQLVirtualTableModuleServerDB = class(TSQLVirtualTableModuleSQLite3)
```

*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*

```
TTestSQLite3Engine = class(TSynTestCase)
```

*A parent test case which will test most functions, classes and types defined and implemented in the SQLite3 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 833).*

```
procedure DatabaseDirectAccess;
```

*Test direct access to the SQLite3 engine*

- i.e. via TSQLDataBase and TSQLRequest classes

```
procedure VirtualTableDirectAccess;
```

*Test direct access to the Virtual Table features of SQLite3*

```
procedure _TSQLRestClientDB;
```

*Test the TSQLRestClientDB, i.e. a local Client/Server driven usage of the framework*

- validates TSQLModel, TSQLRestServer and TSQLRestServerStatic 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  
- also test FTS implementation if INCLUDE\_FTS3 conditional is defined  
- 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 SQLite3 unit, i.e. the SQLite3 engine itself, with a file-based approach*

*Used for DI-2.2.2 (page 833).*

**TTestMemoryBased = class(TTestSQLite3Engine)**

*This test case will test most functions, classes and types defined and implemented in the SQLite3 unit, i.e. the SQLite3 engine itself, with a memory-based approach*

- this class will also test the TSQLRestServerStatic class, and its 100% Delphi simple database engine

*Used for DI-2.2.2 (page 833).*

**TTestFileBasedWAL = class(TTestFileBased)**

*This test case will test most functions, classes and types defined and implemented in the SQLite3 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 833).*

#### Functions or procedures implemented in the *SQLite3* unit:

| Functions or procedures    | Description                                                | Page |
|----------------------------|------------------------------------------------------------|------|
| RegisterVirtualTableModule | Initialize a Virtual Table Module for a specified database | 574  |
| VarDataFromValue           | Set a SQLite3 value into a TVarData                        | 574  |
| VarDataToContext           | Set a TVarData into a SQLite3 result context               | 575  |

**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)

**procedure** VarDataFromValue(Value: TSQLite3Value; var Res: TVarData);

*Set a SQLite3 value into a TVarData*

- only handle varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0]) types in TVarData  
- therefore it's not a true Variant typecast, only a way of data sharing e.g. for the TSQLVirtualTable.Insert or Update methods  
- will call the corresponding sqlite3\_value\_\*( ) function to retrieve the data with the less overhead (e.g. memory allocation or copy) as possible

**function** VarDataToContext(Context: TSQLite3FunctionContext; **const** Res: TVarData): boolean;

*Set a TVarData into a SQLite3 result context*

- only handle varNull, varInt64, varDouble, varString (mapping a constant UTF8Char), and varAny (BLOB with size = VLongs[0]) types in TVarData
- therefore it's not a true Variant typecast, only a way of data sharing e.g. for the TSQLVirtualTableCursor.Column method
- will call the corresponding sqlite3\_result\_\*() function and return true, or will return false if the TVarData type is not handled

#### 1.4.7.19. SQLite3Commons unit

*Purpose:* Common ORM and SOA classes

- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

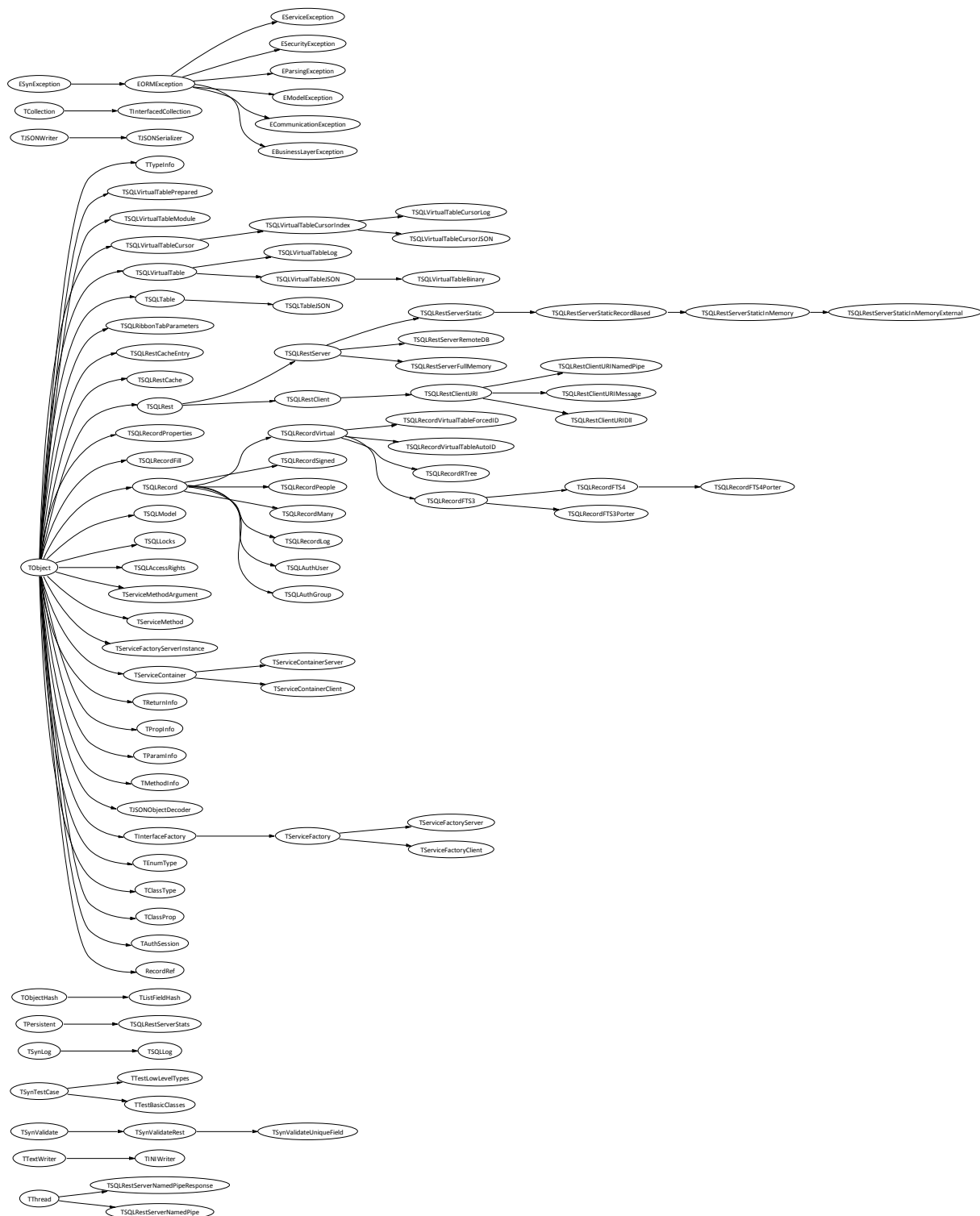
The *SQLite3Commons* unit is quoted in the following items:

| SWRS #       | Description                                                                                                                      | Page |
|--------------|----------------------------------------------------------------------------------------------------------------------------------|------|
| DI-2.1.1     | Client-Server framework                                                                                                          | 828  |
| DI-2.1.1.1   | RESTful framework                                                                                                                | 828  |
| DI-2.1.1.2.1 | In-Process communication                                                                                                         | 829  |
| DI-2.1.1.2.2 | Named Pipe protocol                                                                                                              | 829  |
| DI-2.1.1.2.3 | Windows Messages protocol                                                                                                        | 829  |
| DI-2.1.1.2.4 | HTTP/1.1 protocol                                                                                                                | 830  |
| DI-2.1.2     | UTF-8 JSON format shall be used to communicate                                                                                   | 830  |
| DI-2.1.3     | The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information) | 831  |
| DI-2.2.1     | The <i>SQLite3</i> engine shall be embedded to the framework                                                                     | 832  |
| DI-2.2.2     | The framework libraries, including all its <i>SQLite3</i> related features, shall be tested using Unitary testing                | 833  |

Units used in the *SQLite3Commons* unit:

| Unit Name         | Description                                                                                                                                                               | Page |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |

| Unit Name        | Description                                                                                                                                                                                                                                                                                                                                                                    | Page |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCrypto</i> | Fast cryptographic routines (hashing and cypher) <ul style="list-style-type: none"><li>- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms</li><li>- optimized for speed (tuned assembler and VIA PADLOCK optional support)</li><li>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17</li></ul> | 383  |
| <i>SynZip</i>    | Low-level access to ZLib compression (1.2.5 engine version) <ul style="list-style-type: none"><li>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17</li></ul>                                                                                                                                                   | 560  |



SQLite3Commons class hierarchy

## Objects implemented in the *SQLite3Commons* unit:

| Objects | Description | Page |
|---------|-------------|------|
|---------|-------------|------|

| Objects                 | Description                                                                                                                                 | Page |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|------|
| EBusinessLayerException | Exception raised in case of an error in project implementation logic                                                                        | 602  |
| ECommunicationException | Exception raised in case of a Client-Server communication error                                                                             | 602  |
| EModelException         | Exception raised in case of wrong Model definition                                                                                          | 602  |
| EORMException           | Generic parent class of all custom Exception types of this unit                                                                             | 602  |
| EParsingException       | Exception raised in case of unexpected parsing error                                                                                        | 602  |
| ESecurityException      | Exception raised in case of any authentication error                                                                                        | 602  |
| EServiceException       | Exception dedicated to interface based service implementation                                                                               | 602  |
| RecordRef               | Usefull object to type cast TRecordReference type value into explicit TSQLRecordClass and ID                                                | 631  |
| TAuthSession            | Class used to maintain in-memory sessions                                                                                                   | 668  |
| TClassProp              | A wrapper to published properties of a class                                                                                                | 591  |
| TClassType              | A wrapper to class type information, as defined by the Delphi RTTI                                                                          | 592  |
| TEnumType               | A wrapper to enumeration type information, as defined by the Delphi RTTI                                                                    | 593  |
| TINIWriter              | Simple writer to a Stream, specialized for writing an object as INI                                                                         | 601  |
| TInterfacedCollection   | Any TCollection used between client and server shall inherit from this class                                                                | 643  |
| TInterfaceFactory       | A common ancestor for any class needing interface RTTI                                                                                      | 643  |
| TJSONObjectDecoder      | Record/object helper to handle JSON object decoding                                                                                         | 590  |
| TJSONSerializer         | Simple writer to a Stream, specialized for writing an object as JSON                                                                        | 601  |
| TListFieldHash          | Class able to handle a O(1) hashed-based search of a property in a TList                                                                    | 681  |
| TMethodInfo             | A wrapper around a method definition                                                                                                        | 600  |
| TParamInfo              | A wrapper around an individual method parameter definition                                                                                  | 600  |
| TPropInfo               | A wrapper containing a property definition, with GetValue() and SetValue() functions for direct Delphi / UTF-8 SQL type mapping/conversion: | 595  |
| TReturnInfo             | A wrapper around method returned result definition                                                                                          | 599  |
| TServiceContainer       | A global services provider class                                                                                                            | 649  |
| TServiceContainerClient | A services provider class to be used on the client side                                                                                     | 650  |
| TServiceContainerServer | A services provider class to be used on the server side                                                                                     | 650  |

| Objects                        | Description                                                                                 | Page |
|--------------------------------|---------------------------------------------------------------------------------------------|------|
| TServiceCustomAnswer           | A record type to be used as result for a function method for custom content                 | 642  |
| TServiceFactory                | An abstract service provider, as registered in TServiceContainer                            | 643  |
| TServiceFactoryClient          | A service provider implemented on the client side                                           | 648  |
| TServiceFactoryServer          | A service provider implemented on the server side                                           | 645  |
| TServiceFactoryServer Instance | Server-side service provider uses this to store one internal instance                       | 711  |
| TServiceMethod                 | Describe a service provider method                                                          | 641  |
| TServiceMethodArgument         | Describe a service provider method argument                                                 | 640  |
| TServiceRunningContext         | Will identify the currently running service on the server side                              | 707  |
| TSQLAccessRights               | Set the User Access Rights, for each Table                                                  | 666  |
| TSQLAuthGroup                  | Table containing the available user access rights for authentication                        | 667  |
| TSQLAuthUser                   | Table containing the Users registered for authentication                                    | 668  |
| TSQLLocks                      | Used to store the locked record list, in a specified table                                  | 624  |
| TSQLLog                        | Logging class with enhanced RTTI                                                            | 709  |
| TSQLModel                      | A Database Model (in a MVC-driven way), for storing some tables types as TSQLRecord classes | 627  |
| TSQLQueryCustom                | Store one custom query parameters                                                           | 625  |
| TSQLRecord                     | Root class for defining and mapping database records                                        | 609  |
| TSQLRecordFill                 | Internal data used by TSQLRecord.FillPrepare()/FillPrepareMany() methods                    | 608  |
| TSQLRecordFTS3                 | A base record, corresponding to a FTS3 table, i.e. implementing full-text                   | 633  |
| TSQLRecordFTS3Porter           | This base class will create a FTS3 table using the Porter Stemming algorithm                | 634  |
| TSQLRecordFTS4                 | A base record, corresponding to a FTS4 table, which is an enhancement to FTS3               | 634  |
| TSQLRecordFTS4Porter           | This base class will create a FTS4 table using the Porter Stemming algorithm                | 634  |
| TSQLRecordLog                  | A base record, with a JSON-logging capability                                               | 639  |
| TSQLRecordMany                 | Handle "has many" and "has many through" relationships                                      | 635  |
| TSQLRecordPeople               | A record mapping used in the test classes of the framework                                  | 708  |

| Objects                         | Description                                                                                         | Page |
|---------------------------------|-----------------------------------------------------------------------------------------------------|------|
| TSQLRecordProperties            | Some information about a given TSQLRecord class properties                                          | 603  |
| TSQLRecordRTree                 | A base record, corresponding to an R-Tree table                                                     | 632  |
| TSQLRecordSigned                | Common ancestor for tables with digitally signed RawUTF8 content                                    | 639  |
| TSQLRecordVirtual               | Parent of all virtual classes                                                                       | 627  |
| TSQLRecordVirtualTableAutoID    | Record associated to Virtual Table implemented in Delphi, with ID generated automatically at INSERT | 707  |
| TSQLRecordVirtualTableForcedID  | Record associated to a Virtual Table implemented in Delphi, with ID forced at INSERT                | 707  |
| TSQLRest                        | A generic REpresentational State Transfer (REST) client/server class                                | 654  |
| TSQLRestCache                   | Implement a fast cache content at the TSQLRest level                                                | 652  |
| TSQLRestCacheEntry              | For TSQLRestCache, stores a table settings and values                                               | 651  |
| TSQLRestCacheEntryValue         | For TSQLRestCache, stores a table values                                                            | 651  |
| TSQLRestClient                  | A generic REpresentational State Transfer (REST) client                                             | 686  |
| TSQLRestClientURI               | A generic REpresentational State Transfer (REST) client with URI                                    | 690  |
| TSQLRestClientURIDll            | Rest client with remote access to a server through a dll                                            | 696  |
| TSQLRestClientURIMessage        | Rest client with remote access to a server through Windows messages                                 | 696  |
| TSQLRestClientURINamedPipe      | Rest client with remote access to a server through a Named Pipe                                     | 697  |
| TSQLRestServer                  | A generic REpresentational State Transfer (REST) server                                             | 669  |
| TSQLRestServerCallbackParams    | Store all parameters for a TSQLRestServerCallback event handler                                     | 607  |
| TSQLRestServerFullMemory        | A REST server using only in-memory tables                                                           | 685  |
| TSQLRestServerNamedPipe         | Server thread accepting connections from named pipes                                                | 664  |
| TSQLRestServerNamedPipeResponse | Server child thread dealing with a connection through a named pipe                                  | 665  |
| TSQLRestServerRemoteDB          | A REST server using a TSQLRestClient for all its ORM process                                        | 686  |
| TSQLRestServerSessionContext    | Used to store the current execution context of a remote request                                     | 607  |
| TSQLRestServerStatic            | REST server with direct access to an external database engine                                       | 679  |
| TSQLRestServerStaticInMemory    | REST server with direct access to a memory-stored database                                          | 685  |



| Objects                              | Description                                                                                                                             | Page |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|------|
| TSQLRestServerStaticInMemoryExternal | REST server with direct access to a memory database, to be used as external table                                                       | 685  |
| TSQLRestServerStaticRecordBased      | Abstract REST server exposing some internal TSQLRecord-based methods                                                                    | 680  |
| TSQLRestServerStats                  | If defined, the server statistics will contain precise working time process used for statistics update in TSQLRestServer.URI()          | 665  |
| TSQLRibbonTabParameters              | Defines the settings for a Tab for User Interface generation                                                                            | 625  |
| TSQLTable                            | Wrapper to an ORM result table, statically stored as UTF-8 text                                                                         | 582  |
| TSQLTableJSON                        | Get a SQL result from a JSON message, and store it into its own memory                                                                  | 589  |
| TSQLTableSortParams                  | Contains the parameters used for sorting                                                                                                | 582  |
| TSQLVirtualTable                     | Abstract class able to access a Virtual Table content                                                                                   | 701  |
| TSQLVirtualTableBinary               | A TSQLRestServerStaticInMemory-based virtual table using Binary storage                                                                 | 706  |
| TSQLVirtualTableCursor               | Abstract class able to define a Virtual Table cursor                                                                                    | 703  |
| TSQLVirtualTableCursorIndex          | A generic Virtual Table cursor associated to Current/Max index properties                                                               | 704  |
| TSQLVirtualTableCursorJSON           | A Virtual Table cursor for reading a TSQLRestServerStaticInMemory content                                                               | 704  |
| TSQLVirtualTableCursorLog            | A Virtual Table cursor for reading a TSynLogFile content                                                                                | 707  |
| TSQLVirtualTableJSON                 | A TSQLRestServerStaticInMemory-based virtual table using JSON storage                                                                   | 705  |
| TSQLVirtualTableLog                  | Implements a read/only virtual table able to access a .log file, as created by TSynLog                                                  | 706  |
| TSQLVirtualTableModule               | Parent class able to define a Virtual Table module                                                                                      | 700  |
| TSQLVirtualTablePrepared             | The WHERE and ORDER BY statements as set by TSQLVirtualTable.Prepare                                                                    | 699  |
| TSQLVirtualTablePreparedConstraint   | A WHERE constraint as set by the TSQLVirtualTable.Prepare() method                                                                      | 698  |
| TSQLVirtualTablePreparedOrderBy      | An ORDER BY clause as set by the TSQLVirtualTable.Prepare() method                                                                      | 699  |
| TSynValidateRest                     | Will define a validation to be applied to a TSQLRecord field, using if necessary an associated TSQLRest instance and a TSQLRecord class | 697  |
| TSynValidateUniqueField              | Will define a validation for a TSQLRecord Unique field                                                                                  | 698  |

| Objects                       | Description                                                                                                             | Page |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------|------|
| TTestBasicClasses             | This test case will test some generic classes defined and implemented in the SQLite3Commons unit                        | 708  |
| TTestLowLevelTypes            | This test case will test most low-level functions, classes and types defined and implemented in the SQLite3Commons unit | 708  |
| TTypeInfo                     | A wrapper containing type information definition                                                                        | 594  |
| TVirtualTableModuleProperties | Used to store and handle the main specifications of a TSQLVirtualTableModule                                            | 699  |

**TSQLTableSortParams = record**

*Contains the parameters used for sorting*

- FieldCount is 0 if it was never sorted
- used to sort data again after a successful data update with TSQLTableJSON.FillFrom()

**TSQLTable = class(TObject)**

*Wrapper to an ORM result table, statically stored as UTF-8 text*

- contain all result in memory, until destroyed
- first row contains the field names
- following rows contains the data itself
- GetA() or GetW() can be used in a TDrawString
- will be implemented as TSQLTableDB for direct SQLite3 database engine call, or as TSQLTableJSON for remote access through optimized JSON messages

*Used for DI-2.1.2 (page 830).*

**constructor** Create(const Tables: array of TClass; const aSQL: RawUTF8);

*Initialize the result table*

- you can use RecordClassesToClasses() wrapper function to convert an array of TSQLRecordClass into the expected array of TClass

**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** ExpandAsString(Row,Field: integer; Client: TObject; **out** Text: 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 and sftRecord/sftID
- 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)

**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** FieldIndex(const FieldName: shortstring): integer; overload;

*Get the Field index of a FieldName*

- return -1 if not found, index (0..FieldCount-1) if found

**function** FieldIndex(FieldName: PUTF8Char): integer; overload;

*Get the Field index of a FieldName*

- return -1 if not found, index (0..FieldCount-1) if found

**function** FieldIndexID: integer;

*Field index of a the 'ID' field, -1 if none*

**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** FieldTable(Field: integer): TClass;

*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; EnumTypeInfo: PPointer): TSQLFieldType;

*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** FieldValue(const FieldName: shortstring; Row: integer): PUTF8Char;

*Get the Field content (encoded as UTF-8 text) from a property name*

- return nil if not found

**function** Get(Row, Field: integer): PUTF8Char;

*Read-only access to a particular field value, as UTF-8 encoded text*

- points to memory buffer allocated by Init()

**function** GetA(Row, Field: integer): WinAnsiString;

*Read-only access to a particular field value, as Win Ansi text*

**function** GetAsInteger(Row, Field: integer): integer;

*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 ('\\uFFF0base64encodedbinary')
- 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 ('\\uFFF0base64encodedbinary')
- 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** 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** GetJSONValues(Expand: boolean): RawUTF8; overload;

*Same as above, but returning result into a RawUTF8*

*Used for DI-2.1.2 (page 830).*

**function** GetRowValues(Field: integer; Sep: AnsiChar=','): 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** 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 ('\\uFFF0base64encodedbinary')

- 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** GetTimeLog(Row,Field: integer; Expanded: boolean; FirstTimeChar: AnsiChar = 'T'): RawUTF8;

*Read-only access to a particular TTimeLog field value*

- return the result as Iso8601.Text() Iso-8601 encoded text

**function** GetU(Row,Field: integer): RawUTF8;

*Read-only access to a particular field value, as RawUTF8 text*

**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 trimmed 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): integer;

*Return the (previously hidden) ID value, 0 on error*

**function** IDColumnHide: boolean;

*If the ID column is available, hides it from fResults[]*

- usefull for simpler 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) of a particular field value*

**function** NewRecord(RecordType: TClass=nil): TObject;

*Create a new TSQLRecord instance for a specific Table*

- set a fixed TSQLRecord class or leave it nil to create one instance of the first associated record class (from QueryTables[])
- use this method to create a working copy of a table's record, e.g.
- the record will be freed when the TSQLTable will be destroyed: you don't need to make a Try..Finally..Free..end block with it

**function** QueryRecordType: TClass;

*Retrieve QueryTables[0], if existing*

**function** RowFromID(aID: integer): 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

**function** SearchFieldEquals(const aValue: RawUTF8; FieldIndex: integer): integer;

*Search for a value inside the raw table data*

- returns 0 if not found, or the matching Row number otherwise

**function** SearchValue(const aUpperValue: 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 preceding 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 aUpperValue: 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 preceding 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 comparaison function for a field, nil if not available (bad field index or field is blob)*

- field type is guessed from first data row

```
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 DeleteColumnValues(Field: integer);
```

*Delete the specified Column text from the Table*

- don't delete the Column: only delete UTF-8 text in all rows for this field

```
procedure DeleteRow(Row: integer);
```

*Delete the specified data Row from the Table*

- only overwrite the internal fResults[] pointers, don't free any memory, nor modify the internal DataSet

```
procedure FieldLengthMeanIncrease(aField, aIncrease: integer);
```

*Increase a particular Field Length Mean value*

- to be used to customize the field appearance (e.g. for adding of left checkbox for Marked[] fields)

```
procedure GetCSVValues(Dest: TStream; Tab: boolean; CommaSep: AnsiChar=','; AddBOM: boolean=false);
```

*Save the table in CSV format*

- 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 GetJSONValues(JSON: TStream; Expand: boolean; RowFirst: integer=0;
RowLast: integer=0); overload;
```

*Save the table values in JSON format*

- 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,..] }
- RowFirst and RowLast can be used to ask for a specified row extent of the returned data (by default, all rows are retrieved)

*Used for DI-2.1.2 (page 830).*

```
procedure GetRowValues(Field: integer; out Values: TRawUTF8DynArray); 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

```
procedure GetRowValues(Field: integer; out Values: TIntegerDynArray); overload;
```

*Get all values for a specified field into a dynamic Integer array*

```
procedure IDArrayFromBits(const Bits; var IDs: TIntegerDynArray);
```

*Get all IDs where individual bit in Bits are set*

```
procedure IDArrayToBits(var Bits; var IDs: TIntegerDynArray);
```

*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: TIntegerDynArray);
```

*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 SortBitsFirst(var Bits);
```

*Sort result Rows, according to the Bits set to 1 first*

```
procedure SortFields(Field: integer; Asc: boolean=true; PCurrentRow:
PInteger=nil; FieldType: TSQLFieldType=sftUnknown);
```

*Sort result Rows, 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 comparaison 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)

```
property FieldCount: integer read fFieldCount;
```

*Read-only access to the number of fields for each Row in this table*



**property** InternalState: cardinal **read** fInternalState **write** 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** OwnerMustFree: Boolean **read** fOwnerMustFree **write** fOwnerMustFree;

*If the TSQLRecord is the owner of this table, i.e. if it must free it*

**property** QuerySQL: RawUTF8 **read** fQuerySQL;

*Contains the associated SQL statement on Query*

**property** QueryTables: TClasses **read** fQueryTables;

*Contains the associated record class on Query*

**property** RowCount: integer **read** fRowCount;

*Read-only access to the number of data Row in this table*

- first row contains field name

- then 1..RowCount rows contain the data itself

**TSQLTableJSON = class(TSQLTable)**

*Get a SQL result from a JSON message, and store it into its own memory*

*Used for DI-2.1.1 (page 828), DI-2.1.2 (page 830).*

**constructor** Create(const Tables: array of TClass; const aSQL, aJSON: RawUTF8);  
**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

- you can use RecordClassesToClasses() wrapper function to convert an array of TSQLRecordClass into the expected array of TClass

*Used for DI-2.1.2 (page 830).*

**constructor** Create(const Tables: array of TClass; const aSQL: RawUTF8;  
 JSONBuffer: PUTF8Char; JSONBufferLen: integer); **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 aJSON parameter making a private copy of the data

- you can use RecordClassesToClasses() wrapper function to convert an array of TSQLRecordClass into the expected array of TClass

*Used for DI-2.1.2 (page 830).*

```
function UpdateFrom(const aJSON: RawUTF8; var Refreshed: boolean; PCurrentRow: PInteger): boolean;
```

*Update the result table content from a JSON-formated 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 PPUTF8CharArray is made inplace and is very fast (only one memory buffer is allocated for the whole data)

*Used for DI-2.1.2 (page 830).*

```
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
- 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

```
TJSONObjectDecoder = object(TObject)
```

*Record/object helper to handle JSON object decoding*

- used e.g. by GetJSONObjectAsSQL() function

```
DecodedFieldNames: PRawUTF8Array;
```

*Internal pointer over field names to be used*

- either FieldNames, either Fields[] array as defined in Decode()

```
FieldCount: integer;
```

*Number of fields decoded in FieldNames[] and FieldValues[]*

```
FieldLen: integer;
```

*Size of the TEXT data (in bytes) in FieldValues[]*

```
FieldNames: array[0..MAX_SQLFIELDS-1] of RawUTF8;
```

*Contains the decoded field names or value*

```
FieldNull: TSQLFieldBits;
```

*Decode() will set a bit for each field set JSON null value*

```
FieldValues: array[0..MAX_SQLFIELDS-1] of RawUTF8;
```

*Contains the decoded field names or value*

```
InlinedParams: boolean;
```

*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 successful 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

**function** EncodeAsSQLPrepared(const TableName: RawUTF8; Update: boolean): RawUTF8;

*Encode as a SQL-ready INSERT or UPDATE statement with ? as values*

- after a successful call to Decode()
- FieldValues[] content will be ignored

**function** SameFieldNames(const Fields: TRawUTF8DynArray): boolean;

*Returns TRUE if the specified array match the decoded fields names*

- after a successful call to Decode()

**procedure** AssignFieldNamesTo(var Fields: TRawUTF8DynArray);

*Set the specified array to the fields names*

- after a successful call to Decode()

**procedure** Decode(var P: PUTF8Char; const Fields: TRawUTF8DynArray; Params: TJSONObjectDecoderParams; RowID: integer=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 returns the next object start or nil on unexpected end of input
- if InlineParams is TRUE, FieldValues[] strings will be quoted
- if RowID is set, a RowID column will be added within the returned content

**procedure** Decode(JSON: RawUTF8; const Fields: TRawUTF8DynArray; Params: TJSONObjectDecoderParams; RowID: Integer=0; ReplaceRowIDWithID: Boolean=false); overload;

*Decode the JSON object fields into FieldNames[] and FieldValues[]*

- overloaded method expecting a RawUTF8 buffer, calling Decode(P: PUTF8Char)

**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

*Used for DI-2.1.3 (page 831).*

**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** FieldCountWithParents: integer;

*Return the total count of the published properties in this class and all its parents*

**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 831).*

**ClassType: TClass;**

*The class type*

**ParentInfo: PTypeInfo;**

*The parent class type information*

**PropCount: SmallInt;**

*The number of published properties*

**UnitName: string[255];**

*The name (without .pas extension) of the unit where 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*

**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 831).*

**BaseType: PTypeInfo;**

*The base type of this enumeration*

- always use PEnumType(typeinfo(TEnumType))^.BaseType or more usefull 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*

**OrdType: TOrdType;**

*Specify ordinal storage size and sign*

**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 GetEnumNameOrd(Value: Integer): PShortString;**

*Get the corresponding enumeration name*

- return the first one if Value is invalid (>MaxValue)

**function GetEnumNameTrimmed(const Value): RawUTF8;**

*Get the corresponding enumeration name, without the first lowercase chars (otDone -> 'Done')*

- Value will be converted to the matching ordinal value (byte or word)

**function GetEnumNameTrimmedValue(Value: PUTF8Char): 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** GetEnumNameTrimmedValue(**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** GetEnumNameValue(Value: PUTF8Char): Integer; overload;

*Get the corresponding enumeration ordinal value, from its name*

- 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*

- return -1 if not found (don't use directly this value to avoid any GPF)

**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

**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 831).*

**Kind:** TTypeKind;

*The value type family*

**Name:** ShortString;

*The declared name of the type ('String', 'Word', 'RawUnicode'...)*

**function** ClassSQLFieldType: TSQLFieldType;

*Get the SQL type of this Delphi class type*

**function** ClassType: PClassType;

*Get the class type information*

**function** EnumBaseType: PEnumType;

*Get the enumeration type information*

**function** FloatType: TFloatType;

*For floating point types, get the storage size and precision*

**function** InheritsFrom(AClass: TClass): boolean;

*Fast and easy find if a class type inherits from a specific class type*

**function** OrdType: TOrdType;

*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*

**function** SQLFieldType: TSQLFieldType;

*Get the SQL type of this Delphi type, as managed with the database driver*

**TPropInfo = object(TObject)**

*A wrapper containing a property definition, with GetValue() and SetValue() functions for direct Delphi / UTF-8 SQL type mapping/conversion:*

- handle byte, word, integer, cardinal, Int64 properties as INTEGER
- handle boolean properties as INTEGER (0 is false, anything else is true)
- handle enumeration properties as INTEGER, storing the ordinal value of the enumeration (i.e. starting at 0 for the first element)
- handle enumerations set properties as INTEGER, each bit corresponding to an enumeration (therefore a set of up to 64 elements can be stored in such a field)
- handle RawUTF8 properties as TEXT (UTF-8 encoded) - this is the preferred field type for storing some textual content in the ORM
- handle WinAnsiString properties as TEXT (UTF-8 decoded in WinAnsi char set)
- handle RawUnicode properties as TEXT (UTF-8 decoded as UTF-16 Win32 unicode)
- handle Single, Double and Extended properties as FLOAT
- handle TDateTime properties as ISO-8061 encoded TEXT
- handle TTimeLog properties as proprietary fast INTEGER date time
- handle Currency property as FLOAT (safely converted to/from currency)
- handle TSQLRecord descendant properties as INTEGER ROWID index to another record (warning: the value contains pointer(ROWID), not a valid object memory - you have to manually retrieve the record, using a integer(IDField) typecast)
- handle TSQLRecordMany descendant properties as an "has many" instance (this is a particular case of TSQLRecord: it won't contain pointer(ID), but an object)
- handle TRecordReference properties as INTEGER RecordRef-like value (use TSQLRest.Retrieve(Reference) to get a record content)
- handle TSQLRawBlob properties as BLOB
- handle dynamic arrays as BLOB, in the TDynArray.SaveTo binary format (is able to handle dynamic arrays of records, with records or strings within records)
- handle records as BLOB, in the RecordSave binary format (our code is ready for that, but Delphi doesn't create the RTTI for records so it won't work)
- WideString, shortstring, UnicodeString (i.e. Delphi 2009+ generic string), indexed properties are not handled yet (use faster RawUnicodeString instead of WideString and UnicodeString) - in fact, the generic string type is handled

*Used for DI-2.1.3 (page 831).*

**GetProc: PtrInt;**

*Contains the offset of a field, or the getter method set by 'read' Delphi declaration*



**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 (2147483648=\$80000000 indicates no default) when an ordinal or set property is saved as TPersistent default name index value is 0*

**PropType: PTypeInfo;**

*The type definition of this property*

**SetProc: PtrInt;**

*Contains the offset of a field, or the setter method set by 'write' Delphi declaration*

- if this field is nil (no 'write' was specified), SetValue() use GetProc to get the field memory address to save into

**StoredProc: PtrInt;**

*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 false", it is created as UNIQUE in the SQL database and its bit is set in Model.fIsUnique[]

**function GetCaption: string;**

*Get the corresponding caption name, from the property name*

- return generic "string" type, i.e. UnicodeString for Delphi 2009+  
 - internally call UnCamelCase() then System.LoadResStringTranslate() if available

**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 GetDynArray(Instance: TObject): TDynArray;**

*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 GetExtendedValue(Instance: TObject): Extended;**

*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** GetFieldAddr(Instance: TObject): pointer;

*Low-level getter of the field value memory pointer*

- return NIL if there a getter method

**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** GetHash(Instance: TObject; CaseInsensitive: boolean): cardinal;

*Retrieve 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

**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** GetLongStrValue(Instance: TObject): RawUTF8;

*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

**function** GetOrdValue(Instance: TObject): Integer;

*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** GetSQLFromFieldValue(const FieldValue: RawUTF8): RawUTF8;

*Return the field value as SQL statement ready*

- e.g. round strings with the ' character, and escape the text using double quotes, according to the official SQLite3 documentation

- expect enumerates (and boolean) values already encoded as integer

**function** GetValue(Instance: TObject; ToSQL: boolean; wasSQLString: PBoolean=nil): RawUTF8;

*Convert the published 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 ("\"uFFF0base64encodedbinary")
- getter method (read Get\*) is called if available
- handle Delphi values into UTF-8 SQL conversion
- sftBlobDynArray or sftBlobRecord are returned as BLOB literals ("X'53514C697465'" e.g.) if ToSQL is true, or base-64 encoded stream for JSON ("\"uFFF0base64encodedbinary")
- handle TPersistent, TCollection, TRawUTF8List or TStringList with ObjectToJSON

*Used for DI-2.1.3 (page 831).*

**function** IsBlob: boolean;

*Return true if this property is a BLOB (TSQLRawBlob)*

**function** IsSimpleField: boolean;

*Return true if this property is a valid simple field (INTEGER,FLOAT,TEXT) but not a BLOB (TSQLRawBlob)*

- use directly TSQLRecord.GetBit64(fJSONFields,i) if possible (faster)

**function** IsStored: boolean;

*Return FALSE if was marked as "stored false", or TRUE by default*

**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** SameValue(Item1,Item2: TObject; CaseInsensitive: boolean): boolean;

*Compare the content of the property of two object*

- 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

**function** SetBinary(Instance: TObject; P: PAnsiChar): PAnsiChar;

*Read the published property value from a binary buffer*

- returns next char in input buffer on success, or nil in case of invalid content supplied e.g.

**procedure** AppendName(var Text: RawUTF8; const Optional: RawUTF8='');

*Return Text+Name[+Optional]*

**procedure** CopyValue(Source, Dest: TObject);

*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

**procedure** GetBinary(Instance: TObject; W: TFileBufferWriter);

*Append the published property value into a binary buffer*

**procedure** GetValueVar(Instance: TObject; ToSQL: boolean; var result: RawUTF8; wasSQLString: PBoolean);

*Convert the published 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)

**procedure** NormalizeValue(var Value: RawUTF8);

*Normalize the content of Value, so that GetValue(Object,true) should return the same content (true for ToSQL format)*

**procedure** SetExtendedValue(Instance: TObject; const Value: Extended);

*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** 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** 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** 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

**procedure** SetOrdValue(Instance: TObject; Value: Integer);

*Low-level setter of the ordinal property value of a given instance*

- this method will check if the corresponding property is ordinal

**procedure** SetValue(Instance: TObject; Value: PUTF8Char);

*Convert UTF-8 encoded text into the published property value*

- setter method (write Set\*) is called if available

- if no setter exists (no write declaration), the getted field address is used

- handle UTF-8 SQL to Delphi values conversion

- expect BLOB fields encoded as SQLite3 BLOB 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 TStringList with JSONTToObject

*Used for DI-2.1.3 (page 831).*

**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 return information*

**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); **reintroduce**;

*Write the published integer, Int64, floating point values, string and enumerates (e.g. boolean) properties of the object*

- won't handle variant, shortstring and widestring properties
- add a new INI-like section with [Value.ClassName] if WithSection is true
- the object must have been compiled with the \$M+ define, i.e. must inherit from TPersistent or TSQLRecord
- the enumerates properties are stored with their integer index value

**TJSONSerializer = class(TJSONWriter)**

*Simple writer to a Stream, specialized for writing an object as JSON*

- resulting JSON content will be UTF-8 encoded
- use an internal buffer, faster than string+string

**destructor** Destroy; **override**;

*Relase all used memory and handles*

**class procedure** RegisterCustomSerializer(aClass: TClass; aReader:  
TJSONSerializerCustomReader; aWriter: TJSONSerializerCustomWriter);

*Define a custom serialization for a given class*

- by default, TSQLRecord, TPersistent, TString, 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

```
procedure WriteObject(Value: TObject; HumanReadable: boolean=false;
DontStoreDefault: boolean=true; FullExpand: boolean=false); 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 variant, shortstring and widestring properties
- the object must have been compiled with the \$M+ define, i.e. must inherit from TPersistent or TSQLRecord
- the enumerates properties are stored with their integer index value by default, but will be written as text if FullExpand is true: in this case, the resulting content WON'T be readable with JSOToObject() function
- TList objects are not handled by default - they will be written only if FullExpand is set to true (and JSOToObject 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
- you can add some custom serializers via the RegisterCustomSerializer() class method, to serialize any Delphi class
- will write also the properties published in the parent classes

```
procedure WriteObjectAsString(Value: TObject; HumanReadable: boolean=false;
DontStoreDefault: boolean=true; FullExpand: boolean=false);
```

*Same as WriteObject(), but will double all internal " and bound with "*

- this implementation will avoid most memory allocations

```
EORMException = class(ESynException)
```

*Generic parent class of all custom Exception types of this unit*

```
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*

```
EServiceException = class(EORMException)
```

*Exception dedicated to interface based service implementation*

## **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 831).*

**BlobFields: array of PPropInfo;**

*List all BLOB fields of this TSQLRecord*

- i.e. generic sftBlob fields (not sftBlobDynArray or sftBlobRecord)

**ClassProp: PClassProp;**

*Fast access to the RTTI properties attribute*

**DynArrayFields: array of PPropInfo;**

*List of all sftBlobDynArray fields of this TSQLRecord*

**ExternalDatabase: TObject;**

*Opaque structure used on the Server side to specify e.g. the DB connection*

- will define such a generic TObject, to avoid any unnecessary dependency to the SynDB unit in SQLite3Commons
- in practice, will be assigned by VirtualTableExternalRegister() to a TSQLDBConnectionProperties instance

**ExternalTableName: RawUTF8;**

*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/Jet/SQLite3 backend
- equals SQLTableName by default (may be overridden e.g. by SQLite3DB's VirtualTableExternalRegister procedure)

**Fields: array of PPropInfo;**

*List all fields, as retrieved from RTTI*

**FieldsName: TRawUTF8DynArray;**

*List all fields names, as retrieved from RTTI*

- FieldsName[] := RawUTF8(Fields[0].Name);

**FieldType: array of TSQLFieldType;**

*Fast access to the SQL Field Type of all published properties*

**Filters: array of TObjectList;**

*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 TObjectList - for TSynTableFieldProperties there are separated Filters[] and Validates[] array, for better performance

**HasNotSimpleFields:** boolean;

*If this class has any BLOB or TSQLRecordMany fields*  
- i.e. some fields to be ignored

**HasTypeFields:** TSQLFieldTypes;

*Set of field types appearing in this record*

**IsUniqueFieldsBits:** TSQLFieldBits;

*Bit set to 1 for an unique field*  
- an unique field is defined as "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

**ManyFields:** array of PPropInfo;

*List all TSQLRecordMany fields of this TSQLRecord*

**Model:** TSQLModel;

*The associated TSQLModel instance*  
- if this record is associated to multiple TSQLModels, the latest registered will be stored here

**ModelTableIndex:** integer;

*The index in the Model.Tables[] array*  
- e.g. allow O(1) search of a TSQLRecordClass in a model

**SimpleFields:** array of PPropInfo;

*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

**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 soInsert and soSelect e.g.

**SQLInsertSet:** RawUTF8;

*All fields, excluding the ID field, exposed as 'COL1,COL2'*  
- to be used e.g. for INSERT statements

**SQLSelectAll:** array[boolean] of RawUTF8;

*The SQL statement for reading all simple fields*  
- SQLSelectAll[false] with RowID (i.e. SQLFromSelectWhere('\*',') content)  
- SQLSelectAll[true] with ID and potentially external table name  
- to be checked if we may safely call EngineList()

**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



**SQLTableNameUpperWithDot: RawUTF8;**

*The Table name in the database in uppercase with a final '.'*

- e.g. 'TEST.' for TSQLRecordTest class
- can be used with IdempPChar() for fast check of a table name

**SQLTableSimpleFields: array[boolean,boolean] of RawUTF8;**

*The simple field names in a SQL SELECT compatible format: 'COL1,COL2' e.g.*

- format is

SQLTableSimpleFields[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

**SQLUpdateSet: array[boolean] of RawUTF8;**

*The updated fields exposed as 'COL1=?,COL2=?'*

- SQLUpdateSet[false]=simple fields, SQLUpdateSet[true]=all fields excluding ID (but including TCreateTime fields - as used in TSQLVirtualTableExternal.Update method)
- to be used e.g. for UPDATE statements

**Table: TSQLRecordClass;**

*The TSQLRecord class*

**constructor** Create(aTable: TSQLRecordClass);

*Initialize the properties content*

**destructor** Destroy; **override;**

*Release associated used memory*

**function** AddFilterOrValidate(aFieldIndex: integer; aFilter: TSynFilterOrValidate): TSynFilterOrValidate; **overload;**

*Register a custom filter or validation rule to the class for a specified field*

- this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)
- will return the specified associated TSynFilterOrValidate instance
- will return nil in case of an invalid field index

**function** AddFilterOrValidate(const aFieldName: RawUTF8; aFilter: TSynFilterOrValidate): TSynFilterOrValidate; **overload;**

*Register a custom filter or Validate to the class for a specified field*

- this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)
- will return the specified associated TSynFilterOrValidate instance

**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** CreateJSONWriter(JSON: TStream; Expand: boolean; withID: boolean; **const** aFields: TSQLFieldBits; KnownRowCount: integer): TJSONSerializer;

*Create a TJSONWriter, ready to be filled with TSQLRecord.GetJSONValues(W)*

**function** FieldIndex(**const** PropName: shortstring): integer;

*Return the Field Index number in published properties of this record*  
 - returns -1 if not found  
 - warning: also returns -1 if PropName is 'ID', which is a correct field name but not a published property

**function** FieldIndexFromRawUTF8(**const** PropName: RawUTF8): integer;

*Return the Field Index number in published properties of this record*  
 - returns -1 if not found  
 - warning: also returns -1 if PropName is 'ID', which is a correct field name but not a published property

**function** FieldIndexsFromRawUTF8(**const** aFields: **array of** RawUTF8; **var** Bits: TSQLFieldBits): boolean;

*Set all bits corresponding to the supplied field names*  
 - returns TRUE on success, FALSE if any field name is not existing

**function** FieldProp(**const** PropName: shortstring): PPropInfo;

*Retrieve a Field property RTTI information from a Property Name*

**function** FieldPropFromRawUTF8(**const** PropName: RawUTF8): PPropInfo;

*Retrieve a Field property RTTI information from a Property Name*

**function** IsFieldName(**const** PropName: RawUTF8): boolean;

*Return TRUE if the given name is either ID/RowID, either a property name*

**function** SQLAddField(FieldIndex: integer): RawUTF8;

*Return the UTF-8 encoded SQL statement source to alter the table for adding the specified field*

**function** SQLFromSelectWhere(**const** SQLSelect, SQLWhere: RawUTF8): RawUTF8;

*Compute the SQL statement to be executed for a specific SELECT*

**procedure** SetSimpleFieldsExpandedJSONWriter(W: TJSONWriter; withID: boolean; Occasion: TSQLOccasion);

*Initialize the JSON writer parameters with simple fields*  
 - recreate especially the ColNames[] and other necessary properties  
 - is used e.g. in TSQLRestClientURI.BatchUpdate and BatchAdd methods

**property** Kind: TSQLRecordVirtualKind **read** fKind **write** SetKind;

*Define if a normal table (rSQLite3), an FTS3/FTS4/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** RecordManyDestClass: TSQLRecordClass **read** fRecordManyDestClass;

*For a TSQLRecordMany class, points to the Dest property class*

**property** RecordManyDestProp: PPropInfo **read** fRecordManyDestProp;

*For a TSQLRecordMany class, points to the Dest property RTTI*

**property** RecordManySourceClass: TSQLRecordClass **read** fRecordManySourceClass;

*For a TSQLRecordMany class, points to the Source property class*

**property** RecordManySourceProp: PPropInfo **read** fRecordManySourceProp;

*For a TSQLRecordMany class, points to the Source property RTTI*

**TSQLRestServerSessionContext = record**

*Used to store the current execution context of a remote request*

- if RESTful authentication is enabled, it will be filled as expected

**Group: integer;**

*The corresponding TAuthSession.User.GroupRights.ID value*

- is undefined if Session is 0 or 1 (no authentication running)

**ID: integer;**

*The associated TSQLRecord.ID, as decoded from URI scheme*

- this property will be set from incoming URI, even if RESTful authentication is not enabled

**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

**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

TSQLRest.HandleAuthentication = FALSE

**User: integer;**

*The corresponding TAuthSession.User.ID value*

- is undefined if Session is 0 or 1 (no authentication running)

**TSQLRestServerCallbackParams = record**

*Store all parameters for a TSQLRestServerCaLLBack event handler*

- having a dedicated record avoid changing the implementation methods signature if the framework add some parameters to this structure

- see TSQLRestServerCaLLBack for general code use

**Context:** TSQLRestServerSessionContext;

*The corresponding authentication session ID*

- Context.Session=1 (CONST\_AUTHENTICATION\_NOT\_USED) if authentication mode is not enabled
- Context.Session=0 (CONST\_AUTHENTICATION\_SESSION\_NOT\_STARTED) if the session not started yet
- if authentication is enabled, Context.User and Context.Group can be checked to perform security during service execution
- Context.ID is the record ID as specified at the URI level (if any)

**ErrorMsg:** PRawUTF8;

*A pointer to an optional error message to be sent back to the client*

- to be used to specify the numerical error message returned as integer result of the TSQLRestServerCallback function

**Head:** PRawUTF8;

*A pointer to the Header to be sent back to the client*

- you can use this value e.g. to change the result mime-type

**MethodIndex:** integer;

*The index of the callback published method within the internal class list*

**Parameters:** PUTF8Char;

*URI inlined parameters*

- use UriDecodeValue\*() functions to retrieve the values

**Resp:** RawUTF8;

*The response to be sent back to the client*

**SentData:** RawUTF8;

*The message body (e.g. HTTP body) as send by the client*

**Table:** TSQLRecordClass;

*The Table as specified at the URI level (if any)*

**TableIndex:** integer;

*The index in the Model of the Table specified at the URI level (if any)*

**URI:** RawUTF8;

*The URI address, just before parameters*

- can be either the table name (in RESTful protocol), or a service name

**TSQLRecordFill = class(TObject)**

*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)

**destructor** Destroy; **override;**

*Finalize the mapping*

**function** Fill(aRow: integer): Boolean; overload;

*Fill a TSQLRecord published properties from a TSQLTable row*  
 - use the mapping prepared with Map() method

**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

**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** 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

**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  
 - this 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

*Used for DI-2.1.1 (page 828), DI-2.1.2 (page 830), DI-2.1.3 (page 831).*

**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; aID: integer; 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; 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; FormatSQLWhere: PUTF8Char; **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:

```
Rec := TSQLMyRecord.Create(aClient, 'Count=:(%):' [aCount], []);
```

or (letting the inlined parameters being computed by FormatUTF8)

```
Rec := TSQLMyRecord.Create(aClient, 'Count=?', [], [aCount]);
```

or even better, using the other Create overloaded constructor:

```
Rec := TSQLMyRecord.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; FormatSQLWhere: PUTF8Char; **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 '?'

**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:

```
aRec := TSQLMyRec.Create(Client,FormatUTF8('Salary>? AND Salary<?',[],[1000,2000]));
```

or call the overloaded constructor with BoundsSQLWhere array of parameters

**constructor** CreateAndFillPrepare(aClient: TSQLRest; FormatSQLWhere: PUTF8Char; **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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields



```
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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields

```
constructor CreateAndFillPrepare(aClient: TSQLRest; const aIDs:  

TIntegerDynArray; 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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields



**constructor** CreateAndFillPrepare(aClient: TSQLRest; FormatSQLWhere: PUTF8Char; 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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields

**constructor** CreateAndFillPrepareMany(aClient: TSQLRest; aFormatSQLJoin: PUTF8Char; 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 Destroy) - 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,
  'Owner=? and Categories.Dest.Name=? and (Sizes.Dest.Name=? or Sizes.Dest.Name=?)',[],
  ['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)
    writeln(aProd.Name, ' ',aProd.Owner, ' ',aProd.Categories.Dest.Name, ' ',aProd.Sizes.Dest.Name);
    // 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

**destructor** Destroy; **override**;

*Release the associated memory*

- in particular, release all TSQLRecordMany instance created by the constructor of this TSQLRecord

**class function** AddFilterOrValidate(**const** aFieldName: RawUTF8; aFilter: TSynFilterOrValidate): TSynFilterOrValidate;

*Register a custom filter or Validate to the class for a specified field*

- this will be used by TSQLRecord.Filter and TSQLRecord.Validate methods (in default implementation)

- will return the specified associated TSynFilterOrValidate instance

- this function is just a wrapper around RecordProps.AddFilterOrValidate

**class function** CaptionName(Action: PShortString=nil; ForHint: boolean=false): string; **virtual**;

*Get the captions to be used for this table*

- 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. UnicodeString 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

**function** ClassProp: PClassProp;

*Return the RTTI property information for this record*

*Used for DI-2.1.3 (page 831).*

**function** CreateCopy: TSQLRecord;

*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)

**function** DynArray(**const** DynArrayFieldName: shortstring): 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(DynArrayFieldIndex: integer): 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** FillOne: 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
- call FillRow() to update published properties values

**function** FillPrepare(aClient: TSQLRest; FormatSQLWhere: PUTF8Char; **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() to get Table.RowCount row values
- 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
- 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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields

```
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() to get Table.RowCount row values
- 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
- 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]));
```

or call the overloaded FillPrepare() method directly with BoundsSQLWhere array of parameters

- aCustomFieldsCSV can be used to specify which fields must be retrieved (default is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the 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; FormatSQLWhere: PUTF8Char; 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() to get Table.RowCount row values
- 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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields

```
function FillPrepare(aClient: TSQLRest; const aIDs: TIntegerDynArray; 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() to get Table.RowCount row values
- 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 is to retrieve all table fields, but you may need to access only one or several fields, and will save remote bandwidth by specifying the needed fields): notice that you should not use this optional parameter if you want to Update the retrieved record content later, since the missing fields will be left with previous values - but BatchUpdate() will set only ID, TModTime and mapped fields

```
function FillPrepareMany(aClient: TSQLRest; aFormatSQLJoin: PUTF8Char; const
aParamsSQLJoin, aBoundsSQLJoin: array of const): boolean;
```

*Prepare to loop through a JOINed statement including TSQLRecordMany fields*

- all TSQLRecordMany.Dest 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','for boy','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, '
',aProd.Sizes.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 explicetely 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 FillRow()*

- the Row number (property FillCurrentRow) is reset to 1
- return true on success, false if no Row data is available

```
function Filter(const aFields: TSQLFieldBits=[0..MAX_SQLFIELDS-1]): boolean;  
overload; virtual;
```

*Filter the specified fields values of the current TSQLRecord instance*

- by default, this will perform all TSynFilter as registered by [RecordProps.]AddFilterOrValidate()
- inherited classes may add some custom filtering 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 the specified fields values of the current TSQLRecord instance*

- this version will call the overloaded Filter() method above
- return TRUE if all field names were correct and processed, FALSE otherwise

```
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 usefull to have this method available
- will return '' in case of wrong property name
- BLOB and dynamic array fields are returned as '\uFFFF0base64encodedbinary'

```
function GetJSONValues(Expand: boolean; withID: boolean; Occasion: TSQLOccasion;  
UsingStream: TCustomMemoryStream=nil): RawUTF8; overload;
```

*Same as above, but returning result into a RawUTF8*

- if UsingStream is not set, it will use a temporary THeapMemoryStream instance

*Used for DI-2.1.2 (page 830), DI-2.1.3 (page 831).*

```
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 flsUnique[] array, which set the bits values to 1 if a property field was published with "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 convenience

**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 convenience

**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 is very fast, and works with all versions of Delphi - including 64 bit target)

**function** RecordReference(Model: TSQLModel): TRecordReference;

*Return the TRecordReference 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() above

**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** SetBinaryValues(**var** P: 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.)
- 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** SetFieldVarData(aFieldIndex: integer; **const** aValue: TVarData): boolean;

*Set a field value from a custom TVarData sub type (not a true variant)*

- the field values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])

**function** SetFieldVarDatas(**const** Values: TVarDataDynArray): boolean;

*Set all field values from a supplied array of TVarData sub type*

- the field values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- 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): **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): 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()

```
procedure ClearProperties;
```

*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 and sftCreateTime / TCreateTime 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(const JSONRecord: RawUTF8); 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

```
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(P: PUTF8Char); overload;

*Fill all published properties of this object from a JSON result*

- the data inside P^ is modified (unescaped 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

**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()

**procedure** FillFrom(Table: TSQLTable; Row: integer); overload;

*Fill all published properties of this object from a TSQLTable result row*

- call FillPrepare() then FillRow()

**procedure** FillPrepare(Table: TSQLTable; aCheckTableName: TSQLCheckTableName=ctnNoCheck); overload;

*Prepare to get values from a TSQLTable result*

- then call FillRow() to get Table.RowCount row values
- 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** FillRow(aRow: integer; aDest: TSQLRecord=nil); 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

**procedure** FillValue(const PropName: ShortString; Value: PUTF8Char);

*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)

**procedure** GetBinaryValues(W: TFileBufferWriter);

*Write the field values into the binary buffer*

- won't write the ID field (should be stored before, with the Count e.g.)

**procedure** GetFieldVarData(aFieldIndex: integer; **var** aValue: TVarData; **var** temp: RawByteString);

*Retrieve a field value into a custom TVarData sub type (not a true variant)*

- the field values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- the temp RawByteString is used as a temporary storage for TEXT or BLOB and should be available during all access to the TVarData content

**procedure** GetJSONValues(W: TJSONSerializer); **overload;**

*Same as above, but in a TJSONWriter (called by the first two overloaded functions)*

*Used for DI-2.1.2 (page 830), DI-2.1.3 (page 831).*

**procedure** GetJSONValues(JSON: TStream; Expand: boolean; withID: boolean; Occasion: TSQLOccasion); **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

*Used for DI-2.1.2 (page 830), DI-2.1.3 (page 831).*

**class procedure** InitializeTable(Server: TSQLRestServer; **const** FieldName: RawUTF8); **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)
- 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 SQLite3 BLOB literals ("x'01234'" e.g.) or '\uFFFF0base64encodedbinary'

**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** 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: integer read GetID write fID;

*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 usually; you should not have to write aRecord.ID := someID in your code, since the ID is set during Retrieve or Add of the record

**property** InternalState: cardinal read fInternalState;

*This property contains the internal state counter of the server database when the data was retrieved from it*

- may 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

**TSQLLocks = object(TObject)**

*Used to store the locked record list, in a specified table*

- the maximum count of the locked list is fixed to 512 by default, which seems correct for common usage

**Count: integer;**

*The number of locked records stored in this object*

**ID: array[0..MAX\_SQLLOCKS-1] of integer;**

*Contains the locked record ID*

- an empty position is marked with 0 after Unlock()

**Ticks: array[0..MAX\_SQLLOCKS-1] of cardinal;**

*Contains the time and date of the Lock*

- filled internally by the very fast GetTickCount function (faster than TDateTime or TSystemTime/GetLocalTime)
- used to purge to old entries - see PurgeOlderThan() method below

**function** isLocked(aID: integer): boolean;

*Return true if a record, specified by its ID, is locked*

**function** Lock(aID: integer): boolean;

*Lock a record, specified by its ID*

- returns true on success, false if was already locked

**function** UnLock(aID: integer): 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

- also purge values where GetTickCount wrapped around to zero after 49.7 days (for consistency)

**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 SQLite3ToolBar unit and TSQLModel.Create() overloaded method

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 TSQLRecord.Caption(nil) method
- but you can override this value by setting a pointer to a resourcestrng

**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 TSQLRecord.Caption(nil) method
- but you can override this value by setting a pointer to a resourcestrng

**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 resourcestrng

**EditFieldHintsToReport:** boolean;

*If the default report must contain the edit field hints*

- i.e. if the resourcestrng 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 (i.e. TRecordReference/TSQLRecord published propert) 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 IIClient (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

**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

**RecordReferences: array of record TableIndex: integer; FieldType: TSQLFieldType;  
FieldRecordClass: TSQLRecordClass; FieldName: PShortString; end;**

*This array contain all TRecordReference and TSQLRecord properties existing in the database model*

- 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 by TSQLRestServer.AfterDeleteForceCoherency

**TableProps: array of TSQLRecordProperties;**

*The associated 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



```
constructor Create(TabParameters: PSQLRibbonTabParameters; TabParametersCount,  
TabParametersSize: integer; const NonVisibleTables: array of TSQLRecordClass;  
const aRoot: RawUTF8='root'); 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(CloneFrom: TSQLModel); 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(const Tables: array of TSQLRecordClass; const aRoot:  
RawUTF8='root'; aVirtualsRemain: boolean=false); 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 flsUnique[] array from "stored false" published properties of every TSQLRecordClass
- if aVirtualsRemain is FALSE this will reset all supplied classes to be defined as non-virtual (i.e. Kind=rSQLite3): VirtualTableExternalRegister explicit calls are to be made if expected

```
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 true 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

```
function EventName(const Event): string;
```

*Get the text conversion of a given Event, ready to be displayed*

```
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 false" in its property definition
- reflects the internal private flsUnique property



**function** GetMainFieldName(Table: TSQLRecordClass; 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 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** 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(aTable: TSQLRecordClass): integer; overload;

*Get the index of aTable in Tables[]*

**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(SQLTableName: PUTF8Char): integer; overload;

*Get the index of a table in Tables[]*

- expects SQLTableName to be SQL-like formatted (i.e. without TSQL[Record])

**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
- 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** isLocked(aTable: TSQLRecordClass; aID: integer): boolean; overload;

*Return true if a specified record is locked*

**function** isLocked(aRec: TSQLRecord): boolean; overload;

*Return true if a specified record is locked*

**function** Lock(aTable: TSQLRecordClass; aID: integer): boolean; overload;

*Lock a record*

- returns true on success, false if was already locked or if there's no place more in the lock table (as fixed by MAX\_SQLLOCKS const, i.e. 512)

**function** Lock(aTableIndex, aID: integer): boolean; overload;

*Lock a record*

- returns true on success, false if was already locked or if there's no place more in the lock table (as fixed by MAX\_SQLLOCKS const, i.e. 512)

**function** Lock(aRec: TSQLRecord): boolean; overload;

*Lock a record*

- returns true on success, false if was already locked or if there's no place more in the lock table (as fixed by MAX\_SQLLOCKS const, i.e. 512)

**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: integer): TRecordReference;

*Return the TRecordReference pointing to the specified record*

**function** UnLock(aTable: TSQLRecordClass; aID: integer): 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** VirtualTableModule(aClass: TSQLRecordClass): TSQLVirtualTableClass;

*Retrieve a Virtual Table module associated to a class*

**function** VirtualTableRegister(aClass: TSQLRecordClass; aModule: TSQLVirtualTableClass): 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)  
 - 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 SQLite3ToolBar unit

**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** UnLockAll;

*UnLock all previously locked records*

**property** Actions: PEnumType read fActions;

*Get the enumerate type information about the possible actions to be 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** Owner: TSQLRest read fOwner write fOwner;

*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** Root: RawUTF8 read fRoot;

*The Root URI path of this Database Model*

**property** Table[const SQLTableName: RawUTF8]: TSQLRecordClass read GetTable;

*Get a class from a table name*

- expects SQLTableName to be SQL-like formatted (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** Tables: TSQLRecords read fTables;

*Get the classes list (TSQLRecord descendent) of all available tables*

**property** URI[aClass: TSQLRecordClass]: RawUTF8 read getURI;

*Get the URI for a class in this Model, as 'ModelRoot/SQLTableName'*

**RecordRef = object(TObject)**

*Usefull object to type cast TRecordReference type value into explicit TSQLRecordClass and ID*

- 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

- 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: cardinal;**

*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

**function** ID: integer;

*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: integer);

*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

**TSQLRecordRTree = class**(TSQLRecordVirtual)

*A base record, corresponding to an R-Tree table*

- 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 must have only sftFloat (double) fields, grouped by pairs, each as minimum- and maximum-value, up to 5 dimensions (i.e. 11 columns, including the ID property)

- the ID: integer 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

**class function** RTreeSQLFunctionName: RawUTF8;

*Will return 'MapBox\_in' e.g. for TSQLRecordMapBox*

## TSQLRecordFTS3 = class(TSQLRecordVirtual)

*A base record, corresponding to a FTS3 table, i.e. implementing full-text*

- FTS3/FTS4 table are SQLite virtual tables which 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://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: integer 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 request 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 - 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
- in order to make FTS3/FTS4 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):

```
var IDs: TIntegerDynArray;
if FTSMATCH(TSQLMyFTS3Table, 'text MATCH "linu*"', IDs) then
  // you've all matching IDs in IDs[]
```

## 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](http://sqlite.org/fts3.html#section_1_2)
- this method must be called server-side
- returns TRUE on success

## property DocID: integer 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>

```
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, TSQLRecordFTS4 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](http://sqlite.org/fts3.html#section_1_1)

```
TSQLRecordFTS4Porter = class(TSQLRecordFTS4)
```

*This base class will create a FTS4 table using the Porter Stemming algorithm*

- see <http://sqlite.org/fts3.html#tokenizer>

## TSQLRecordMany = 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

```

TSQLEDest = class(TSQLRecord);
TSQLESource = class;
TSQLEDestPivot = class(TSQLRecordMany)
private
  fSource: TSQLSource;
  fDest: TSQLEDest;
  fTime: TDateTime;
published
  property Source: TSQLSource read fSource; // map Source column
  property Dest: TSQLEDest read fDest; // map Dest column
  property AssociationTime: TDateTime read fTime write fTime;
end;
TSQLESource = class(TSQLRecord)
private
  fDestList: TSQLEDestPivot;
published
  DestList: TSQLEDestPivot read fDestList;
end;
  
```

- 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: integer; **out** DestIDs: TIntegerDynArray): boolean; overload;

*Retrieve all Dest items IDs associated to the specified Source*



```
function DestGet(aClient: TSQLRest; out DestIDs: TIntegerDynArray): 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: Integer): 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: Integer; out DestIDs: TIntegerDynArray): 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: Integer; 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: integer=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, referred 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>? AND Salary<?',[],[1000,2000]));
```

```
function FillManyFromDest(aClient: TSQLRest; aDestID: integer; 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>? AND Salary<?',[],[1000,2000]));
```

```
function IDWhereSQL(aClient: TSQLRest; aID: integer; 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: Integer; NoDuplications: 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: Integer; NoDuplications: boolean=false): boolean; overload;

*Add a Dest record to the Source record list*

- returns TRUE on success, FALSE on error
- if NoDuplications 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)

**function** ManyDelete(aClient: TSQLRest; aDestID: Integer; aUseBatchMode: boolean=false): 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
- if aUseBatchMode is TRUE, it will use aClient.BatchDelete() instead of the slower aClient.Delete() method - but you shall call it within a BatchStart / BatchSend block

**function** ManyDelete(aClient: TSQLRest; aSourceID, aDestID: Integer; aUseBatchMode: boolean=false): 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 aUseBatchMode is TRUE, it will use aClient.BatchDelete() instead of the slower aClient.Delete() method - but you shall call it within a BatchStart / BatchSend block

**function** ManySelect(aClient: TSQLRest; aDestID: Integer): 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: Integer): 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: integer; out SourceIDs: TIntegerDynArray): 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, e.g.

**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 trimmed*

- do nothing is 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 internaly)*

- 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*

- very secured SHA-256 hashing is used internaly

- 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** SignatureTime;

**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;

**TServiceMethodArgument = object(TObject)**

*Describe a service provider method argument*

**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)

**OffsetInStack: integer;**

*Byte offset in the CPU stack of this argument*

- if -1 for EAX, -2 for EDX and -3 for ECX registers

**ParamName: PShortString;**

*The argument name, as declared in Delphi*

**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

**TypeInfo: PTypeInfo;**

*The low-level RTTI information of this argument*

**TypeName: PShortString;**

*The type name, as declared in Delphi*

**ValueDirection: TServiceMethodValueDirection;**

*How the variable is to be passed (by value vs by reference)*

**ValueIsString: boolean;**

*True for smvRawUTF8, smvString, smvWideString and kind of parameter*  
- smvRecord has it to false, even if they are Base-64 encoded within the JSON content

**ValueType: TServiceMethodValueType;**

*We do not handle all kind of Delphi variables*

**ValueVar: TServiceMethodValueVar;**

*How the variable may be stored*

**function SerializeToContract: RawUTF8;**

*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

**TServiceMethod = object(TObject)**

*Describe a 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

**ArgsResultIndex: integer;**

*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 transfert, not a JSON object

**ArgsSizeInStack: cardinal;**

*Needed CPU stack size (in bytes) for all arguments*

**ArgsUsed: set of TServiceMethodValueType;**

*Contains all used kind of arguments*

**ArgsUsedCount: array[TServiceMethodValueVar] of integer;**

*Contains the count of variables for all used kind of arguments*

**ExecutionDenied: set of 0..255;**

*The list of denied TSQLAuthGroup ID(s)*

- used on server side within TSQLRestServer.LaunchService method
- 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

**ExecutionOptions: TServiceMethodExecutionOptions;**

*Execution options for this method*

- you can e.g. force a method to be executed in the main thread by calling TServiceFactoryServer.ExecuteInMainThread()

**MethodIndex: integer;**

*Method index in the original interface*

- our custom methods start at index 3, since QueryInterface, \_AddRef, and \_Release methods are always defined by default

**Padding: array[0..2] of byte;**

*For better alignment of the fMethods[] items*

**URI: RawUTF8;**

*The method URI*

- basically the method name as declared in Delphi code (e.g. 'Add' for ICalculator.Add)
- this property value is hashed internally for faster access

**function InternalExecute(Instances: array of pointer; Par: PUTF8Char; Res: TTextWriter; var aHead: RawUTF8): boolean;**

*Execute the corresponding method of a given TInterfacedObject instance*

- will retrieve a JSON array of parameters from Par
- will append a JSON array of results in Res, or set an Error message

**TServiceCustomAnswer = record**

*A record type to be used as result for a function method for custom content*

- 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*

**Header: RawUTF8;**

*The response type, as encoded in the HTTP header*

- useful to set the response mime-type - see e.g. the TEXT\_CONTENT\_TYPE\_HEADER or HTML\_CONTENT\_TYPE\_HEADER constants or GetMimeContentType() function
- in order to be handled as expected, this field SHALL be set (<>) (otherwise it will be transmitted with default JSON object format)

**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

**constructor** Create; reintroduce; virtual;

*This constructor which will call GetClass to initialize the collection*

**TInterfaceFactory = class(TObject)**

*A common ancestor for any class needing interface RTTI*

**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

**function** FindMethodIndex(const aMethodName: RawUTF8): integer;

*Find the index of a particular method in internal Methods[] list*

**property** Methods: TServiceMethodDynArray read fMethods;

*The declared internal methods*

- list list does not contain the default AddRef/

**TServiceFactory = class(TInterfaceFactory)**

*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:

```
if Services.GUID(IID_ICalculator).Get(I) then
  result := I.Add(10,20);
```

**constructor** Create(aRest: TSQLRest; aInterface: PTypeInfo; aInstanceCreation: TServiceInstanceImplementation);

*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

**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_
```

or (if rmJSON\_RPC is used):

```
POST /root/Interface
(...)
{ "method": "_contract_", "params": [] }
```

(e.g. to be checked in TServiceFactoryClient.Create constructor)

**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** InterfaceIID: TGUID **read** fInterfaceIID;

*The registered Interface GUID*

**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-ee2522c8f633}' ]` into `'00amyWGct0y_ze41Isj2Mw'`

- can be substituted to the clear InterfaceURI name

**property** InterfaceTypeInfo: PTypeInfo **read** fInterfaceTypeInfo;

*The registered Interface low-level Delphi RTTI type*

**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: TObject;**

*The implementation instance itself*

**InstanceID: Cardinal;**

*The internal Instance ID, as remotely sent in "id":1*

- is set to 0 when an entry in the array is free

**LastAccess: Cardinal;**

*Last time stamp access of this instance*

**procedure** SafeFreeInstance;

*Used to catch and ignore any exception in Instance.Free*

**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

**constructor** Create(aRestServer: TSQLRestServer; aInterface: PTypeInfo;  
 aInstanceCreation: TServiceInstanceImplementation; aImplementationClass: TClass;  
 aTimeOutSec: cardinal=30\*60); reintroduce;

*Initialize the service provider on the server side*

- expect an implementation class
- for sicClientDriven, sicPerSession, sicPerUser or sicPerGroup modes, a time out (in seconds) can be defined - if the time out 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;

*Release 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 integer): 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 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** 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 integer): 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 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** 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** integer): 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** integer): 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 ExecuteInMainThread(const aMethod: array of RawUTF8; Enable:
boolean=true): TServiceFactoryServer;
```

*Force a method to be executed in the main thread*

- 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 - methods have to handle multi-threading safety carefully, e.g. by using TRTLCriticalSection mutex on purpose
- ExecuteInMainThread() 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
- methods names should be specified as an array (e.g. ['Add','Multiply'])
- if not method name is specified (i.e. []), all methods will be set
- if Enable is left to TRUE default value, it will force main-thread execution; if set to FALSE, it will set it back to in-thread faster method execution
- this method returns self in order to allow direct chaining of setting calls for the service, 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
- all other kind of instance creation will behave the same as sicSingle when accessed directly from this method, i.e. from server side

```
function RestServer: TSQLRestServer;
```

*Just typecast 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)

```
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

```
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
- 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 structures*  
 - especially the internal shared VTable and code Stub

**function** Get(out Obj): Boolean; **override**;

*Retrieve an instance of this interface from the server side*

**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

TServiceContainer = **class**(TObject)

*A global services provider class*  
 - used to maintain a list of interfaces implementation

**constructor** Create(aRest: TSQLRest);

*Initialize the list*

**destructor** Destroy; **override**;

*Release all registered services*

**function** AddInterface(const aInterfaces: array of PTypeInfo; aInstanceCreation:  
 TServiceInstanceImplementation; aContractExpected: RawUTF8=''): boolean;

*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** Count: integer;

*Return the number of registered service interfaces*

**function** GUID(const aGUID: TGUID): TServiceFactory; overload;

*Retrieve a service provider from its GUID*  
 - on match, it will return the service the corresponding interface factory  
 - returns nil if the GUID does not match any registered interface

**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(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

**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\_ze4llsj2Mw'

**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\_ze4llsj2Mw' 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

**TServiceContainerServer = class**(TServiceContainer)

*A services provider class to be used on the server side*

- this will maintain a list of true implementation classes

**function** AddImplementation(aImplementationClass: TClass; **const** aInterfaces: **array of** PTypeInfo; aInstanceCreation: TServiceInstanceImplementation): TServiceFactoryServer;

*Method called on the server side to register a service via its interface(s) and a specified implementation class*

- 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)

**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

**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

**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

**TSQLRestCacheEntryValue = record**

*For TSQLRestCache, stores a table values*

**ID: integer;**

*Corresponding ID*

**JSON: RawUTF8;**

*JSON encoded UTF-8 serialization of the record*

**TimeStamp: cardinal;**

*GetTickCount value when this cached value was stored*

- equals 0 when there is no JSON value cached

**TSQLRestCacheEntry = object(TObject)**

*For TSQLRestCache, stores a table settings and values*

**CacheAll: boolean;**

*The whole specified Table content will be cached*

**CacheEnable: boolean;**

*TRUE if this table should use caching*

- i.e. if was not set, or worth it for this table (e.g. in-memory table)

**Count: integer;**

*The number of entries stored in Values[]*

**Mutex: TRTLCriticalSection;**

*Used to lock the table cache for multi thread safety*

**TimeOut: Cardinal;**

*Time out value (in ms)*

- if 0, caching will never expire

**Value: TDynArray;**

*TDynArray wrapper around the Values[] array*

**Values: TSQLRestCacheEntryValueDynArray;**

*All cached IDs and JSON content*

**function** RetrieveJSON(aID: integer; aValue: TSQLRecord): boolean; overload;

*Unserialize a JSON cached record of a given ID*

**function** RetrieveJSON(aID: integer; var aJSON: RawUTF8): boolean; overload;

*Retrieve a JSON serialization of a given ID from cache*



**procedure** FlushCacheAllEntries;

*Flush cache for all Value[]*

**procedure** FlushCacheEntry(Index: Integer);

*Flush cache for a given Value[] index*

**procedure** SetJSON(aID: integer; **const** aJSON: RawUTF8); 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 cache content 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 in 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: cardinal;

*Returns the memory used by JSON serialization records within this cache*

**function** SetCache(aTable: TSQLRecordClass; aID: Integer): 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): 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; aTimeout: Integer): boolean;

*Set the internal caching time out delay (in ms) for a given table*

- 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; overload;

*Flush the cache*

- this will flush all stored JSON content, but keep the settings (SetCache/SetTimeOut) as before

**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(aTable: TSQLRecordClass; aID: integer); 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: integer; const aJSON: RawUTF8; aAction: TSQLOccasion); overload;

*TSQLRest 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: integer; const aJSON: RawUTF8; aAction: TSQLOccasion); overload;

*TSQLRest 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, aID: integer); overload;

*TSQLRest 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: integer); overload;

*TSQLRest instance shall call this method when a record is deleted*

- this method is dedicated for a record deletion

**property** Rest: TSQLRest read fRest;

*Read-only access to the associated TSQLRest instance*

**TSQLRest = class(TObject)**

*A generic REpresentational State Transfer (REST) client/server class*

*Used for DI-2.1.1 (page 828), DI-2.1.1.1 (page 828).*

**QueryCustom: array of TSQLQueryCustom;**

*The custom queries parameters for User Interface Query action*

**constructor** Create(aModel: TSQLModel);

*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): integer; **overload; virtual; abstract;**

*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
- 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

**function** Add(aTable: TSQLRecordClass; **const** aSimpleFields: **array of const**; ForcedID: integer=0): integer; **overload;**

*Create a new member, from a supplied list of field 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")
- 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 internally the Add virtual method above

**function** Delete(Table: TSQLRecordClass; ID: integer): boolean; **overload; virtual;**

*Delete a member (implements REST DELETE Member)*

- return true on success
- this default method call RecordCanBeUpdated() to check if it is possible

```
function Delete(Table: TSQLRecordClass; const SQLWhere: RawUTF8): boolean;  
overload; virtual;
```

*Delete a member with a WHERE clause (implements REST DELETE Member)*

- 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 Delete(Table: TSQLRecordClass; FormatSQLWhere: PUTF8Char; const  
BoundsSQLWhere: array of const): boolean; overload;
```

*Delete a member with a WHERE clause (implements REST DELETE Member)*

- 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
- it will run Delete(Table,FormatUTF8(FormatSQLWhere,[],BoundsSQLWhere))

```
function ExecuteList(const Tables: array of TSQLRecordClass; const SQL:  
RawUTF8): TSQLTableJSON; virtual; abstract;
```

*Execute directly a SQL statement, expecting a list of results*

- return a result table on success, nil on failure

```
function FTSMATCH(Table: TSQLRecordFTS3Class; const WhereClause: RawUTF8; var  
DocID: TIntegerDynArray): boolean; overload;
```

*Dedicated method used to retrieve free-text matching DocIDs*

- this method will work for both TSQLRecordFTS3 and TSQLRecordFTS4
- 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 FTSMATCH(Table: TSQLRecordFTS3Class; const MatchClause: RawUTF8; var  
DocID: TIntegerDynArray; const PerFieldWeight: array of double): boolean;  
overload;
```

*Dedicated method used to retrieve free-text matching DocIDs with enhanced ranking information*

- this method will work for both TSQLRecordFTS3 and TSQLRecordFTS4
- this method will search in all FTS3 columns, and except some floating-point constants for weighing 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](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** MainFieldID(Table: TSQLRecordClass; **const** Value: RawUTF8): integer;

*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 false" definition on most TSQLRecord
- returns 0 if no matching record was found

**function** MainFieldIDs(Table: TSQLRecordClass; **const** Values: **array of** RawUTF8;  
**var** IDs: TIntegerDynArray): 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 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: Integer;  
 ReturnFirstIfNoUnique: boolean=false): RawUTF8;

*Retrieve the main field (mostly 'Name') value of the specified record*

- use Model.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** MultiFieldValue(Table: TSQLRecordClass; **const** FieldName: **array of**  
 RawUTF8; **var** FieldValue: **array of** RawUTF8; WhereID: integer): 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 InternalListJSON() to get the list

**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 InternalListJSON() to get the list

```
function MultiFieldValues(Table: TSQLRecordClass; const FieldNames: RawUTF8;
WhereClauseFormat: PUTF8Char; 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
- if FieldNames="", all simple fields content is retrieved
- 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 internally MultiFieldValues() to get the list

```
function MultiFieldValues(Table: TSQLRecordClass; const FieldNames: RawUTF8;
WhereClauseFormat: PUTF8Char; 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
- if FieldNames="", all simple fields content is retrieved
- 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 internally MultiFieldValues() 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; 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
- if FieldNames="", all simple fields content is retrieved
- call internally InternalListJSON() to get the list
- using inlined parameters via :(...): in WhereClause is always a good idea

```
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.OneFieldValue(TSQLRecord, 'Name', 'ID=(23):')
- you should better call the corresponding overloaded method as such:  
 aClient.OneFieldValue(TSQLRecord, 'Name', 'ID=?', [aID])
- which is the same as calling:  
 aClient.OneFieldValue(TSQLRecord, 'Name', FormatUTF8('ID=?', [], [23]))
- call internally InternalListJSON() to get the value

```
function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8;  
FormatSQLWhere: PUTF8Char; 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.OneFieldValue(TSQLRecord, 'Name', 'ID=?', [aID])
```

- call internally InternalListJSON() 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;  
WhereClauseFmt: PUTF8Char; 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 internally InternalListJSON() to get the value

```
function OneFieldValue(Table: TSQLRecordClass; const FieldName: RawUTF8;  
WhereID: integer): RawUTF8; overload;
```

*Get the UTF-8 encoded value of an unique field from its ID*

- example of use: OneFieldValue(TSQLRecord, 'Name', 23)

- call internally InternalListJSON() to get the value

```
function OneFieldValues(Table: TSQLRecordClass; const FieldName: RawUTF8; const  
WhereClause: RawUTF8; var Data: TIntegerDynArray): boolean; overload;
```

*Get the integer value of an unique field with a Where Clause*

- example of use: OneFieldValue(TSQLRecordPeople, 'ID', 'Name=(:"Smith"):', Data) (using inlined parameters via :(...): is always a good idea)

- leave WhereClause void to get all records

- call internally InternalListJSON() to get the list

```
function OneFieldValues(Table: TSQLRecordClass; const FieldName: RawUTF8; const  
WhereClause: RawUTF8; var 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 InternalListJSON() to get the list

- returns TRUE on success, FALSE if no data was retrieved



```
function OneFieldValues(Table: TSQLRecordClass; const FieldName, WhereClause: RawUTF8; Strings: TStrings; IDToIndex: PInteger=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 InternalListJSON() 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=''; const Separator: RawUTF8=','): RawUTF8; overload;
```

*Get the CSV-encoded 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 InternalListJSON() to get the list
- using inlined parameters via :(...): in WhereClause is always a good idea

```
class function QueryIsTrue(aTable: TSQLRecordClass; aID: integer; 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 Retrieve(const SQLWhere: RawUTF8; Value: TSQLRecord): boolean; overload; virtual;
```

*Get a member from a SQL statement (implements REST GET member)*

- 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 InternalListJSON(), and fill Value from a temporary TSQLTable
- the TSQLRawBlob (BLOB) fields are not retrieved by this method, to preserve bandwidth: use the RetrieveBlob() methods for handling BLOB fields, or set globally the TSQLRestClientURI.ForceBlobTransfert property to TRUE
- 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(WhereClauseFmt: PUTF8Char; **const** Args,Bounds: array of **const**; Value: TSQLRecord): boolean; overload;

*Get a member from a SQL statement (implements REST GET member)*

- 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)

**function** Retrieve(aID: integer; Value: TSQLRecord; ForUpdate: boolean=false): boolean; overload; **virtual**; **abstract**;

*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 is defined as abstract, i.e. there is no default implementation - it must be implemented 100% RestFul with a GET ModelRoot/TableName/ID and handle the LOCK command if necessary: real RESTful class should implement a GET member from URI in an overridden method
- the TSQLRawBlob (BLOB) fields are not retrieved by this method, to preserve bandwidth: use the RetrieveBlob() methods for handling BLOB fields, or set globally the TSQLRestClientURI.ForceBlobTransfert property to TRUE (that is, by default "Lazy loading" is enabled, but can be disabled on purpose)
- 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(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 globally the TSQLRestClientURI.ForceBlobTransfert property to TRUE
- 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 RetrieveBlob(Table: TSQLRecordClass; aID: integer; 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 member with a supplied member ID and a blob 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: integer; const BlobFieldName: RawUTF8; out BlobData: TSQLRawBlob): boolean; overload; virtual; abstract;
```

*Get a blob field content from its record ID and supplied blob field name*

- implements REST GET member 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/ID/BlobFieldName request for example
- this method retrieve the blob data as a TSQLRawBlob string

```
function TableRowCount(Table: TSQLRecordClass): integer; virtual;
```

*Get the row count of a specified table*

- return -1 on error
- return the row count of the table on success
- calls internally the "SELECT Count(\*) FROM TableName;" SQL statement

```
function TransactionBegin(aTable: TSQLRecordClass; SessionID: cardinal): boolean; virtual;
```

*Begin a transaction (implements REST BEGIN Member)*

- to be used to speed up some SQL statements like Add/Update/Delete methods above
- in the current implementation, the aTable parameter is not used yet
- 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 fTransactionActive flag
- return true if no transaction is active, false otherwise
- in a multi-threaded or Client-Server with multiple concurrent Client connections, you should check the returned value, as such:

```
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
- 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)

```
function UnLock(Table: TSQLRecordClass; aID: integer): boolean; overload;  
virtual; abstract;
```

*UnLock the corresponding record*

- use our custom UNLOCK REST-like method
- returns true on success

```
function UnLock(Rec: TSQLRecord): boolean; overload;
```

*UnLock the corresponding record*

- use our custom UNLOCK REST-like method
- calls internally UnLock() above
- returns true on success

```
function Update(aTable: TSQLRecordClass; aID: integer; const aSimpleFields:  
array of const): boolean; overload;
```

*Update a record from a supplied list of field values*

- implements REST PUT Member
- 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")
- the aSimpleFields must have exactly the same count of parameters as there are "simple fields" in the published properties
- return true on success
- call internally the Update virtual method above

```
function Update(Value: TSQLRecord): boolean; overload; virtual;
```

*Update a record from Value fields content*

- implements REST PUT Member
- return true on success
- this default method call RecordCanBeUpdated() to check if the action is allowed - this method must be overridden to provide effective data update
- 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

```
function UpdateBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName:  
RawUTF8; BlobData: pointer; BlobSize: integer): boolean; overload;
```

*Update a blob field from its record ID and blob field name*

- implements REST PUT member with a supplied member ID and field name
- return true on success
- this default method call RecordCanBeUpdated() to check if the action is allowed
- this method expect the Blob data to be supplied as direct memory pointer and size

```
function UpdateBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName:  
RawUTF8; const BlobData: TSQLRawBlob): boolean; overload; virtual; abstract;
```

*Update a blob field from its record ID and supplied blob field name*

- implements REST PUT member with a supplied member ID and field name
- return true on success
- this default method call RecordCanBeUpdated() to check if the action is allowed
- this method expect the Blob data to be supplied as TSQLRawBlob

**function** UpdateBlob(Table: TSQLRecordClass; aID: integer; **const** BlobFieldName: RawUTF8; BlobData: TStream): boolean; overload;

*Update a blob field from its record ID and blob field name*

- implements REST PUT member with a supplied member ID and field name
- return true on success
- this default method call RecordCanBeUpdated() to check if the action is allowed
- 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

**procedure** Commit(SessionID: cardinal); **virtual**;

*End a transaction (implements REST END Member)*

- write all pending SQL statements to the disk
- default implementation just reset the protected fTransactionActive 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)

**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 Member)*

- restore the previous state of the database, before the call to TransactionBegin
- default implementation just reset the protected fTransactionActive 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)

**property** AcquireWriteTimeOut: cardinal **read** fAcquireWriteTimeOut **write** fAcquireWriteTimeOut;

*The time (in mili seconds) which the server will wait for acquiring a write access to the database*

- 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 2000, i.e. TSQLRestServer.URI will wait up to 2 seconds in order to acquire the right to write on the database before returning a "408 Request Time-out" status error

**property** Cache: TSQLRestCache read GetCache;

*Access the internal caching parameters for a given TSQLRecord*

- 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 direct RESTful/CRUD commands: RETRIEVE, ADD, UPDATE and DELETE (that is, a complex direct SQL UPDATE or via TSQLRecordMany pattern won't be taken in account - only exception is TSQLRestServerStatic 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 code won't fail due to this restriction
- use Cache.SetCache() and Cache.SetTimeout() methods to set the appropriate configuration for this particular TSQLRest instance

**property** Model: TSQLModel read fModel;

*The Database Model associated with this REST Client or Server*

**property** ServerTimeStamp: TTimeLog read GetServerTimeStamp write SetServerTimeStamp;

*The current Date and Time, as retrieved from the server*

- this property will return the timestamp as TTimeLog / Iso8601 / 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 time, i.e. Iso8601Now
- you can set the server-side time offset by setting a value to this property (e.g. using TSQLDBConnection.ServerTimeStamp property for Oracle, MSSQL and MySQL external databases)

**property** Services: TServiceContainer read fServices;

*Access to the interface-based services list*

- may be nil if no service interface has been registered yet

**property** ServicesRouting: TServiceRoutingMode read fRouting write fRouting;

*The routing mode of the service remote request*

- by default, will use an URI-based layout (rmREST), which is more secure (since will use our RESTful authentication scheme), and also 10% faster
- but rmJSON\_RPC can be set (on BOTH client and server sides), if the client would rather use this alternative pattern

**TSQLRestServerNamedPipe = class(TThread)**

*Server thread accepting connections from named pipes*

**constructor** Create(aServer: TSQLRestServer; const PipeName: TFileName);

*Create the server thread*

**destructor** Destroy; **override;**

*Release all associated memory, and wait for all TSQLRestServerNamedPipeResponse children to be terminated*

**property** PipeName: TFileName read fPipeName;

*The associated pipe name*

**TSQLRestServerNamedPipeResponse = class(TThread)**

*Server child thread dealing with a connection through a named pipe*

**constructor** Create(aServer: TSQLRestServer; aMasterThread:  
TSQLRestServerNamedPipe; aPipe: cardinal);

*Create the child connection thread*

**destructor** Destroy; **override**;

*Release all associated memory, and decrement fMasterThread.fChildCount*

**TSQLRestServerStats = class(TPersistent)**

*If defined, the server statistics will contain precise working time process used for statistics update in TSQLRestServer.URI()*

**ProcessTimeCounter: Int64;**

*High-resolution performance counter of the time used to process the requests*

- this value depend on the high-resolution performance counter frequency
- use ProcessTime property below to get the time in seconds

**function** Changed: boolean;

*Return true if IncomingBytes value changed since last call*

**function** DebugMessage: RawUTF8;

*Get a standard message to be displayed with the above statistics*

- return the published properties of this class as a JSON object

**function** ModifPercent: cardinal;

*Percent (0..100) of request which modified the data*

**procedure** ClientConnect;

*Update CLientsCurrent and CLientsMax*

**procedure** ClientDisconnect;

*Update CLientsCurrent and CLientsMax*

**property** ClientsCurrent: QWord read fClientsCurrent;

*Current count of connected clients*

**property** ClientsMax: QWord read fClientsMax;

*Max count of connected clients*

**property** IncomingBytes: QWord read fIncomingBytes;

*Size of data requests processed in bytes (without the transfert protocol overhead)*

**property** Invalid: QWord read fInvalid;

*Count of invalid request*

**property** Modified: QWord read fModified;

*Count of requests which modified the data*

**property** OutcomingBytes: QWord read fOutcomingBytes;

*Size of data responses generated in bytes (without the transfert protocol overhead)*

**property** ProcessTime: RawUTF8 read GetProcessTimeString;

*The global time spent in the server process*

**property** Responses: QWord read fResponses;

*Count of valid responses (returned status code 200/HTML\_SUCCESS or 201/HTML\_CREATED)*

**property** ServiceCalls: QWord read fServices;

*Count of the remote service calls*

**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) is always valid, 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** ToString: RawUTF8;

*Serialize the content as TEXT*

- use the TSQLAuthGroup.AccessRights CSV format

**procedure** Edit(aTableIndex: integer; C, R, U, D: Boolean);

*Wrapper method which can be used to set the CRUD abilities over a table*

- C=Create, R=Read, U=Update, D=Delete rights

```
procedure FromString(P: PUTF8Char);
```

*Unserialize the content from TEXT*  
- use the TSQLAuthGroup.AccessRights CSV format

```
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  
- by default, it won't be accessible remotely by anyone

```
class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8); 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 'synapse' as default password), and associated 'Admin', 'Supervisor', 'User' and 'Guest' groups, with the following access rights to the AuthGroup table:

|            | POST | SQL | Service | Auth R | Auth W | Tables R | Tables W |
|------------|------|-----|---------|--------|--------|----------|----------|
| Admin      | Yes  | Yes | Yes     | Yes    | Yes    | Yes      | Yes      |
| Supervisor | No   | Yes | Yes     | No     | Yes    | Yes      | Yes      |
| User       | No   | Yes | No      | No     | Yes    | Yes      | Yes      |
| Guest      | No   | No  | No      | No     | Yes    | No       | No       |

'Admin' will be the only able to execute remote not SELECT SQL statements for POST commands (reSQL in TSQLAccessRights.AllowRemoteExecute) and modify the Auth tables (i.e. AuthUser and AuthGroup), and Guest won't have access to the interface-based remote JSON-RPC service (no reService)

- you MUST override those default 'synapse' password to a custom value  
- 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 table

```
property AccessRights: RawUTF8 read fAccessRights write fAccessRights;
```

*A textual representation of a TSQLAccessRights buffer*

```
property Ident: RawUTF8 read fIdent write fIdent stored false;
```

*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 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
- by default, it won't be accessible remotely by anyone
- to enhance security, you could use the TSynValidatePassWord filter to this table

**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 **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 **read** fLogonName **write** fLogonName **stored** false;

*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 false" attribute), and therefore indexed

**property** PasswordHashHexa: RawUTF8 **read** fPasswordHashHexa **write** fPasswordHashHexa;

*The hexa encoded associated SHA-256 hash of the password*

**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

## **TAuthSession = class(TObject)**

*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)

**constructor** Create(aServer: TSQLRestServer; aUser: TSQLAuthUser);

*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*

**function** IsValidURL(const aURL: RawUTF8; aURLlength: integer): boolean;

*Check if the session\_signature=... parameter is correct*

- session\_signature=... is expected at the end of the URL, i.e. aURL[aURLlength+1] will point e.g. to '?session\_signature=...': the caller must ensure that aURL[] follows this expected layout
- will expect the format as generated by TSQLRestClientURI.SessionSign()

**property** AccessRights: TSQLAccessRights **read** fAccessRights;

*Copy of the associated user access rights*

- extracted from User.TSQLAuthGroup.SQLAccessRights

**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 handshaking phase)

**property** LastAccess: cardinal **read** fLastAccess;

*Set by the Access method to the current time stamp*

**property** PrivateKey: RawUTF8 **read** fPrivateKey;

*The hexadecimal private key as returned to the connected client as 'SessionID+PrivateKey'*

**property** Timeout: cardinal **read** fTimeOut;

*The number of milliseconds a session is kept alive*

- extracted from User.TSQLAuthGroup.SessionTimeout
- allow direct comparison with GetTickCount API call

**property** User: TSQLAuthUser **read** fUser;

*The associated User*

- this is a true TSQLAuthUser instance, and User.GroupRights will contain also a true TSQLAuthGroup instance

**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 828), DI-2.1.1.1 (page 828), DI-2.1.1.2.1 (page 829), DI-2.1.1.2.2 (page 829), DI-2.1.1.2.3 (page 829), DI-2.1.1.2.4 (page 830), DI-2.2.1 (page 832).*

**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

**OnUpdateEvent:** TNotifySQLEvent;

*A method can be specified here to trigger events after any table update*

- is called BEFORE deletion, and AFTER insertion or update
- 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)

**constructor** Create(aModel: TSQLModel; aHandleUserAuthentication: boolean=false);  
**reintroduce;**

*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)

**destructor** Destroy; **override;**

*Release memory and any existing pipe initialized by ExportServer()*

```
function Auth(var aParams: TSQLRestServerCallBackParams): Integer;
```

*This method will be accessible from ModelRoot/Auth URI, and will be called by the client for authentication and session management*

- method parameters signature matches TSQLRestServerCallBack type
- 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 OK, open the corresponding session
    and returns '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:

```
ModelRoot/url?A=1&B=2&session_signature=012345670123456701234567
```

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 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](http://www.webtoolkit.info/javascript-sha256.html)
- this global callback method is thread-safe

```
function Batch(var aParams: TSQLRestServerCallBackParams): Integer;
```

*This method will be accessible from the ModelRoot/Batch URI, and will execute a set of RESTful commands*

- expect input as JSON commands - see TSQLRestServer.RunBatch, i.e.

```
'{"Table":["cmd":values,...}]'
```

or for multiple tables:

```
'["cmd@Table":values,...]'
```

with cmd in POST/PUT with {object} as value or DELETE with ID

- only accepted context HTTP verb is PUT (for thread-safe and security reasons)

**function** CacheFlush(**var** aParams: TSQLRestServerCallBackParams): Integer;

*This method will be accessible from the ModelRoot/CacheFlush URI, and will flush the server cache*

- this method shall be called by the clients when the Server cache could be not refreshed
- ModelRoot/CacheFlush URI will flush the whole Server cache, for all tables
- ModelRoot/CacheFlush/TableName URI will flush the specified table cache
- ModelRoot/CacheFlush/TableName/ID URI will flush the content of the specified record
- method parameters signature matches TSQLRestServerCallBack type

**function** Stat(**var** aParams: TSQLRestServerCallBackParams): Integer;

*This method will be accessible from ModelRoot/Stat URI, and will retrieve some statistics as a JSON object*

- method parameters signature matches TSQLRestServerCallBack type

**function** TimeStamp(**var** aParams: TSQLRestServerCallBackParams): Integer;

*This method will be accessible from the ModelRoot/TimeStamp URI, and will return the server time stamp TTimeLog/Int64 value as RawUTF8*

- method parameters signature matches TSQLRestServerCallBack type

**function** Add(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false): integer; **override**;

*Implement Server-Side TSQLRest adding*

- uses internally EngineAdd() function for calling the database engine
- call corresponding fStaticData[] if necessary
- on success, returns the new ROWID value; on error, returns 0
- on success, Value.ID is updated with the new ROWID
- if aValue is TSQLRecordFTS3, Value.ID is stored to the virtual table

**function** AfterDeleteForceCoherency(Table: TSQLRecordClass; aID: integer): boolean; **virtual**;

*This method is called internally after any successfull deletion to ensure relational database coherency*

- delete all available TRecordReference properties pointing to this record in the database Model, for database coherency
- delete all available TSQLRecord properties pointing to this record in the database Model, for database coherency
- 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** CloseServerMessage: boolean;

*End any currently initialized message-oriented server*

**function** CloseServerNamedPipe: boolean;

*End any currently initialized named pipe server*

**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 FieldNames at once*

```
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: integer): 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 EngineExecuteAll(const aSQL: RawUTF8): boolean; virtual; abstract;
```

*Execute directly all SQL statement (POST SQL on ModelRoot URI)*

- return true on success
- override this method for proper calling the database engine
- don't call this method in normal cases
- this method must be implemented to be thread-safe

```
function ExecuteList(const Tables: array of TSQLRecordClass; const SQL: RawUTF8): TSQLTableJSON; override;
```

*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 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 829).*

**function** ExportServerNamedPipe(const ServerApplicationName: TFileName): boolean;

*Declare the server on the local machine as a Named Pipe: allows TSQLRestClientURINamedPipe 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\Sqlite3\_' to obtain the full pipe name to initiate ('\\.\pipe\Sqlite3\_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 829).*

**class function** JSONEncodeResult(const OneValue: array of const): RawUTF8;

*Encode some value into a JSON "result": "value" UTF-8 encoded content*

- wrapper around standard JSONEncode() function
- OneValue usually have only ONE parameter, e.g.:  

```
result := JSONEncodeResult([Value]);
```
- returned as '"result":100' if Value=100
- if OneValue has more than one parameter, returns a JSON array of values like  

```
"result":["value1",value2]"
```
- this method will work outside of a true TSQLRestServer instance: you can use e.g.  

```
TSQLRestServer.JSONEncodeResult(['value1',10])
```

**function** Retrieve(aID: integer; Value: TSQLRecord; ForUpdate: boolean=false): boolean; **override**;

*Implement Server-Side TSQLRest Retrieval (GET or LOCK methods)*

- uses internally EngineRetrieve() function for calling the database engine (via fStaticData[] if the table is stored as Static)
- handles locking if necessary
- 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 RetrieveBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName: RawUTF8; out BlobData: TSQLRawBlob): boolean; override;
```

*Implement Server-Side TSQLRest blob field content retrieval*

- implements REST GET member with a supplied member ID and a blob field name
- uses internally EngineRetrieveBlob() function for calling the database engine
- call corresponding fStaticData[] if necessary
- this method retrieve the blob data as a TSQLRawBlob string

```
function ServiceRegister(aClient: TSQLRest; const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceImplementation=sicSingle; const aContractExpected: RawUTF8=''): boolean; overload; virtual;
```

*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 ServiceRegister(aImplementationClass: TClass; const aInterfaces: array of PTypeInfo; aInstanceCreation: TServiceImplementation=sicSingle): TServiceFactoryServer; overload; virtual;
```

*Register a Service 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
- 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 StaticDataCreate(aClass: TSQLRecordClass; const aFileName: TFileName =  
''; aBinaryFile: boolean=false; aServerClass: TSQLRestServerStaticClass=nil):  
TSQLRestServerStatic;
```

*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 TSQLRestServerStaticInMemory instance
- this data handles basic REST commands, since no complete SQL interpreter can be implemented by TSQLRestServerStatic; 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

```
function UnLock(Table: TSQLRecordClass; aID: integer): boolean; override;
```

*Implement Server-Side TSQLRest unlocking*

- implements our custom UNLOCK REST-like method
- locking is handled by TSQLServer.Model
- returns true on success

```
function Update(Value: TSQLRecord): boolean; override;
```

*Implement Server-Side TSQLRest update*

- uses internally EngineUpdate() function for calling the database engine
- call corresponding fStaticData[] if necessary

```
function UpdateBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName:  
RawUTF8; const BlobData: TSQLRawBlob): boolean; override;
```

*Implement Server-Side TSQLRest blob field content update*

- implements REST PUT member with a supplied member ID and field name
- uses internally EngineUpdateBlob() function for calling the database engine
- this method expect the blob data to be supplied as a TSQLRawBlob string

```
function URI(const url, method, SentData: RawUTF8; out Resp, Head: RawUTF8;  
RestAccessRights: PSQLAccessRights): Int64Rec; virtual;
```

*Implement a generic local, piped or HTTP/1.1 provider*

- this is the main entry point of the server, from the client side
- some GET/POST/PUT JSON data can be specified in SentData
- return in result.Lo the HTTP STATUS integer error or success code: 404/HTML\_NOTFOUND e.g. if the url doesn't start with Model.Root (caller can try another TSQLRestServer)
- return in result.Hi the database internal status
- store the data to be sent into Resp, some headers in Head
- default implementation calls protected methods EngineList() Retrieve() Add() Update() Delete() UnLock() EngineExecute() above, which must be overridden by the TSQLRestServer child
- see TSQLRestClient to check how data is expected in our RESTful format
- 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
- handle enhanced REST codes: LOCK/UNLOCK/BEGIN/END/ABORT
- 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>

*Used for DI-2.1.1.2.4 (page 830).*

```
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 BeginCurrentThread(Sender: TThread); virtual;
```

*You can call this method in TThread.Execute to ensure that the thread will be taken in account during process*

- caller must specify the TThread instance running
- used e.g. for execInMainThread option in TServiceMethod.InternalExecute
- this default implementation will call the methods of all its internal TSQLRestServerStatic 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); override;
```

*End a transaction (implements REST END Member)*

- write all pending TSQLVirtualTableJSON data to the disk

**procedure** EndCurrentThread(Sender: TObject); **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 default implementation will call the methods of all its internal TSQLRestServerStatic instances, allowing e.g. TSQLRestServerStaticExternal 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. TSQLRestServerStaticExternal classes defined in SQLite3DB) could flush the database content without proper notification
- this default implementation just do nothing, but SQLite3 unit will call TSQLDataBase.CacheFlush method

**procedure** ServiceMethodByPassAuthentication(const aMethodName: RawUTF8);

*Call this method to disable Authentication method check for a given published method 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)

**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** NoAJAXJSON: boolean **read** fNoAJAXJSON **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 environnement (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

**property** StaticDataServer[aClass: TSQLRecordClass]: TSQLRestServerStatic **read** GetStaticDataServer;

*Retrieve the TSQLRestServerStatic instance used to store and manage a specified TSQLRecordClass in memory*  
- has been associated by the StaticDataCreate method

**property** StaticVirtualTable[aClass: TSQLRecordClass]: TSQLRestServerStatic **read** GetVirtualTable;

*Retrieve a running TSQLRestServerStatic virtual table*  
- associated e.g. to a 'JSON' or 'Binary' virtual table module, or may return a TSQLRestServerStaticExternal instance (as defined in SQLite3DB)  
- 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 would be accessible only via StaticDataServer[], not via StaticVirtualTable[])  
- has been associated by the TSQLModel.VirtualTableRegister 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 TSQLRestServerStatic 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** Stats: TSQLRestServerStats **read** fStats;

*Access to the Server statistics*

**TSQLRestServerStatic = class**(TSQLRestServer)

*REST server 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 (like our TSQLRestServerStaticInMemory class)

**constructor** Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; **const** aFileName: TFileName = ''; aBinaryFile: boolean=false); **virtual**;

*Initialize the server 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 (this virtual method will just ignore this parameter, which will be used for overridden constructor only)

**function** EngineExecuteAll(**const** aSQL: RawUTF8): boolean; **override**;

*Overriden method for direct in-memory database engine call*  
- not implemented: always return false  
- this method must be implemented to be thread-safe

```
function EngineUpdateField(Table: TSQLRecordClass; const SetFieldName, SetValue,
WhereFieldName, WhereValue: RawUTF8): boolean; override;
```

*Overriden method for direct in-memory database engine call*

- made public since a TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer
- do nothing method: will return FALSE (aka error)

```
function SearchField(const FieldName: ShortString; const FieldValue: RawUTF8;
var ResultID: TIntegerDynArray): 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 SearchField(const FieldName: ShortString; const FieldValue: Integer;
var ResultID: TIntegerDynArray): 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

```
property FileName: TFileName read fFileName write fFileName;
```

*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 instanced static table

```
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 Server using this in-memory database*

```
property StoredClass: TSQLRecordClass read fStoredClass;
```

*Read only access to the class defining the record type stored in this REST server*

```
property StoredClassProps: TSQLRecordProperties read fStoredClassProps;
```

*Read only access to the class properties of the associated record type*

```
TSQLRestServerStaticRecordBased = class(TSQLRestServerStatic)
```

*Abstract REST server exposing some internal TSQLRecord-based methods*

```
function AddOne(Rec: TSQLRecord): integer; 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: integer): 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 TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** UpdateOne(ID: integer; **const** Values: TVarDataDynArray): boolean;  
**overload; virtual;**

*Manual Update of a TSQLRecord field values from TVarData 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 TSQLRestServerStatic 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): 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 TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**TListFieldHash = class**(TObjectHash)

*Class able to handle a O(1) hashed-based search of a property in a TList*

- used e.g. to hash TSQLRestServerStaticInMemory field values

**constructor** Create(aValues: TList; aFieldIndex: integer; aField: PPropInfo;  
 aCaseInsensitive: boolean);

*Initialize a hash for a record array field*

- aFieldIndex/aField parameters correspond to the indexed field (e.g. stored false published property)
- if CaseInsensitive is TRUE, will apply NormToUpper[] 8 bits uppercase, handling RawUTF8 properties just like the SYSTEMNOCASE collation

**property** CaseInsensitive: boolean **read** fCaseInsensitive;

*If the string comparison shall be case-insensitive*

**property** Field: PPropInfo **read** fProp;

*The corresponding field RTTI*

**property** FieldIndex: integer **read** fField;

*The corresponding field index in the TSQLRecord*



**TSQLRestServerStaticInMemory = class(TSQLRestServerStaticRecordBased)**

*REST server with direct access to a memory-stored database*

- store the associated TSQLRecord values in memory
- handle only one TSQLRecord by server (it's NOT a true Rest Server)
- 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 TSQLRestServerStatic 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 TSQLRestServerStaticInMemory instance

**constructor** Create(aClass: TSQLRecordClass; aServer: TSQLRestServer; **const** aFileName: TFileName = ''; aBinaryFile: boolean=false); **override;**

*Initialize the server 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

**destructor** Destroy; **override;**

*Free used memory*

- especially release all fValue[] instances

**function** AddOne(Rec: TSQLRecord): integer; **override;**

*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** EngineDelete(Table: TSQLRecordClass; ID: integer): boolean; **override;**

*Overriden method for direct in-memory database engine call*

- made public since a TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** EngineUpdateField(Table: TSQLRecordClass; **const** SetFieldName, SetValue, WhereFieldName, WhereValue: RawUTF8): boolean; **override;**

*Overriden method for direct in-memory database engine call*

- made public since a TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** GetOne(aID: integer): TSQLRecord; **override**;

*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 TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** IDToIndex(ID: integer): integer;

*Retrieve the index in Items[] of a particular ID*

- return -1 if this ID was not found
- use fast binary search algorithm (since Items[].ID should be increasing)

**function** LoadFromBinary(Stream: TStream): boolean;

*Load the values from binary data*

- 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** SaveToBinary(Stream: TStream): integer;

*Save the values into binary data*

- 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** SaveToJSON(Expand: Boolean): RawUTF8; **overload**;

*Save the values into JSON data*

**function** SearchField(const FieldName: ShortString; const FieldValue: RawUTF8;  
**var** ResultID: TIntegerDynArray): 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** UpdateOne(ID: integer; const Values: TVarDataDynArray): boolean;  
**override**;

*Manual Update of a TSQLRecord field values from TVarData 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 TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**function** UpdateOne(Rec: TSQLRecord): 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 TSQLRestServerStatic instance may be created stand-alone, i.e. without any associated Model/TSQLRestServer

**procedure** LoadFromJSON(const aJSON: RawUTF8); **overload**;

*Load the values from JSON data*

**procedure** LoadFromJSON(JSONBuffer: PUTF8Char; JSONBufferLen: integer); **overload**;

*Load the values from JSON data*

**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 TSQLRestServerStatic instance is destroyed: should should want to call in in some cases, in order to force the data to be saved regularly
- 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** fBinaryFile;

*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

**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** GetCount;

*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)

**property** ID[Index: integer]: integer **read** GetID;

*Read-only access to the ID of a TSQLRecord values*

**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

**TSQLRestServerStaticInMemoryExternal = class(TSQLRestServerStaticInMemory)**

*REST server with direct access to a memory database, to be used as external table*

- this is the kind of in-memory table expected by TSQLVirtualTableJSON, in order to be consistent with the internal DB cache

**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 TSQLVirtualTableJSON virtual tables could flush the database content without proper notification

- this overridden implementation will call Owner.FlushInternalDBCache

**TSQLRestServerFullMemory = class(TSQLRestServer)**

*A REST server using only in-memory tables*

- this server will use TSQLRestServerStaticInMemory 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)

**constructor** Create(aModel: TSQLModel; **const** aFileName: TFileName='';  
 aBinaryFile: boolean=false; aHandleUserAuthentication: boolean=false);  
**reintroduce; virtual;**

*Initialize a REST server with a database file*

- all classes of the model will be created as TSQLRestServerStaticInMemory

- then data persistence will be created using aFileName

- if aFileName is left void (''), data will not be persistent

**destructor Destroy; override;**

*Write any modification on file (if needed), and release all used memory*

**function** EngineExecuteAll(**const** aSQL: RawUTF8): boolean; **override;**

*Overriden method for direct in-memory database engine call*

- not implemented: always return false

**procedure LoadFromFile; virtual;**

*Load the content from the specified file name*

- do nothing if file name was not assigned

**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 TSQLRestServerStaticInMemory LoadFromJSON/LoadFromBinary  
 SaveToJSON/SaveToBinary methods for optimized storage

**property** FileName: TFileName read fFileName write fFileName;

*The file name used for data persistence*

**TSQLRestServerRemoteDB = class(TSQLRestServer)**

*A REST server using a TSQLRestClient for all its ORM process*

- this server will use an internal TSQLRestClient instance to handle all ORM operations (i.e. access to objects)
- 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)

**constructor** Create(aRemoteClient: TSQLRestClient; aHandleUserAuthentication: boolean=false); **reintroduce; virtual;**

*Initialize a REST server associated to a given TSQLRestClient instance*

- the specified TSQLRestClient will be used for all ORM and data process
- the supplied TSQLRestClient.Model will be used for TSQLRestServerRemoteDB
- note that the TSQLRestClient instance won't be freed - caller shall manage its life time

**function** AfterDeleteForceCoherency(Table: TSQLRecordClass; aID: integer): 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

**function** EngineExecuteAll(const aSQL: RawUTF8): boolean; **override;**

*Implement Server-Side TSQLRest deletion overridden method for remote database engine call*

- will return false - i.e. not implemented - since it is a server side operation

**function** ExecuteList(const Tables: array of TSQLRecordClass; const SQL: RawUTF8): TSQLTableJSON; **override;**

*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

**property** Client: TSQLRestClient read fClient;

*The remote ORM client used for data persistence*

**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** Add(Value: TSQLRecord; SendData: boolean; ForceID: boolean=false): integer; **override**;

*Create a new member (implements REST POST Collection)*

- URI is 'ModelRoot/TableName' with POST method
- 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
- server must return Status 201/HTML\_CREATED on success
- server must send on success an header entry with 'Location: ModelRoot/TableName/ID'
- on success, returns the new ROWID value; on error, returns 0
- on success, Value.ID is updated with the new ROWID
- if aValue is TSQLRecordFTS3, Value.ID is stored to the virtual table

**function** Delete(Table: TSQLRecordClass; ID: integer): boolean; **override**;

*Delete a member (implements REST DELETE Collection/Member)*

- URI is 'ModelRoot/TableName/ID' with DELETE method
- server must return Status 200/HTML\_SUCCESS OK on success

**function** EngineExecute(const SQL: RawUTF8): boolean; **overload**; **virtual**; **abstract**;

*Execute directly a SQL statement*

- return true on success

**function** EngineExecuteFmt(SQLFormat: PUTF8Char; const Args: array of const): boolean; **overload**;

*Execute directly a SQL statement with supplied parameters*

- expect the same format as FormatUTF8() function, replacing all '%' chars with Args[] values
- return true on success

**function** EngineExecuteFmt(SQLFormat: PUTF8Char; const Args, Bounds: array of const): boolean; **overload**;

*Execute directly a SQL statement with supplied parameters*

- expect the same format as FormatUTF8() function, replacing all '%' chars with Args[] values, and all '?' chars with Bounds[] (inlining them with :(...): and auto-quoting strings)
- return true on success

**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:
RawUTF8; SQLWhereFormat: PUTF8Char; 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 ListFmt(const Tables: array of TSQLRecordClass; const SQLSelect:
RawUTF8; SQLWhereFormat: PUTF8Char; 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 Refresh(aID: integer; Value: TSQLRecord; var Refreshed: boolean):
boolean;
```

*Get a member from its ID (implements REST GET Collection/Member)*

- URI is 'ModelRoot/TableName/ID' with GET method
- returns true on server returned 200/HTML\_SUCCESS OK success, false on error
- set Refreshed to true if the content changed

```
function Retrieve(aID: integer; Value: TSQLRecord; ForUpdate: boolean=false):
boolean; override;
```

*Get a member from its ID (implements REST GET Collection/Member)*

- URI is 'ModelRoot/TableName/ID' with GET method
- server must return Status 200/HTML\_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 RetrieveBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName:
RawUTF8; out BlobData: TSQLRawBlob): boolean; override;
```

*Get a blob field content from its record ID and supplied blob field name*

- implements REST GET member with a supplied member ID and a blob field name
- URI is 'ModelRoot/TableName/ID/BlobFieldName' with GET method
- server must return Status 200/HTML\_SUCCESS OK on success
- this method retrieve the blob data as a TSQLRawBlob string

```
function RetrieveBlobFields(Value: TSQLRecord): boolean;
```

*Get all BLOB fields of the supplied value from the remote server*

- call internally by Retrieve method when ForceBlobTransfert is TRUE



```
function RTreeMatch(DataTable: TSQLRecordClass; const DataTableBlobFieldName:
RawUTF8; RTreeTable: TSQLRecordRTreeClass; const DataTableBlobField:
RawByteString; var DataID: TIntegerDynArray): 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

- 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 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

```
function Update(Value: TSQLRecord): boolean; override;
```

*Update a member (implements REST PUT Collection/Member)*

- URI is 'ModelRoot/TableName/ID' with PUT method

- server must return Status 200/HTML\_SUCCESS OK on success

```
function UpdateBlob(Table: TSQLRecordClass; aID: integer; const BlobFieldName:
RawUTF8; const BlobData: TSQLRawBlob): boolean; override;
```

*Update a blob field from its record ID and supplied blob field name*

- implements REST PUT member with a supplied member ID and field name

- URI is 'ModelRoot/TableName/ID/BlobFieldName' with PUT method

- server must return Status 200/HTML\_SUCCESS OK on success

- this method expect the blob data to be supplied as a TSQLRawBlob string

```
function UpdateBlobFields(Value: TSQLRecord): boolean;
```

*Update all BLOB fields of the supplied Value*

- uses the UpdateBlob() method to send the BLOB properties content to the Server

- called internally by Add and Update methods when ForceBlobTransfert is TRUE

- you can use this method by hand, to avoid several calls to UpdateBlob()

```
procedure Commit(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED); 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** fForceBlobTransfert **write** fForceBlobTransfert;

*Is set to TRUE, all BLOB fields are 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

**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.)*

**TSQLRestClientURI = class(TSQLRestClient)**

*A generic REpresentational State Transfer (REST) client with URI*

- URI are standard Collection/Member implemented as ModelRoot/TableName/ID
- handle RESTful commands GET POST PUT DELETE LOCK UNLOCK

*Used for DI-2.1.1 (page 828), DI-2.1.1.1 (page 828), DI-2.1.1.2.2 (page 829), DI-2.1.1.2.3 (page 829), DI-2.1.1.2.4 (page 830).*

**destructor** Destroy; **override**;

*Release memory, and unLock all still locked records by this client*

**function** BatchAdd(Value: TSQLRecord; SendData: boolean; ForceID: 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, Value.ID is stored to the virtual table
- Value class MUST match the TSQLRecordClass used at BatchTransactionBegin, or may be of any kind if no class was specified
- BLOB fields are NEVER transmitted here, even if ForceBlobTransfert=TRUE

**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: integer): 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 BatchTransactionBegin() call: it will fail if no class was specified for this BATCH sequence

**function** BatchDelete(Table: TSQLRecordClass; ID: integer): 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 BatchTransactionBegin() call

**function** BatchSend(var Results: TIntegerDynArray): integer;

*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 method
- will return the URI Status value, i.e. 200/HTML\_SUCCESS OK on success
- a dynamic array of integers will be created in Results, containing all ROWID created for each BatchAdd call, 200 (=HTML\_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): boolean;

*Begin a BATCH sequence to speed up huge database change*

- 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

**function** BatchUpdate(Value: TSQLRecord): integer;

*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 BatchTransactionBegin, 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)

**function** CallbackGet(const aMethodName: RawUTF8; const aParameters: array of const; out aResponse: RawUTF8; aTable: TSQLRecordClass=nil; aID: integer=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/HTML\_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 aParameters: array of const; aTable: TSQLRecordClass=nil; aID: integer=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: integer=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/HTML\_SUCCESS on success)
- this version will use a PUT with the supplied raw UTF-8 data

**function** EngineExecute(const SQL: RawUTF8): boolean; **override;**

*Execute directly a SQL statement*

- URI is 'ModelRoot' with POST method, and SQL statement sent as UTF-8
- server must return Status 200/HTML\_SUCCESS OK on success

**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/HTML\_SUCCESS OK on success

**function** ServerCacheFlush(aTable: TSQLRecordClass=nil; aID: integer=0): boolean;

*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** ServiceRegister(const aInterfaces: array of PTypeInfo;  
aInstanceCreation: TServiceInstanceImplementation=sicSingle; const  
aContractExpected: RawUTF8=''): boolean; **virtual**;

*Register a Service on the client side via its interface*

- 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** 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
- returns true on success
- calling this method is optional, depending on your user right policy: your Server need to handle authentication
- 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)

```
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 Add/Update/Delete methods above
- 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 can should better use the dedicated TransactionBeginRetry() method in case of Client concurrent access

```
function TransactionBeginRetry(aTable: TSQLRecordClass; Retries: integer=10):  
boolean;
```

*Begin a transaction (implements REST BEGIN Member)*

- this version retries a TransactionBegin() to be successfull 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: integer): boolean; override;
```

*UnLock the corresponding record*

- URI is 'ModelRoot/TableName/ID' with UNLOCK method
- returns true on success

```
function UpdateFromServer(const Data: array of TObjet; 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

```
procedure Commit(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED); override;
```

*End a transaction (implements REST END Member)*

- write all pending SQL statements to the disk

```
procedure RollBack(SessionID: cardinal=CONST_AUTHENTICATION_NOT_USED); override;
```

*Abort a transaction (implements REST ABORT Member)*

- restore the previous state of the database, before the call to TransactionBegin

```
property MaximumAuthenticationRetry: Integer read
```

```
fMaximumAuthenticationRetry write fMaximumAuthenticationRetry;
```

*Maximum additional retry occurrence*

- default is 0, 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 HTML\_FORBIDDEN "403 Forbidden" error code

```
property OnSetUser: TNotifyEvent 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 SessionUser property to retrieve the current authenticated user, or nil if authentication failed
- could be used to refresh the User Interface layout according to current authenticated user rights

```
property SessionUser: TSQLAuthUser read fSessionUser;
```

*The current user as set by SetUser() method*

- returns nil if no User was currently authenticated
- only available fields by default are LogonName and PasswordHashHexa



### **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 829).*

**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 829).*

**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 829).*

**destructor** Destroy; override;

*Release memory and handles*

### **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

*Used for DI-2.1.1.2.3 (page 829).*

**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 829).*

**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 829).*

**destructor** Destroy; **override**;

*Release the internal Window class created, if any*

**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 window

**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

*Used for DI-2.1.1.2.2 (page 829).*

**constructor** Create(aModel: TSQLModel; **const** ApplicationName: TFileName);

*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\Sqlite3\_' to obtain the full pipe name to connect to ('\\.\pipe\Sqlite3\_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 829).*

**destructor** Destroy; **override**;

*Release memory and handles*

**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 Process method call by TSQLRecord.Validate()

**property** ProcessRec: TSQLRecord **read** fProcessRec;

*The associated TSQLRecord instance*

- this value is updated by TSQLRecord.Validate with the current TSQLRecord instance to be validated

- it can be used in the overridden Process method

**property** ProcessRest: TSQLRest read fProcessRest;

*The associated TSQLRest instance*

- this value is updated by TSQLRecord.Validate with the current TSQLRest used for the validation
- it can be used in the overridden Process method

**TSynValidateUniqueField = class(TSynValidateRest)**

*Will define a validation for a TSQLRecord Unique field*

- 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 ProcessRest and ProcessRec properties, as set by TSQLRecord.Validate

**TSQLVirtualTablePreparedConstraint = 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

**Operator: 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'...)

**Value: TVarData;**

*The associated expression*

- TSQLVirtualTable.Prepare() must set this property VType to <> varEmpty (=0) e.g. to varAny, if an expression is expected at TSQLVirtualTableCursor.Search() 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*

**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: Double;**

*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

**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;**

*Numver 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:** TSQLRestServerStaticClass;

*The associated TSQLRestServerStatic class used for storage*  
 - is e.g. TSQLRestServerStaticInMemory for TSQLVirtualTableJSON,  
 TSQLRestServerStaticExternal 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: TSQLRestServerStaticClass read fFeatures.StaticClass;

*The associated TSQLRestServerStatic class used for storage*

- e.g. returns TSQLRestServerStaticInMemory for TSQLVirtualTableJSON, or TSQLRestServerStaticExternal 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: TSQLRestServerStatic 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: TVarDataDynArray; out insertedRowID: Int64): boolean; **virtual**;

*Called to insert a virtual table row content*

- the column values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant UTF8Char), and varAny (BLOB with size = VLongs[0])
- 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/TSQVirtual/TSQVirtualTable\*
- when the class is instantiated, 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 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: TVarDataDynArray): boolean; **virtual**;

*Called to update a virtual table row content*

- the column values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- 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: TSQLRestServerStatic **read** fStatic;

*The associated virtual table storage*

- can be e.g. a TSQLRestServerStaticInMemory for TSQLVirtualTableJSON, or a TSQLRestServerStaticExternal for TSQLVirtualTableExternal, or nil for TSQLVirtualTableLog

**property** TableName: RawUTF8 **read** fTableName;

*The name of the Virtual Table, as specified following the TABLE keyword in the CREATE VIRTUAL TABLE statement*

**TSQLVirtualTableCursor = 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: TSQLVirtualTable); **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: TVarData): boolean; **virtual**; **abstract**;

*Called to retrieve a column value of the current data row*

- handled types in aResult are varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char) and varAny (BLOB with size = VLongs[0])
- 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: TSQLVirtualTablePrepared): boolean; **virtual**; **abstract**;

*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 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: TSQLVirtualTable **read** fTable;

*The associated Virtual Table class instance*

**TSQLVirtualTableCursorIndex = class(TSQLVirtualTableCursor)**

*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 TSQLRestServerStaticInMemory content*  
 - this is the cursor class associated to TSQLVirtualTableJSON

**function** Column(aColumn: integer; var aResult: TVarData): boolean; **override;**

*Called to retrieve a column value of the current data row*

- handled types in aResult are varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char) and varAny (BLOB with size = VLongs[0])
- 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 TSQLRestServerStaticInMemory-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'

**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: TVarDataDynArray; out insertedRowID: Int64): boolean; **override;**

*Called to insert a virtual table row content*

- the column values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- 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
- 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: TVarDataDynArray): boolean; **override**;

*Called to update a virtual table row content*

- the column values are available via some TVarData of type varNull, varInt64, varDouble, varString (mapping a constant PUTF8Char), and varAny (BLOB with size = VLongs[0])
- column order follows the Structure method, i.e. StoredClassProps.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 TSQLRestServerStaticInMemory-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 implemented by TSQLVirtualTableJSON
- 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: TVarData): boolean; override;
```

*Called to retrieve a column value of the current data row*

```
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:

```
Model.VirtualTableRegister(TSQLRecordDali1,TSQLVirtualTableJSON);
```

```
TSQLRecordVirtualTableAutoID = class(TSQLRecordVirtual)
```

*Record associated to Virtual Table implemented in Delphi, with ID generated automatically at INSERT*

- will use TSQLVirtualTableModule / TSQLVirtualTable / TSQLVirtualTableCursor classes for a generic Virtual table mechanism

- 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:

```
Model.VirtualTableRegister(TSQLRecordDali1,TSQLVirtualTableJSON);
```

```
TServiceRunningContext = record
```

*Will identify the currently running service on the server side*

- is the type of the global ServiceContext threadvar

```
Factory: TServiceFactoryServer;
```

*The currently running service factory*

- it can be used within server-side implementation to retrieve the associated TSQLRestServer instance

**RunningThread:** TThread;

*The thread which launched the request*

- is set by TSQLRestServer.BeginCurrentThread from multi-thread server handlers - e.g. TSQLite3HttpServer or TSQLRestServerNamedPipeResponse

**Session:** ^TSQLRestServerSessionContext;

*The currently running session identifier which launched the method*

- make available the current session or authentication parameters (including e.g. user details via Factory.RestServer.SessionGetUser)

**TTestLowLevelTypes = class(TSynTestCase)**

*This test case will test most low-level functions, classes and types defined and implemented in the SQLite3Commons unit*

*Used for DI-2.2.2 (page 833).*

**procedure** EncodeDecodeJSON;

*Some low-level JSON encoding/decoding*

**procedure** RTTI;

*Some low-level RTTI access*

- especially the field type retrieval from published properties

**procedure** UrlEncoding;

*Some low-level Url encoding from parameters*

**TTestBasicClasses = class(TSynTestCase)**

*This test case will test some generic classes defined and implemented in the SQLite3Commons unit*

*Used for DI-2.2.2 (page 833).*

**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*

**TSQLRecordPeople = class(TSQLRecord)**

*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 SQLite3.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): 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)

**TSQLLog = class**(TSynLog)

*Logging class with enhanced RTTI*

- will write TObject/TSQLRecord, enumerations and sets content as JSON
- is the default logging family used by SQLite3Commons/SQLite3

### Types implemented in the SQLite3Commons unit:

**PClassProp = ^TClassProp;**

*Pointer to TClassProp*

**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

**TClasses = array of TClass;**

*Abstract array of classes, will be filled e.g. with TSQLRecord descendants*

**TCreateTime = type** TTimeLog;

*An Int64-encoded date and time of the record creation*

- can be used as published property field in TSQLRecord for sftCreateTime
- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds - same as Iso8601ToSeconds()
- type cast any value of TModTime/TCreateTime/TTimeLog with the Iso8601 object below for easy access to its content

**TFindWhereEqualEvent = procedure**(aDest: pointer; aRec: TSQLRecord; aIndex: integer)  
**of object;**

*Event prototype called by FindWhereEqual() method*



```
TFloatType = ( ftSingle, ftDouble, ftExtended, ftComp, ftCurr );
```

*Specify floating point (ftFloat) storage size and precision*

```
TJSONSerializerCustomReader = function(const aValue: TObject; aFrom: PUTF8Char; var  
aValid: Boolean): PUTF8Char of object;
```

*Method prototype to be used for custom un-serialization of a class*

- to be used with TJSONSerializer.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 TJSONSerializerCustomWriter callback

```
TJSONSerializerCustomWriter = procedure(const aSerializer: TJSONSerializer; aValue:  
TObject; aHumanReadable, aDontStoreDefault, aFullExpand: Boolean) of object;
```

*Method prototype to be used for custom serialization of a class*

- to be used with TJSONSerializer.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 TJSONSerializerCustomReader callback

```
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
- use internally for computation an abstract "year" of 16 months of 32 days of 32 hours of 64 minutes of 64 seconds - same as Iso8601ToSeconds()
- type cast any value of TModTime/TCreateTime/TTimeLog with the Iso8601 object below for easy access to its content

```
TNotifySQLEvent = function(Sender: TSQLRestServer; Event: TSQLEvent; aTable:  
TSQLRecordClass; aID: integer): boolean of object;
```

*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)
- 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)

```
TONAuthenticationFailed = function(Retry: integer; var aUserName, aPassword:  
string): 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

**TOnRecordUpdate = procedure(Value: TSQLRecord) of object;**

*Used by TSQLRestClientURI.Update() to let the client perform the record update (refresh associated report e.g.)*

**TOnTableUpdate = procedure(aTable: TSQLTableJSON; State: TOnTableUpdateState) of object;**

*Used by TSQLRestClientURI.UpdateFromServer() to let the client perform the rows update (for Marked[] e.g.)*

**TOnTableUpdateState = ( tusPrepare, tusChanged, tusNoChange );**

*Possible call parameters for TOnTableUpdate Event*

**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*

**TParamFlag =  
 ( pfVar, pfConst, pfArray, pfAddress, pfReference, pfOut, pfResult );**

*The available kind of method parameters*

**TParamFlags = set of TParamFlag;**

*A set of kind of method parameters*

**TRecordReference = type PtrUInt;**

*A reference to another record in any table in the database Model*  
 - stored as an 32 bits unsigned integer (i.e. a pointer=TObject)  
 - 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

**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 );**

*The possible Server-side instance implementation patterns for 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  
 - sicShared: 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

```
TServiceMethodArgumentDynArray = array of TServiceMethodArgument;
```

*Describe a service provider method arguments*

```
TServiceMethodDynArray = array of TServiceMethod;
```

*Describe a service provider methods*

```
TServiceMethodExecutionOption = ( execInMainThread );
```

*Possible service provider method execution options*

```
TServiceMethodExecutionOptions = set of TServiceMethodExecutionOption;
```

*Set of per-method execution options for a service provider*

```
TServiceMethodValueDirection = ( smdConst, smdVar, smdOut, smdResult );
```

*Handled kind of parameters direction for a service provider 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

```
TServiceMethodValueType =  
( smvNone, smvSelf, smvBoolean, smvEnum, smvSet, smvInteger, smvCardinal, smvInt64,  
smvDouble, smvDateTime, smvCurrency, smvRawUTF8, smvString, smvWideString,  
smvRecord, smvObject, smvDynArray );
```

*Handled kind of parameters for a 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, after registration via a TTextWriter.RegisterCustomJSONSerializer call

```
TServiceMethodValueTypes = set of TServiceMethodValueType;
```

*Set of parameters for a service provider method*

```
TServiceMethodValueVar =  
( smvvNone, smvvSelf, smvv64, smvvRawUTF8, smvvString, smvvWideString, smvvRecord,  
smvvObject, smvvDynArray );
```

*Handled kind of parameters internal variables*

- reference-counted variables will have their own storage
- all non referenced-counted variables are stored within some 64 bit content

```
TServiceRoutingMode = ( rmREST, rmJSON_RPC );
```

*The routing mode of the service remote request*

- by default, will use an URI-based layout (rmREST), in which the service 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...] (one benefit of using URI is that it will be more secured in our RESTful authentication scheme: each method and even client driven session will be signed properly)

- if rmJSON\_RPC is used, the URI will define the interface, then the method name will be inlined with parameters, e.g.

```
POST /root/Calculator
(...)
{"method": "Add", "params": [1,2]}
```

or, for a sicClientDriven mode service:

```
POST /root/ComplexNumber
(...)
{"method": "Add", "params": [20,30], "id": 1234}
```

```
TSQLAction =
( actNoAction, actMark, actUnmarkAll, actmarkAllEntries, actmarkOlderThanOneDay,
actmarkOlderThanOneWeek, actmarkOlderThanOneMonth, actmarkOlderThanSixMonths,
actmarkOlderThanOneYear, actmarkInverse );
```

*Standard actions for User Interface generation*

```
TSQLActions = set of TSQLAction;
```

*Set of standard actions for User Interface generation*

```
TSQLAllowRemoteExecute = set of ( reSQL, reService, reUrlEncodedSQL,
reUrlEncodedDelete);
```

*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)
- 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

```
TSQLCheckTableName = ( ctnNoCheck, ctnMustExist, ctnTrimExisting );
```

*The possible options for handling table names*

```
TSQLEvent = ( seAdd, seUpdate, seDelete );
```

*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, sftInteger, sftID,
sftRecord, sftBoolean, sftFloat, sftDateTime, sftTimeLog, sftCurrency, sftObject,
sftBlob, sftBlobDynArray, sftMany, sftModTime, sftCreateTime );
```

*The available types for any SQL field property, as managed with the database driver*

```
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*

```
TSQLListLayout = ( llLeft, llUp, llClient, llLeftUp );
```

*Defines the way the TDrawGrid is displayed by User Interface generation*

```
TSQLOccasion = ( soSelect, soInsert, soUpdate, soDelete );
```

*Used to defined the 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

```
TSQLQueryEvent = function(aTable: TSQLRecordClass; aID: integer; 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 typecast 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, qoEqualToWithCase, qoNotEqualToWithCase, 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

```
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; so use RetrieveBlob() methods for handling BLOB fields

**TSQLRecordClass = class of TSQLRecord;**

*Class-reference type (metaclass) of TSQLRecord*

**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')

**TSQLRecords = array of TSQLRecordClass;**

*A dynamic array used to store the TSQLRecord classes in a Database Model*

**TSQLRecordTreeCoords = array[0..RTREE\_MAX\_DIMENSION-1] of packed record min, max: double; end;**

*This kind of record array can be used for direct coordinates storage*

**TSQLRecordVirtualKind =**

**( rSQLite3, rFTS3, rFTS4, rRTree, rCustomForcedID, rCustomAutoID );**

*The kind of SQLite3 (virtual) table*

- TSQLRecordFTS3 will be associated with vFTS3, TSQLRecordFTS4 with vFTS4, TSQLRecordRTree with vRTree, any native SQLite3 table as vSQLite3, and a TSQLRecordVirtualTable\*ID with 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 SQLite3DB unit

**TSQLRestCacheEntryValueDynArray = array of TSQLRestCacheEntryValue;**

*For TSQLRestCache, stores all tables values*

**TSQLRestServerCallBack = function(var aParams: TSQLRestServerCallBackParams): Integer of object;**

*Method prototype which must be used to implement the Server-Side*

*ModelRoot/[TableName/ID/]MethodName RESTful GET/PUT request of the Framework*

- 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/ID/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/ID/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 specifically be retrieved on purpose
- for ModelRoot/TableName/ID/MethodName, the just created instance will be freed by TSQLRestServer.URI when the method returns
- aParams.Parameters values are set from incoming URI, and each parameter can be retrieved with a loop like this:

```
if not UriDecodeNeedParameters(aParams.Parameters, 'SORT,COUNT') then
  exit;
while aParams.Parameters<>nil do begin
  UriDecodeValue(aParams.Parameters, 'SORT=', aSortString);
  UriDecodeValueInteger(aParams.Parameters, 'COUNT=', aCountInteger, @aParams.Parameters);
end;
```

- aParams.SentData is set with incoming data from the GET/PUT method
- aParams.Context 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() protected method to retrieve the user details
- aParams.Context.Method will indicate the used HTTP verb (e.g. GET/POST/PUT..)
- implementation must return the HTTP error code (e.g. 200/HTML\_SUCCESS) as an integer value, and any response in aParams.Resp as a JSON object by default (using e.g. TSQLRestServer.JSONEncodeResult), since default mime-type is JSON\_CONTENT\_TYPE:

```
{"result": "OneValue"}
```

or a JSON object containing an array:

```
{"result": ["One", "two"]}
```

- implementation can return an optional HTTP header (useful to set the response mime-type - see e.g. the TEXT\_CONTENT\_TYPE\_HEADER constant) in aParams.Head^
- implementation can return an optional error text in aParams.ErrorMsg^ in order to specify the HTTP error code with plain text (which will be sent as JSON error object into the client)
- a typical implementation may be:

```
function TSQLRestServerTest.Sum(var aParams: TSQLRestServerCallbackParams): Integer;
var a,b: Extended;
begin
  if not UriDecodeNeedParameters(aParams.Parameters, 'A,B') then begin
    result := HTML_NOTFOUND; // invalid Request
    aParams.ErrorMsg^ := 'Missing Parameter';
    exit;
  end;
  while Params.Parameters<>nil do begin
    UriDecodeExtended(aParams.Parameters, 'A=', a);
    UriDecodeExtended(aParams.Parameters, 'B=', b, @aParams.Parameters);
  end;
  aParams.Resp := JSONEncodeResult([a+b]);
  // same as : aParams.Resp := JSONEncode(['result',a+b]);
```



```
    result := HTML_SUCCESS;
  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/MethodName/[ID]) 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, and CallbackPut; see also TSQLModel.getURICallBack and JSONDecode function

```
function TSQLRecordPeople.Sum(aClient: TSQLRestClientURI; a, b: double): double;
var err: integer;
begin
  val(aClient.CallbackGetResult('sum',['a',a,'b',b]),result,err);
end;
```

```
TSQLURIMethod =
( mNone, mGET, mPOST, mPUT, mDELETE, mBEGIN, mEND, mABORT, mLOCK, mUNLOCK, mSTATE
);
```

*The available THTTP methods transmitted between client and server*

```
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*

```
TSQLVirtualTableTransaction =
( vttBegin, vttSync, vttCommit, vttRollBack, vttSavePoint, vttRelease,
vttRollBackTo );
```

*The available transaction levels*

```
TSynFilterOrValidateClass = class of TSynFilterOrValidate;
```

*Class-reference type (metaclass) for a TSynFilter or a TSynValidate*

```
TTypeKind =
( tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat, tkString, tkSet, tkClass,
tkMethod, tkWChar, tkLString, tkWString, tkVariant, tkArray, tkRecord, tkInterface,
tkInt64, tkDynArray );
```

*Available type families for Delphi 6 up to XE values*

```
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 *SQLite3Commons* 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*

`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

`COPIABLE_FIELDS: TSQLFieldTypes = [low(TSQLFieldType)..high(TSQLFieldType)] - [sftUnknown, sftMany];`

*Kind of fields which can be copied from one TSQLRecord instance to another*

`FULL_ACCESS_RIGHTS: TSQLAccessRights = (AllowRemoteExecute: [reSQL, reService, 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*

- this constant will set AllowRemoteExecute field to true
- is used by default only TSQLRestClientDB.URI() method, for direct local access right

`HTML_BADREQUEST = 400;`

*HTML Status Code for "Bad Request"*

`HTML_CREATED = 201;`

*HTML Status Code for "Created"*

`HTML_FORBIDDEN = 403;`

*HTML Status Code for "Forbidden"*

`HTML_NOTFOUND = 404;`

*HTML Status Code for "Not Found"*

`HTML_NOTIMPLEMENTED = 501;`

*HTML Status Code for "Not Implemented"*

`HTML_SUCCESS = 200;`

*HTML Status Code for "Success"*

`HTML_TIMEOUT = 408;`

*HTML Status Code for "Request Time-out"*

`HTML_UNAVAILABLE = 503;`

*HTML Status Code for "Service Unavailable"*

`INSERT_WITH_ID = [rFTS3, rFTS4, rRTree, 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];

*If the TSQLRecordVirtual table kind is a FTS3/FTS4 virtual table*

MAX\_SQLLOCKS = 512;

*Maximum number of the locked record in a Table (used in TSQLLocks)*

- code is somewhat faster and easier with a fixed cache size  
- 512 seems big enough on practice

MAX\_SQLTABLES = 256;

*There is no RTTI generated for them: so it won't work :( see [http://docwiki.embarcadero.com/RADStudio/en/Classes\\_and\\_Objects#Published\\_Members](http://docwiki.embarcadero.com/RADStudio/en/Classes_and_Objects#Published_Members)*

- should be defined globally, e.g. in Synapse.inc 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)

NOT\_SIMPLE\_FIELDS: TSQLFieldTypes = [sftUnknown, sftBlob, sftMany];

*Kind of fields not retrieved during normal query, update or adding*

RTREE\_MAX\_DIMENSION = 5;

*Maximum handled dimension for TSQLRecordRTree*

- this value is the one used by SQLite3 R-Tree virtual table

SUPERVISOR\_ACCESS\_RIGHTS: TSQLAccessRights = (AllowRemoteExecute:  
[reService, 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*

TEXT\_FIELDS: TSQLFieldTypes = [sftAnsiText, sftUTF8Text, sftDateTime, sftObject];

*Kind of 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 *SQLite3Commons* unit:

| Functions or procedures           | Description                                                                                  | Page |
|-----------------------------------|----------------------------------------------------------------------------------------------|------|
| BlobToBytes                       | Create a TBytes from TEXT-encoded blob data                                                  | 722  |
| BlobToStream                      | Create a memory stream from TEXT-encoded blob data                                           | 722  |
| BlobToTSQLRawBlob                 | Fill a TSQLRawBlob from TEXT-encoded blob data                                               | 722  |
| ClassFieldIndex                   | Retrieve a Field property index from a Property Name                                         | 722  |
| ClassFieldProp                    | Retrieve a Field property RTTI information from a Property Name                              | 722  |
| ClassFieldPropWithParents         | Retrieve a Field property RTTI information from a Property Name                              | 723  |
| ClassFieldPropWithParentsFromUTF8 | Retrieve a Field property RTTI information from a Property Name                              | 723  |
| CopyObject                        | Copy object properties                                                                       | 723  |
| ExtractInlineParameters           | This function will extract inlined :(1234): parameters into Types[]/Values[]                 | 723  |
| GetEnumCaption                    | Retrieve the ready to be displayed text of an enumeration                                    | 723  |
| GetEnumNameTrimmed                | Get the corresponding enumeration name, without the first lowercase chars (otDone -> 'Done') | 723  |
| GetJSONObjectAsSQL                | Decode JSON fields object into an UTF-8 encoded SQL-ready statement                          | 724  |
| GetJSONObjectAsSQL                | Decode JSON fields object into an UTF-8 encoded SQL-ready statement                          | 724  |
| GetObjectComponent                | Retrieve an object's component from its property name and class                              | 724  |
| GetRowCountNotExpanded            | Get the number of rows stored in the not-expanded JSON content                               | 724  |
| InternalClassProp                 | Retrieve the class property RTTI information for a specific class                            | 724  |
| InternalMethodInfo                | Retrieve a method RTTI information for a specific class                                      | 724  |
| isBlobHex                         | Return true if the TEXT is encoded as SQLite3 BLOB literals (X'53514C697465' e.g.)           | 724  |
| IsNotAjaxJSON                     | Returns TRUE if the JSON content is in expanded format                                       | 724  |
| JSONGetObject                     | Retrieve a JSON '{"Name":Value,...}' object                                                  | 725  |
| JSONIgnoreFieldName               | Go to the end of a field name in a JSON '"FieldName":Value' pair                             | 725  |
| JSONIgnoreFieldValue              | Go to the end of a value in a JSON '"FieldName":Value,' pair                                 | 725  |
| JSONIgnoreObject                  | Go to the end of a JSON '{"Name":Value,...}' object                                          | 725  |
| JSONToObject                      | Read an object properties, as saved by ObjectToJSON function                                 | 725  |

| Functions or procedures | Description                                                                                                                                                                                | Page |
|-------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| ObjectFromInterface     | Low-level function to retrieve the class instance implementing a given interface                                                                                                           | 725  |
| ObjectToJSON            | Will serialize any TObject into its UTF-8 JSON representation                                                                                                                              | 726  |
| ReadObject              | Read an object properties, as saved by TINIWriter.WriteObject() method                                                                                                                     | 726  |
| ReadObject              | Read an object properties, as saved by TINIWriter.WriteObject() method                                                                                                                     | 726  |
| RecordClassesToClasses  | Wrapper to convert an array of TSQLRecordClass to a generic array of TClass                                                                                                                | 726  |
| RecordReference         | Create a TRecordReference with the corresponding parameters                                                                                                                                | 726  |
| RecordRefToID           | Convert a dynamic array of TRecordRef into its corresponding IDs                                                                                                                           | 726  |
| SetDefaultValuesObject  | Set any default integer or enumerates (including boolean) published properties values for an object                                                                                        | 727  |
| SetWeak                 | Assign a Weak interface reference, to be used for circular references                                                                                                                      | 727  |
| SetWeakZero             | {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 | 727  |
| SQLFromSelectWhere      | Compute the SQL statement to be executed for a specific SELECT on Tables                                                                                                                   | 727  |
| SQLGetOrder             | Get the order table name from a SQL statement                                                                                                                                              | 727  |
| SQLParamContent         | Guess the content type of an UTF-8 SQL value, in :(...): format                                                                                                                            | 728  |
| TSQLRawBlobToBlob       | Creates a TEXT-encoded version of blob data from a memory data                                                                                                                             | 728  |
| TSQLRawBlobToBlob       | Creates a TEXT-encoded version of blob data from a TSQLRawBlob                                                                                                                             | 728  |
| UnJSONFirstField        | Get the FIRST field value of the FIRST row, from a JSON content                                                                                                                            | 728  |
| URIRequest              | This function can be exported from a DLL to remotely access to a TSQLRestServer                                                                                                            | 728  |
| UrlDecodeObject         | Decode a specified parameter compatible with URI encoding into its original object contents                                                                                                | 729  |
| UrlEncode               | Encode supplied parameters to be compatible with URI encoding                                                                                                                              | 729  |
| UTF8CompareBoolean      | Special comparaison function for sorting sftBoolean UTF-8 encoded values in the SQLite3 database or JSON content                                                                           | 729  |
| UTF8CompareCurr64       | Special comparaison function for sorting sftCurrency UTF-8 encoded values in the SQLite3 database or JSON content                                                                          | 729  |

| Functions or procedures | Description                                                                                                                                                 | Page |
|-------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| UTF8CompareDouble       | Special comparaison function for sorting sftFloat UTF-8 encoded values in the SQLite3 database or JSON content                                              | 729  |
| UTF8CompareInt64        | Special comparaison function for sorting sftInteger or sftTimeLog / sftModTime / sftCreateTime UTF-8 encoded values in the SQLite3 database or JSON content | 729  |
| UTF8CompareISO8601      | Special comparaison function for sorting sftDateTime UTF-8 encoded values in the SQLite3 database or JSON content                                           | 729  |
| UTF8CompareRecord       | Special comparaison function for sorting ftRecord (TRecordReference/RecordRef) UTF-8 encoded values in the SQLite3 database or JSON content                 | 729  |
| UTF8CompareUInt32       | Special comparaison function for sorting sftEnumerate, sftSet or sftID UTF-8 encoded values in the SQLite3 database or JSON content                         | 729  |
| UTF8ContentType         | Guess the content type of an UTF-8 encoded field value, as used in TSQLTable.Get()                                                                          | 730  |
| WriteObject             | Write an object properties, as saved by TINIWriter.WriteObject() method                                                                                     | 730  |
| WriteObject             | Write an object properties, as saved by TINIWriter.WriteObject() method                                                                                     | 730  |

**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 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 or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT

- the caller must free the stream instance after use

**function** BlobToTSQlRawBlob(P: PUTF8Char): TSQlRawBlob;

*Fill a TSQlRawBlob from TEXT-encoded blob data*

- blob data can be encoded as SQLite3 BLOB literals (X'53514C697465' e.g.) or or Base-64 encoded content ('\uFFF0base64encodedbinary') or plain TEXT

**function** ClassFieldIndex(ClassType: TClass; const PropName: shortstring): integer;

*Retrieve a Field property index from a Property Name*

**function** ClassFieldProp(ClassType: TClass; const PropName: shortstring): PPropInfo;

*Retrieve a Field property RTTI information from a Property Name*

**function** ClassFieldPropWithParents(aClassType: TClass; **const** PropName: shortstring): PPropInfo;

*Retrieve a Field property RTTI information from a Property Name*  
 - this special version also search into parent properties (default is only current)

**function** ClassFieldPropWithParentsFromUTF8(aClassType: TClass; PropName: UTF8Char): PPropInfo;

*Retrieve a Field property RTTI information from a Property Name*  
 - this special version also search into parent properties (default is only current)

**procedure** CopyObject(aFrom, aTo: TObject);

*Copy object properties*  
 - copy only Integer, Int64, enumerates (including boolean), object properties and string properties (excluding WideString, but including UnicodeString)  
 - 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** ExtractInlineParameters(**const** SQL: RawUTF8; **var** Types: TSQLFieldTypeArray; **var** Values: TRawUTF8DynArray; **var** maxParam: integer): RawUTF8;

*This function will extract inlined :(1234): parameters into Types[]/Values[]*  
 - will return the generic SQL statement with ? instead of :(1234):  
 - call internaly SQLParamContent() function for inline parameters decoding  
 - will set maxParam=0 in case of no inlined parameters  
 - recognized types are only sftInteger, sftFloat, sftDateTime ('\uFFF1...'), sftUTF8Text and sftBlob ('\uFFFF0...')  
 - sftUnknown is returned on invalid content

**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** GetEnumNameTrimmed(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 GetJSONObjectAsSQL(var P: PUTF8Char; const Fields: TRawUTF8DynArray;  
Update, InlinedParams: boolean; RowID: Integer=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 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 830).*

```
function GetJSONObjectAsSQL(JSON: RawUTF8; Update, InlinedParams: boolean; RowID:  
Integer=0; ReplaceRowIDWithID: Boolean=false): RawUTF8; overload;
```

*Decode JSON fields object into an UTF-8 encoded SQL-ready statement*

- expect JSON expanded object as "COL1"="VAL1",...} pairs
- make its own temporary copy of JSON data before calling GetJSONObjectAsSQL() above
- returns 'COL1="VAL1", COL2=VAL2' if UPDATE is true (UPDATE SET format)
- returns '(COL1, COL2) VALUES ("VAL1", VAL2)' otherwise (INSERT format)
- if InlinedParams is set, will create prepared parameters like 'COL2=: (VAL2):'
- if RowID is set, a RowID column will be added within the returned content

*Used for DI-2.1.2 (page 830).*

```
function GetObjectComponent(Obj: TPersistent; const ComponentName: shortstring;  
ComponentClass: TClass): pointer;
```

*Retrieve an object's component from its property name and class*

- usefull to set User Interface component, e.g.

```
function GetRowCountNotExpanded(P: PUTF8Char; FieldCount: integer; var RowCount:  
integer): PUTF8Char;
```

*Get the number of rows stored in the not-expanded JSON content*

```
function InternalClassProp(ClassType: TClass): PClassProp;
```

*Retrieve the class property RTTI information for a specific class*

```
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 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** JSONGetObject(**var** P: PUTF8Char; ExtractID: PInteger; **var** EndOfObject: AnsiChar): 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 (will expect this "ID" property to be the FIRST in the "Name":Value pairs)

**function** JSONIgnoreFieldName(P: PUTF8Char): PUTF8Char;

*Go to the end of a field name in a JSON '"FieldName":Value' pair*

- returns nil if P was not formatted as expected
- returns the position of Value

**function** JSONIgnoreFieldValue(P: PUTF8Char): PUTF8Char;

*Go to the end of a value in a JSON '"FieldName":Value,' pair*

- returns nil if P was not formatted as expected
- returns the position of the expected ending ',' or '}' delimiter

**function** JSONIgnoreObject(P: PUTF8Char): PUTF8Char;

*Go to the end of a JSON '{"Name":Value,...}' object*

- returns nil if P was not formatted as expected
- returns the position after the expected ending '}' delimiter

**function** JSONToObject(**var** ObjectInstance; From: PUTF8Char; **var** Valid: boolean; TObjectListItemClass: TClass=nil): PUTF8Char;

*Read an object properties, as saved by ObjectToJSON function*

- ObjectInstance must be an existing TObject instance
- the data inside From^ is modified (unescaped and transformed): don't call JSONToObject(pointer(JSONRawUTF8)) but makes a temporary copy of the JSONRawUTF8 text before calling this function
- handle Integer, Int64, enumerate (including boolean), set, floating point, TDateTime, TCollection, TStrings, TRawUTF8List, and string properties (excluding ShortString and WideString, but including UnicodeString under Delphi 2009+)
- won't handle TList/TObjectList (even if ObjectToJSON is able to serialize them) since has now way of knowing the object type to add (TCollection.Add is missing), unless you set the TObjectListItemClass property as expected, and provide a TObjectList object (TList won't be handled since it may leak memory when calling TList.Clear)
- will release 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 TJJSONSerializer.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** 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 our TInterfacedObjectFake kind of interface implementation classes

```
function ObjectToJSON(Value: TObject; HumanReadable: boolean=false;
DontStoreDefault: boolean=true): 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 and enumerate (e.g. boolean) properties of the object
- won't handle variant, shortstring and widestring properties
- 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 the TJSONSerializer.RegisterCustomSerializer() class method
- call internally TJSONSerializer.WriteObject

```
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 values and string properties (excluding WideString, but including UnicodeString)
- 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

```
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 string properties (excluding WideString, but including UnicodeString)
- 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

```
function RecordClassesToClasses(const Tables: array of TSQLRecordClass):
TClasses;
```

*Wrapper to convert an array of TSQLRecordClass to a generic array of TClass*

- used e.g. by TSQLRestClientURI.List before calling TSQLTableJSON.Create

```
function RecordReference(Model: TSQLModel; aTable: TSQLRecordClass; aID:
integer): TRecordReference;
```

*Create a TRecordReference with the corresponding parameters*

```
procedure RecordRefToID(var aArray: TIntegerDynArray);
```

*Convert a dynamic array of TRecordRef into its corresponding IDs*

**procedure** SetDefaultValuesObject(Value: TObject);

*Set any default integer or enumerates (including boolean) published properties values for an object*  
- reset only the published properties of the current class level (do NOT reset the properties content published in the parent classes)

**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;{\$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*

- 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 FreeInstance virtual method in order to reset any reference to nil: FreeInstance 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 FreeInstance 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
- implementation should be thread-safe

**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** 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** SQLParamContent(P: PUTF8Char; out ParamType: TSQLFieldType; out ParamValue: RawUTF8): 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 ':(\uFFFF0base64encodedbinary):' pattern (or ':(null):'), and return raw binary (for direct blob parameter assignment)
- sftDateTime will be recognized from ':(\uFFFF1"2012-05-04"):' pattern, i.e. JSON\_SQLDATE\_MAGIC-prefixed string as returned by DateToSQL() or DateTimeToSQL() functions
- sftUnknown is returned on invalid content
- if ParamValue is not nil, the pointing RawUTF8 string is set with the value inside :(...): without double quoting in case of sftUTF8Text

**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** UnJSONFirstField(var P: PUTF8Char): RawUTF8;

*Get the FIRST field value of the FIRST row, from a JSON content*

- e.g. usefull to get an ID without converting a JSON content into a TSQLTableJSON

*Used for DI-2.1.2 (page 830).*

**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 HTML\_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 829).*

**function** UrlDecodeObject(U, Upper: PUTF8Char; var ObjectInstance; Next: PPUTF8Char=nil): boolean;

*Decode a specified parameter compatible with URI encoding into its original object contents*

- ObjectInstance must be an existing TObject instance
- will call internaly JSOToOject() 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** UrlEncode(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 := UrlEncodeFull(['select','\*','where','ID=12','offset',23,'object',aObject]);
- parameters can be either textual, integer or extended, or any TObject (standard UrlEncode()) will only handle
- TObject serialization into UTF-8 will be processed by the ObjectToJSON() function

**function** UTF8CompareBoolean(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftBoolean UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareCurr64(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftCurrency UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareDouble(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftFloat UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareInt64(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftInteger or sftTimeLog / sftModTime / sftCreateTime UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareISO8601(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftDateTime UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareRecord(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting ftRecord (TRecordReference/RecordRef) UTF-8 encoded values in the SQLite3 database or JSON content*

**function** UTF8CompareUInt32(P1,P2: PUTF8Char): PtrInt;

*Special comparaison function for sorting sftEnumerate, sftSet or sftID UTF-8 encoded values in the SQLite3 database or JSON content*



**function** UTF8ContentType(P: PUTF8Char): TSQLFieldType;

*Guess the content type of an UTF-8 encoded field value, as used in TSQLTable.Get()*

- if P is 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 '\uFFFF0' 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, sftRecord, sftBoolean or sftModTime / sftCreateTime / sftTimeLog type

**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 string properties (excluding WideString, but including UnicodeString)
- 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 string properties (excluding WideString, but including UnicodeString)
- 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

#### Variables implemented in the *SQLite3Commons* unit:

**ServiceContext:** TServiceRunningContext;

*This thread-specific variable will be set with the currently running service context (on the server side)*

- is set by TServiceFactoryServer.ExecuteMethod() just before calling the implementation method of a service, allowing to retrieve the current execution context
- 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

**SQLite3Log:** TSynLogClass = TSQLLog;

*TSQLLog class is used for logging for all our ORM related functions*

- this global variable can be used to customize it

**USEFASTMM4ALLOC:** boolean = false;

*If this variable is TRUE, the URIRequest() function won't use Win32 API GlobalAlloc() function, but fastest native Getmem()*

- can be also usefull for debugg

*Used for DI-2.1.1.2.1 (page 829).*



#### 1.4.7.20. SQLite3HttpClient unit

*Purpose:* HTTP/1.1 RESTFUL JSON mORMot Client classes

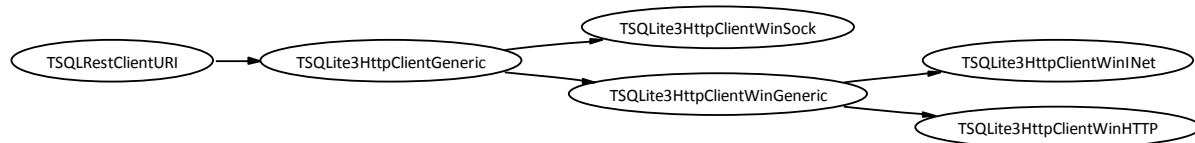
- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

The *SQLite3HttpClient* unit is quoted in the following items:

| SWRS #       | Description       | Page |
|--------------|-------------------|------|
| DI-2.1.1.2.4 | HTTP/1.1 protocol | 830  |

Units used in the *SQLite3HttpClient* unit:

| Unit Name             | Description                                                                                                                                                                         | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                               | 575  |
| <i>SynCommons</i>     | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17           | 229  |
| <i>SynCrtSock</i>     | Classes implementing HTTP/1.1 client and server protocol<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 369  |
| <i>SynLZ</i>          | SynLZ Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                             | 445  |



*SQLite3HttpClient class hierarchy*

Objects implemented in the *SQLite3HttpClient* unit:

| Objects                      | Description                                                                                  | Page |
|------------------------------|----------------------------------------------------------------------------------------------|------|
| TSQLite3HttpClientGeneric    | Generic HTTP/1.1 RESTFUL JSON SQLite3 Client class                                           | 732  |
| TSQLite3HttpClientWinGeneric | HTTP/1.1 RESTFUL JSON SQLite3 Client abstract class using either WinINet either TWinHTTP API | 732  |
| TSQLite3HttpClientWinHTTP    | HTTP/1.1 RESTFUL JSON SQLite3 Client class using WinHTTP API                                 | 733  |
| TSQLite3HttpClientWinINet    | HTTP/1.1 RESTFUL JSON SQLite3 Client class using WinINet API                                 | 733  |
| TSQLite3HttpClientWinSock    | HTTP/1.1 RESTFUL JSON SQLite3 Client class using SynCrtSock / WinSock                        | 732  |

**TSQlite3HttpClientGeneric = class(TSQLRestClientURI)**

*Generic HTTP/1.1 RESTFUL JSON SQLite3 Client class*

**KeepAliveMS: cardinal;**

*The time (in milliseconds) to keep the connection alive with the TSQlite3HttpServer*  
 - default is 20000, i.e. 20 seconds

**TSQlite3HttpClientWinSock = class(TSQlite3HttpClientGeneric)**

*HTTP/1.1 RESTFUL JSON SQLite3 Client class using SynCrtSock / WinSock*

- 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

**constructor** Create(**const** aServer, aPort: AnsiString; aModel: TSQLModel);  
**reintroduce;**

*Connect to TSQlite3HttpServer on aServer:aPort*

**destructor** Destroy; **override;**

*Release all memory, internal SQLite3 client and HTTP handlers*

**property** Socket: THttpClientSocket **read** fSocket;

*Internal HTTP/1.1 compatible client*

**TSQlite3HttpClientWinGeneric = class(TSQlite3HttpClientGeneric)**

*HTTP/1.1 RESTFUL JSON SQLite3 Client abstract class using either WinINet either TWinHTTP API*

- not to be called directly, but via TSQlite3HttpClientWinINet or (even better)  
 TSQlite3HttpClientWinHTTP overridden classes

**constructor** Create(**const** aServer, aPort: AnsiString; aModel: TSQLModel; aHttps: boolean=false; **const** aProxyName: AnsiString=''; **const** aProxyByPass: AnsiString=''); **reintroduce;**

*Connect to TSQlite3HttpServer 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

**destructor** Destroy; **override;**

*Release all memory, internal SQLite3 client and HTTP handlers*

**property** WinAPI: TWinHttpAPI **read** fWinAPI;

*Internal class instance used for the connection*

- will return either a TWinINet, either a TWinHTTP class instance

```
TSQLite3HttpClientWinINet = class(TSQLite3HttpClientWinGeneric)
```

*HTTP/1.1 RESTFUL JSON SQLite3 Client class using WinINet API*

- this class is 15/20 times slower than TSQLite3HttpClient 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

```
TSQLite3HttpClientWinHTTP = class(TSQLite3HttpClientWinGeneric)
```

*HTTP/1.1 RESTFUL JSON SQLite3 Client class using 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 used in your client programs: TSQLite3HttpClient 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 *SQLite3HttpClient* unit:

```
TSQLite3HttpClient = TSQLite3HttpClientWinHTTP;
```

*HTTP/1.1 RESTFUL JSON SQLite3 default Client class*

- under Windows, map the TSQLite3HttpClientWinHTTP class

*Used for DI-2.1.1.2.4 (page 830).*

#### 1.4.7.21. *SQLite3HttpServer* unit

*Purpose:* HTTP/1.1 RESTFUL JSON mORMot Server classes

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

The *SQLite3HttpServer* unit is quoted in the following items:

| SWRS #       | Description                                                                                                       | Page |
|--------------|-------------------------------------------------------------------------------------------------------------------|------|
| DI-2.1.1.2.4 | HTTP/1.1 protocol                                                                                                 | 830  |
| DI-2.2.2     | The framework libraries, including all its <i>SQLite3</i> related features, shall be tested using Unitary testing | 833  |

Units used in the *SQLite3HttpServer* unit:

| Unit Name             | Description                                                                                                                                                                         | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                               | 575  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17           | 229  |
| <i>SynCrtSock</i>     | Classes implementing HTTP/1.1 client and server protocol<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 369  |
| <i>SynLZ</i>          | SynLZ Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                             | 445  |



*SQLite3HttpServer class hierarchy*

#### Objects implemented in the *SQLite3HttpServer* unit:

| Objects            | Description                                | Page |
|--------------------|--------------------------------------------|------|
| TSQLite3HttpServer | HTTP/1.1 RESTFUL JSON SQLite3 Server class | 734  |

**TSQLite3HttpServer = class(TObject)**

*HTTP/1.1 RESTFUL JSON SQLite3 Server class*

- this server is multi-threaded and not blocking
- 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 a pure Delphi THttpServer instance; a solution is to call the THttpApiServer.AddUrlAuthorize class method during program setup for the desired port, in order to allow it for every user
- just create it and it will serve SQL statements as UTF-8 JSON
- for a true AJAX server, expanded data is preferred - your code may contain:  
`DBServer.NoAJAXJSON := false;`

*Used for DI-2.1.1.2.4 (page 830).*

```
constructor Create(const aPort: AnsiString; aServer: TSQLRestServer; const  
aDomainName: AnsiString='+'; DontUseHttpApiServer: Boolean=false;  
aRestAccessRights: PSQLAccessRights=nil; ServerThreadPoolCount: Integer=32);  
reintroduce; overload;
```

*Create a Server Thread, binded and listening on a TCP port to HTTP JSON 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
- aDomainName is the URLprefix to be used for HttpAddUrl API call

*Used for DI-2.1.1.2.4 (page 830).*

```
constructor Create(const aPort: AnsiString; const aServers: array of  
TSQLRestServer; const aDomainName: AnsiString='+'; DontUseHttpApiServer:  
Boolean=false; ServerThreadPoolCount: Integer=32); reintroduce; overload;
```

*Create a Server Thread, binded and listening on a TCP port to HTTP JSON 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
- 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 TSQLite3HttpApiServer instance
- if DontUseHttpApiServer is set, kernel-mode HTTP.SYS server won't be used and standard Delphi code will be called instead (not recommended)
- by default, the PSQLAccessRights 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)

*Used for DI-2.1.1.2.4 (page 830).*

```
destructor Destroy; override;
```

*Release all memory, internal SQLite3 server (if any) and HTTP handlers*

```
function AddServer(aServer: TSQLRestServer; aRestAccessRights:  
PSQLAccessRights=nil): boolean;
```

*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
- return true on success, false on error (e.g. duplicated Root value)

*Used for DI-2.1.1.2.4 (page 830).*

**property** DBServer[Index: integer]: TSQLRestServer read GetDBServer;

*Read-only access to all internal servers*

*Used for DI-2.1.1.2.4 (page 830).*

**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** HttpServer: THttpServerGeneric read fHttpServer;

*The associated running HTTP server instance*

- either THttpApiServer, either THttpServer

**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

#### Constants implemented in the *SQLite3HttpServer* unit:

HTTP\_DEFAULT\_ACCESS\_RIGHTS: PSQLAccessRights = @SUPERVISOR\_ACCESS\_RIGHTS;

*The default access rights used by the HTTP server if none is specified*

#### 1.4.7.22. *SQLite3i18n* unit

*Purpose:* Internationalization (i18n) routines and classes

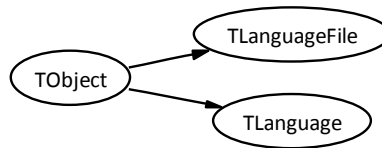
- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

#### The *SQLite3i18n* unit is quoted in the following items:

| SWRS #     | Description    | Page |
|------------|----------------|------|
| DI-2.3.1.3 | Automated i18n | 834  |

#### Units used in the *SQLite3i18n* unit:

| Unit Name             | Description                                                                                                                                                               | Page |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                     | 575  |
| <i>SynCommons</i>     | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SQLite3i18n class hierarchy*

#### Objects implemented in the *SQLite3i18n* unit:

| Objects       | Description                                                                | Page |
|---------------|----------------------------------------------------------------------------|------|
| TLanguage     | A common record to identify a language                                     | 737  |
| TLanguageFile | Class to load and handle translation files (fr.msg, de.msg, ja.msg.. e.g.) | 738  |

#### **TLanguage = object(TObject)**

*A common record to identify a Language*

*Used for DI-2.3.1.3 (page 834).*

**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 a 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 834).*

## **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 834).*

**constructor** Create(const aFileName: TFileName; aLanguageLocale: TLanguages); overload;

*Specify a text file containing the translation messages for a Language*

*Used for DI-2.3.1.3 (page 834).*

**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 834).*

**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 834).*

**function** DateTimeToText(const ISO: Iso8601): 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 834).*

**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 834).*

**function** DateToText(const ISO: Iso8601): string; overload;

*Convert a date into a ready to be displayed text on the screen*

*Used for DI-2.3.1.3 (page 834).*

**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 834).*

**function** PropToString(Prop: PPropInfo; 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 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 834).*

**function** TimeToText(const ISO: Iso8601): string; overload;

*Convert a time into a ready to be displayed text on the screen*

*Used for DI-2.3.1.3 (page 834).*

**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 834).*

**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

**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 834).*

#### Types implemented in the *SQLite3i18n* 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,
lngAramaic, lngCroatian, lngHungarian, lngArmenian, lngIndonesian, lngInterlingue,
lngIcelandic, lngItalian, lngJapanese, lngKorean, lngTibetan, lngLithuanian,
lngMalgash, 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, lngPashtol );
```

*Languages handled by this SQLite3i18n 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 SQLite3i18n unit:

```
LanguageAbr: packed array[TLanguages] of RawByteString =
('he','gr','la','ad','bs','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','sr','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]i18n\programname*

#### Functions or procedures implemented in the SQLite3i18n unit:

| Functions or procedures | Description                                                                                                                                                                                                            | Page |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| AnyTextFileToString     | 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 | 742  |
| DateTimeToIso           | Generic US/English date/time to VCL text conversion                                                                                                                                                                    | 743  |
| GetText                 | Translate the 'Text' term into current language, with no    nor \$\$[\$[\$]]                                                                                                                                           | 743  |
| i18nAddLanguageCombo    | Add combo-box items, for all available languages on disk                                                                                                                                                               | 743  |
| i18nAddLanguageItems    | Add strings items, for all available languages on disk                                                                                                                                                                 | 743  |
| i18nAddLanguageMenu     | Add sub-menu items to the Menu, for all available languages on disk                                                                                                                                                    | 743  |
| i18nLanguageToRegistry  | Save the default language to the registry                                                                                                                                                                              | 743  |
| i18nRegistryToLanguage  | Get the default language from the registry                                                                                                                                                                             | 743  |
| Iso2S                   | Convert a custom date/time into a VCL-ready string                                                                                                                                                                     | 743  |
| LanguageAbrToIndex      | LanguageAbrToIndex('GR')=1, e.g.                                                                                                                                                                                       | 744  |
| LanguageAbrToIndex      | LanguageAbrToIndex('GR')=1, e.g.                                                                                                                                                                                       | 744  |
| LanguageName            | Return the language text, ready to be displayed (after translation if necessary)                                                                                                                                       | 744  |
| LanguageToLCID          | Convert a i18n language index into a Windows LCID                                                                                                                                                                      | 744  |
| LCIDToLanguage          | Convert a Windows LCID into a i18n language                                                                                                                                                                            | 744  |
| LoadResString           | Our hooked procedure for reading a string resource                                                                                                                                                                     | 744  |
| S2U                     | Convert any generic VCL Text into an UTF-8 encoded String                                                                                                                                                              | 744  |
| U2S                     | Convert an UTF-8 encoded text into a VCL-ready string                                                                                                                                                                  | 744  |
| _                       | Translate the 'English' term into current language                                                                                                                                                                     | 744  |

**function** AnyTextFileToString(const FileName: TFileName): 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*

- by use of this function, the TLanguageFile.LoadFromFile() method is able to display any Unicode message into the 8 bit standard Delphi VCL, (for Delphi 2 to 2007) or with the new Unicode VCL (for Delphi 2009 and up)
- before Delphi 2009, the current string code page is used (i.e. CurrentAnsiConvert)

**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/DateToText/TimeToText

**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 PtrInt(Objects[]) will be the corresponding language ID
- return the current language index in List.Items[]

**procedure** i18nAddLanguageMenu(const MsgPath: TFileName; Menu: TMenuItem);

*Add sub-menu items to the Menu, for all available languages on disk*

- uses internally i18nAddLanguageItems() function above
- current language is checked
- all created MenuItem.OnClick event will launch Language.LanguageClick to change the current language in the registry

**function** i18nLanguageToRegistry(const Language: TLanguages): string;

*Save the default Language to the registry*

- language will be changed at next startup
- return a message ready to be displayed on the screen
- return "string" type, i.e. UnicodeString for Delphi 2009 and up

**function** i18nRegistryToLanguage: TLanguages;

*Get the default Language from the registry*

**function** Iso2S(Iso: TTimeLog): string;

*Convert a custom date/time into a VCL-ready string*

- this function must be assigned to i18nDateText global var of SQLite3Commons unit
- wrapper to Language.DateTimeToText method

**function** LanguageAbrToIndex(const value: RawUTF8): TLanguages; overload;

*LanguageAbrToIndex('GR')=1, e.g.*

- return LANGUAGE\_NONE if not found

**function** LanguageAbrToIndex(p: pAnsiChar): TLanguages; overload;

*LanguageAbrToIndex('GR')=1, e.g.*  
 - return LANGUAGE\_NONE if not found

**function** LanguageName(aLanguage: TLanguages): string;

*Return the Language text, ready to be displayed (after translation if necessary)*  
 - e.g. LanguageName(IngEnglish)='English'  
 - return "string" type, i.e. UnicodeString for Delphi 2009 and up

**function** LanguageToLCID(Language: TLanguages): integer;

*Convert a i18n Language index into a Windows LCID*

**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 SQLite3Commons 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 834).*

**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 834).*

**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 834).*

#### Variables implemented in the *SQLite3i18n* 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

#### 1.4.7.23. SQLite3Pages unit

*Purpose:* Reporting unit

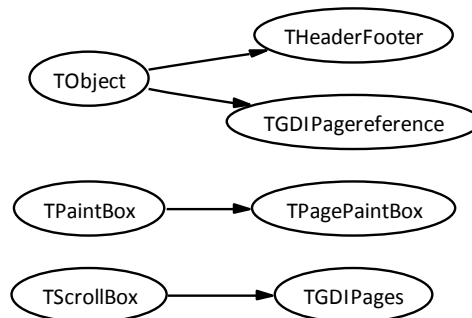
- this unit is a part of the freeware Synapse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**The SQLite3Pages unit is quoted in the following items:**

| SWRS #   | Description                                                                                | Page |
|----------|--------------------------------------------------------------------------------------------|------|
| DI-2.3.2 | A reporting feature, with full preview and export as PDF or TXT files, shall be integrated | 835  |

**Units used in the SQLite3Pages unit:**

| Unit Name         | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                      | 229  |
| <i>SynGdiPlus</i> | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynPdf</i>     | PDF file generation<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                        | 459  |



*SQLite3Pages class hierarchy*

#### Objects implemented in the *SQLite3Pages* unit:

| Objects           | Description                                                             | Page |
|-------------------|-------------------------------------------------------------------------|------|
| TColRec           | Internal format of a text column                                        | 747  |
| TGDIPagereference | Internal structure used to store bookmarks or links                     | 747  |
| TGDIPages         | Report class for generating documents from code                         | 747  |
| THeaderFooter     | Internal format of the header or footer text                            | 747  |
| TPagePaintBox     | Hack the TPaintBox to allow custom background erase                     | 747  |
| TSavedState       | A report layout state, as used by SaveLayout/RestoreSavedLayout methods | 746  |

**TSavedState = record**

*A report layout state, as used by SaveLayout/RestoreSavedLayout methods*

**THederFooter = 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*

**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 835).*

**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 TGDIPPlusFull.ForceUseDrawString 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

**ForcePrintAsBitmap:** 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 usefull, since slows the printing a lot and makes huge memory usage

**GroupsMustBeOnSamePage:** 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 "Pages" 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)

**OnPopupMenuClick:** 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

**OnPopupMenuPopup:** 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

**OnStringToUnicode:** TOnStringToUnicodeEvent;

*Customize left aligned text conversion from Ansi*

- to be used before Delphi 2009/2010/XE only, in order to force some character customization (e.g. <= or >=)

**PopupMenuClass:** TPopupMenuClass;

*User can customize this class to create an advanced popup menu instance*

**PreviewSurfaceBitmap:** TBitmap;

*The bitmap used to draw the page*

**constructor** Create(AOwner: TComponent); **override;**

*Creates the reporting component*

**destructor** Destroy; **override;**

*Finalize the component, releasing all used memory*

**function** 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

**function** 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

**function** CurrentGroupPosStart: integer;

*Distance (in mm's) from the top of the page to the top of the current group*

- returns CurrentYPos if no group is in use

**function** ExportPDF(aPdfFileName: TFileName; ShowErrorOnScreen: boolean;  
LaunchAfter: boolean=true): boolean;

*Export the current report as PDF*

- 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** NewPopupMenuItem(const aCaption: string; Tag: integer=0; SubMenu: TMenuItem=nil; OnClick: TNotifyEvent=nil; ImageIndex: integer=-1): TMenuItem;

*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 forming 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 forming*

- 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 forming*

- 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); **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);

*Will add the current 'Page n/n' text at the specified position*

- PageText must be of format 'Page %d/%d', in the desired language

**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.*

**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

**procedure** AppendRichEdit(RichEditHandle: HWnd);

*Append a Rich Edit content to the current report*



**procedure** BeginDoc;

*Begin a Report document*

- Every report must start with BeginDoc and end with EndDoc

**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(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** DrawBMP(rec: TRect; bmp: TBitmap); **overload**;

*Stretch draws a bitmap image at the specified page coordinates in mm's*

**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** DrawDashedLine;

*Draw a Dashed Line between the left & right margins*

**procedure** DrawGraphic(graph: TGraphic; bLeft, bWidth: integer; **const** Legend: SynUnicode='');  
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);

*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)

**procedure** DrawTextAcrossCols(**const** StringArray: array of SynUnicode;  
BackgroundColor: TColor=clNone);

*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);

*Draw one line of text, with the current alignment*

**procedure** DrawTextFmt(**const** s: string; **const** Args: array of **const**);

*Draw some text as a paragraph, with the current alignment*

- this method use format() like parameters

**procedure** DrawTextU(**const** s: RawUTF8);

*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)

**procedure** DrawTextW(**const** s: SynUnicode);

*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)

**procedure** DrawTitle(**const** s: SynUnicode; DrawBottomLine: boolean=false;  
OutlineLevel: Integer=0; **const** aBookmark: string=''; **const** aLink: string='');

*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** 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*

- usefull after habing used AddColumns([]) method e.g.

**procedure** SetColumnBold(index: integer);

*Individually set column bold state*

- usefull after habing 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;

*Show a form with the preview, allowing the user to browse pages and print the report*

**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** 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** 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 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** 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** 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*  
- usefull for Hebrew, Arabic and some Asiatic languages handling  
- set to FALSE by default, for faster content generation  
- the Synopse 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

**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](http://en.wikipedia.org/wiki/Income_statement))
- align numbers on digits, not parentheses

**property** OnDocumentProduced: TNotifyEvent **read** fOnDocumentProducedEvent **write** fOnDocumentProducedEvent;

*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*

**property** OnNewPage: TNewPageEvent **read** fStartNewPage **write** fStartNewPage;

*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*

- nb: it's not possible to change the orientation once a report has started, i.e. after a BeginDoc call

**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: TStringList **read** fPages;

*The TMetaFile list containing all pages content*  
- the TMetaFile is stored in the Objects[] array  
- the text equivalent of the page is stored in the Strings[] array  
- 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*

**property** PrinterName: string **read** fCurrentPrinter;

*The name of the current selected printer*

**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*

**property** ZoomStatus: TZoomStatus **read** fZoomStatus;

*The current Zoom procedure, i.e. zsPercent, zsPageFit or zsPageWidth*



### Types implemented in the *SQLite3Pages* unit:

**TColAlign** = ( caLeft, caRight, caCenter, caCurrency );  
*Text column alignment*

**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 equivalent)

**TReportPopupMenu** =  
( rNone, rNextPage, rPreviousPage, rGotoPage, rZoom, rBookmarks, rPageAsText,  
rPrint, rExportPDF, rClose );  
*The available menu items*

**TTextAlign** = ( 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 *SQLite3Pages* unit:

**FORMAT\_ALIGN\_MASK** = \$300;  
*Alignment bits 8-9*

**FORMAT\_BOLD** = \$400;  
*Fontstyle bits 10-12*

**FORMAT\_DEFAULT** = \$0;  
*TEXT FORMAT FLAGS...*

**FORMAT\_SINGLELINE** = \$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_WIDTH = -1;
```

*Preview page zoom options...*

#### 1.4.7.24. SQLite3SelfTests 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 1.17

#### Units used in the *SQLite3SelfTests* unit:

| Unit Name          | Description                                                                                                                                                                                                                                                                                                             | Page |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i>  | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                               | 229  |
| <i>SynCrtSock</i>  | Classes implementing HTTP/1.1 client and server protocol<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                     | 369  |
| <i>SynCrypto</i>   | Fast cryptographic routines (hashing and cypher)<br>- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms<br>- optimized for speed (tuned assembler and VIA PADLOCK optional support)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 383  |
| <i>SynDB</i>       | Abstract database direct access classes<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                             | 395  |
| <i>SynDBODBC</i>   | ODBC 3.x library direct access classes to be used with our SynDB architecture<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                | 428  |
| <i>SynDBOracle</i> | Oracle DB direct access classes (via OCI)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                           | 432  |

| Unit Name           | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynGdiPlus</i>   | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynLZ</i>        | SynLZ Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                                                                        | 445  |
| <i>SynLZO</i>       | Fast LZO Compression routines<br>- licensed under a MPL/GPL/LGPL tri-license; version 1.13                                                                                                                                                                                                                                                     | 447  |
| <i>SynOleDB</i>     | Fast OleDB direct access classes<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                           | 447  |
| <i>SynPdf</i>       | PDF file generation<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                        | 459  |
| <i>SynSelfTests</i> | Automated tests for common units of the Synopse Framework<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                  | 500  |
| <i>SynZip</i>       | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                | 560  |

#### Functions or procedures implemented in the *SQLite3SelfTests* unit:

| Functions or procedures    | Description | Page |
|----------------------------|-------------|------|
| <i>SQLite3ConsoleTests</i> |             | 761  |

#### **procedure** *SQLite3ConsoleTests*;

*Disable Range checking in our code disable Stack checking in our code expect extended syntax  
disable stack frame generation disable overflow checking in our code expect short circuit boolean  
disable Var-String Checking Typed @ operator enumerators stored as byte by default Open string  
params define HASINLINE USETYPEINFO CPU32 CPU64 this is the main entry point of the tests*  
- this procedure will create a console, then run all available tests

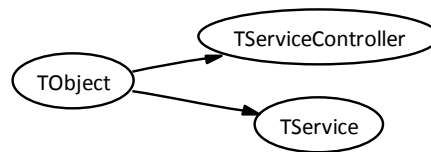
#### **1.4.7.25. SQLite3Service unit**

*Purpose:* Win NT Service managment classes

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

#### Units used in the *SQLite3Service* unit:

| Unit Name             | Description                                                                                                                                                               | Page |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                     | 575  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SQLite3Service class hierarchy*

#### Objects implemented in the *SQLite3Service* unit:

| Objects            | Description                                                                                                                                      | Page |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|------|
| TService           | TService is the class used to implement a service provided by an application                                                                     | 765  |
| TServiceController | TServiceController class is intended to create a new service instance or to maintain (that is start, stop, pause, resume...) an existing service | 762  |

**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 Dependances:  
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. Expected 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.
- DisplayName - display name of a service.
- Path - a path to binary (executable) of the service created.
- OrderGroup - an order group name (unnecessary)
- Dependances - 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 are no dependances).
- 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*

**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*

**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;

*Retrive 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: TServiceCtrlHandler **read** GetCtrlHandler **write** SetCtrlHandler;

*Callback handler for Windows Service Controller*

- if handler is not set, then auto generated handler calls DoCtrlHandle

**property** Data: DWORD **read** FData **write** FData;

*Any data You wish to associate with the service object*

**property** DisplayName: String **read** fDName **write** fDName;

*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: TServiceControlEvents **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 (better)

### Types implemented in the *SQLite3Service* unit:

**TServiceControlEvents** = **procedure**(Sender: TService; Code: DWORD) **of object**;  
*Event triggered for Control handler*

**TServiceCtrlHandler** = **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*

### Constants implemented in the *SQLite3Service* unit:

**CM\_SERVICE\_CONTROL\_CODE** = WM\_USER+1000;  
*Translated caption needed only if with full UI*

### Functions or procedures implemented in the *SQLite3Service* unit:

| Functions or procedures    | Description                                                                           | Page |
|----------------------------|---------------------------------------------------------------------------------------|------|
| CurrentStateToServiceState | Convert the Control Code retrieved from Windows into a service state enumeration item | 767  |
| ServicesRun                | Launch the registered Services execution                                              | 767  |
| ServiceStateText           | Return the ready to be displayed text of a TServiceState value                        | 767  |

**function** CurrentStateToServiceState(CurrentState: DWORD): TServiceState;  
*Convert the Control Code retrieved from Windows into a service state enumeration item*

**procedure** ServicesRun;  
*Launch the registered Services execution*  
- the registered list of service provided by the application is sent to the operating system

**function** ServiceStateText(State: TServiceState): **string**;  
*Return the ready to be displayed text of a TServiceState value*

### Variables implemented in the *SQLite3Service* unit:

**Services**: TList = **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

#### 1.4.7.26. SQLite3ToolBar unit

*Purpose:* Database-driven Office 2007 Toolbar

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

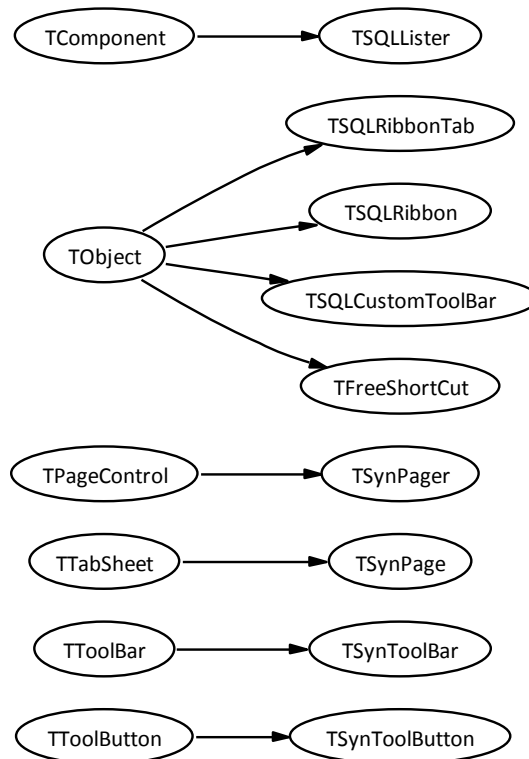
**The *SQLite3ToolBar* unit is quoted in the following items:**

| SWRS #     | Description             | Page |
|------------|-------------------------|------|
| DI-2.3.1.1 | Database Grid Display   | 833  |
| DI-2.3.1.2 | RTTI generated Toolbars | 834  |

**Units used in the *SQLite3ToolBar* unit:**

| Unit Name             | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                          | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                    | 736  |
| <i>SQLite3Pages</i>   | Reporting unit<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                             | 745  |
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                          | 781  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                    | 793  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                      | 229  |
| <i>SynGdiPlus</i>     | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynTaskDialog</i>  | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                 | 553  |

| Unit Name | Description                                                                                                                                                                     | Page |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| SynZip    | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 560  |



*SQLite3ToolBar class hierarchy*

#### Objects implemented in the *SQLite3ToolBar* unit:

| Objects           | Description                                                                                                | Page |
|-------------------|------------------------------------------------------------------------------------------------------------|------|
| TFreeShortCut     | A simple object to get one char shortcuts from caption value                                               | 770  |
| TSQLCustomToolBar | Create one or more toolbars in a ribbon page, according to an enumeration of actions                       | 774  |
| TSQLLister        | A hidden component, used for handling toolbar buttons of actions to be performed on a TSQLRecordClass list | 771  |
| TSQLRibbon        | Store some variables common to all pages, i.e. for the whole ribbon                                        | 776  |
| TSQLRibbonTab     | Store the UI elements and data, one per each Table                                                         | 774  |
| TSynPage          | A Ribbon page, which will contain some toolbars for a TSQLRecord class                                     | 770  |
| TSynPager         | The ribbon pager, which will contain one page per TSQLRecord class                                         | 771  |
| TSynToolBar       | A toolbar on a Ribbon page                                                                                 | 770  |
| TSynToolBarButton | A button on the Ribbon toolbars, corresponding to one action                                               | 770  |

**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

**TSynToolButton = class(TToolButton)**

*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 with 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** OnDbClick;  
*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 833), DI-2.3.1.2 (page 834).*

```
constructor Create(aOwner: TComponent; aClient: TSQLRestClientURI; aClass:
TSQLRecordClass; aGrid: TDrawGrid; aIDColumnHide: boolean; aPager: TSynPager;
aImageList32,aImageList16: TImageList; aOnClick: TSQListerEvent;
aOnValueText: TValueTextEvent; const aGridSelect: RawUTF8= '*';
aHideDisabledButtons: 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 '')
- aOnClick 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 833).*

```
constructor Create(aOwner: TComponent; aClient: TSQLRestClientURI; aClass:
TSQLRecordClass; aGrid: TDrawGrid; aIDColumnHide: boolean; aPager: TSynPager;
aImageList32,aImageList16: TImageList; aOnClick: TSQListerEvent;
aOnValueText: TValueTextEvent; aTable: TSQLTable; aHideDisabledButtons:
boolean); reintroduce; overload;
```

*Same as above, but with a specified TSQLTable*

*Used for DI-2.3.1.1 (page 833).*

```
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

```
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)

```
function FindButton(ActionIndex: integer): TSynToolButton;
```

*Find associate Button for an action*

```
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 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 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 834).*

**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** CreateSubMenuItem(aButtonIndex: **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 834).*

**TSQLRibbonTab = class(TObject)**

*Store the UI elements and data, one per each Table*

*Used for DI-2.3.1.2 (page 834).*

**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** ActionsTBCaptionCSV, ActionsHintCaption: **string**;  
 ActionIsNotButton: pointer; aOnActionClick: TSQLListerEvent; ViewToolBarIndex:  
 integer; aHideDisabledButtons: 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);

*Triggerred when a report popup menu item is clicked*

**property** CurrentID: integer **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: TBooleanDynArray **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 834).*

**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='');
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 834).*

```
destructor Destroy; override;
```

*Release associated memory*

```
function AddToReport(aReport: TGDIPages; aRecord: TSQLRecord; WithTitle:
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 occurred

```
function ExportRecord(aTable: TSQLRecordClass; aID: integer; 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 occurred
- 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 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: integer; 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: integer; 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 *SQLite3ToolBar* unit:

TFreeShortCutSet = **set of** ord('A')..ord('Z');

*Used to mark which shortcut keys have already been affected*

TOnCaptionName = **function**(Action: PShortString; 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; **const** RecordClass: TSQLRecordClass; ActionValue: integer) **of object**;



*This event is called when a button is pressed*

**TSQLRibbonSetActionEvent = function**(TabIndex, ToolbarIndex: integer; TestEnabled: boolean; var A): **string of object**;

*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 ""

**TSynBodyPage = TSynPage;**

*Body page used to display the list and the report on the client screen*

**TSynBodyPager = TSynPager;**

*Body pager used to display the list and the report on the client screen*

**TSynForm = TVistaForm;**

*A Vista-enabled TForm descendant*

- this form will have a button in the TaskBar
- this form will hide the default Delphi application virtual form

**TSynPopupMenu = TPopupMenu;**

*A popup menu to be displayed*

#### Functions or procedures implemented in the *SQLite3ToolBar* unit:

| Functions or procedures      | Description                                                                    | Page |
|------------------------------|--------------------------------------------------------------------------------|------|
| AddIconToImageList           | Add an Icon to the supplied TImageList                                         | 780  |
| CaptionName                  | Retrieve the ready to be displayed text of the given property                  | 781  |
| CreateReportWithIcons        | Create a report containing all icons for a given action enumeration            | 781  |
| ImageListStretch             | Fill a TImageList from the content of another TImageList                       | 781  |
| LoadBitmapFromResource       | Load a bitmap from a .png/.jpg file embedded as a resource to the executable   | 781  |
| LoadImageListFromBitmap      | Load TImageList bitmaps from a TBitmap                                         | 781  |
| LoadImageListFromEmbeddedZip | Load TImageList bitmaps from an .zip archive embedded as a ZIP resource        | 781  |
| NewDrawCellBackground        | Draw the cell of a TDrawGrid according to the current Theming of TabAppearance | 781  |

**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; Action: PShortString; 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): TBitmap;
```

*Load a bitmap from a .png/.jpg file embedded as a resource to the executable*

```
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*

#### 1.4.7.27. SQLite3UI unit

*Purpose:* Grid to display Database content

- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

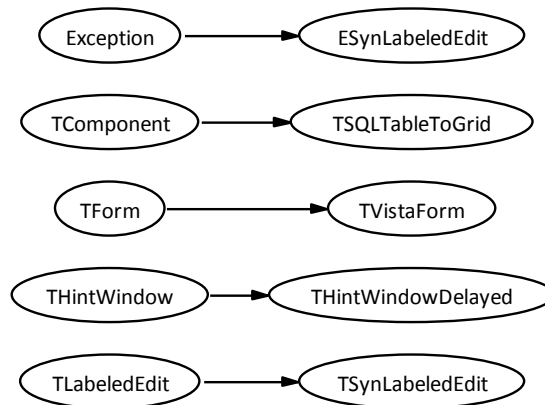
**The SQLite3UI unit is quoted in the following items:**

| SWRS #     | Description           | Page |
|------------|-----------------------|------|
| DI-2.3.1.1 | Database Grid Display | 833  |

**Units used in the SQLite3UI unit:**

| Unit Name      | Description                                                                                                                                                                 | Page |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| SQLite3Commons | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                       | 575  |
| SQLite3i18n    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 736  |

| Unit Name         | Description                                                                                                                                                                  | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*SQLite3UI class hierarchy*

#### Objects implemented in the *SQLite3UI* unit:

| Objects            | Description                                                        | Page |
|--------------------|--------------------------------------------------------------------|------|
| ESynLabeledEdit    | Exception class raised by TSynIntegerLabeledEdit                   | 787  |
| THintWindowDelayed | A THintWindow descendant, with an internal delay to auto-hide      | 782  |
| TSQLTableToGrid    | A hidden component, used for displaying a TSQLTable in a TDrawGrid | 783  |
| TSynLabeledEdit    | TLabeledEdit with optional boundaries check of a Variant value     | 787  |
| TVistaForm         | Vista-enabled TForm descendant                                     | 788  |

**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;**

*Overriden 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 833).*

```
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 833).*

**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): 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

**function** SelectedID: integer;

*Get the ID of the first selected row, 0 on error (no ID field e.g.)*

- usefull even if ID column was hidden with IDCColumnHide

**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;
```

- usefull even if ID column was hidden with IDCColumnHide

**procedure** AfterRefresh(aID: integer);

*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** DrawGridKeyDown(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** DrawGridKeyPress(Sender: TObject; var Key: Char);

*Called by the owner TDrawGrid when the user presses a key*

- used for incremental key lookup

**procedure** DrawGridMouseDown(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** DrawGridMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);

*Called by the owner TDrawGrid when the mouse is over a Cell*

**procedure** DrawGridMouseUp(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** SetCentered(aCol: cardinal); overload;

*Set a column number which must be centered*

**procedure** SetCentered(const Cols: array of cardinal); overload;

*Set columns number which must be centered*

**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=c1Black);

*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** Centered: Int64 read fCentered;

*Individual bits of this field is set to display a column data as centered*

**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** 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** 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** 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)

**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 SQLite3UI 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*

```
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 *SQLite3UI* unit:

| Functions or procedures  | Description                                                                                     | Page |
|--------------------------|-------------------------------------------------------------------------------------------------|------|
| AddApplicationToFirewall | Allow an application to access the network through the Windows firewall                         | 789  |
| AddPortToFirewall        | Open a firewall port on the current computer                                                    | 789  |
| ClearTypeEnable          | Enable the ClearType font display                                                               | 790  |
| CreateAnIcon             | Create an Icon                                                                                  | 790  |
| DrawCheckBox             | Draw a CheckBox in the Canvas Handle of the Wwindow hWnd, in the middle of the Rect coordinates | 790  |
| FillStringGrid           | Fill TStringGrid.Cells[] with the supplied data                                                 | 790  |
| GetShellFolderPath       | Get the corresponding windows folder, from its ID                                               | 790  |
| HideAppFormTaskBarButton | Low level VCL routine in order to hide the application from Windows task bar                    | 790  |
| IsClearTypeEnabled       | Test if the ClearType is enabled for font display                                               | 790  |
| Register                 | Register the TSynIntegerLabeledEdit component in the IDE toolbar                                | 790  |

```
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 Wwindow 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 SQLite3i18n 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()

**function** 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 SQLite3 framework to run: since all User Interface is created from code, and not from the Delphi IDE, you don't have to register anything

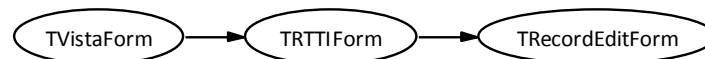
#### 1.4.7.28. SQLite3UIEdit unit

*Purpose:* Record edition dialog, used to edit record content on the screen

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**Units used in the SQLite3UIEdit unit:**

| Unit Name             | Description                                                                                                                                                                                    | Page |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                          | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                    | 736  |
| <i>SQLite3ToolBar</i> | Database-driven Office 2007 Toolbar<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                 | 768  |
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                          | 781  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                    | 793  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                      | 229  |
| <i>SynTaskDialog</i>  | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 553  |



*SQLite3UIEdit class hierarchy*

#### Objects implemented in the *SQLite3UIEdit* unit:

| Objects         | Description                                                      | Page |
|-----------------|------------------------------------------------------------------|------|
| TRecordEditForm | Record edition dialog, used to edit record content on the screen | 792  |
| TRTTIForm       | A common ancestor, used by both TRecordEditForm and TOptionsForm | 791  |

**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 *SQLite3UIEdit* unit:

TOnComponentCreate = **function**(Obj: TObject; Prop: PPropInfo; Parent: TWinControl): TWinControl **of object**;

*Event used for the window creation*

TOnComponentCreated = **procedure**(Obj: TObject; Prop: PPropInfo; Comp: TWinControl) **of object**;

*Event used to customize the input component after creation*

TOnComponentValidate = **function**(EditControl: TWinControl; Prop: PPropInfo): 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)

#### 1.4.7.29. *SQLite3UILogin* unit

*Purpose:* Some common User Interface functions and dialogs

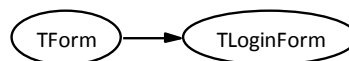
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

#### Units used in the *SQLite3UILogin* unit:

| Unit Name             | Description                                                                                                                                                               | Page |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                     | 575  |
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17     | 781  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



| Unit Name            | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynGdiPlus</i>    | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynTaskDialog</i> | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                 | 553  |



*SQLite3UILogin class hierarchy*

#### Objects implemented in the *SQLite3UILogin* unit:

| Objects    | Description                                  | Page |
|------------|----------------------------------------------|------|
| TLoginForm | Form used to Log User and enter its password | 794  |

```
TLoginForm = class(TForm)
```

*Form used to Log User and enter its password*

```
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*

#### Functions or procedures implemented in the *SQLite3UILogin* unit:

| Functions or procedures | Description                                               | Page |
|-------------------------|-----------------------------------------------------------|------|
| Choose                  | Ask the User to choose between some Commands              | 795  |
| Choose                  | Ask the User to choose between some Commands              | 795  |
| CreateTempForm          | Popup a temporary form with a message over all forms      | 795  |
| EnsureSingleInstance    | Ensure that the program is launched once                  | 795  |
| HtmlEscape              | Convert an error message into html compatible equivalency | 795  |
| InputBox                | Ask the User to enter some string value                   | 795  |

| Functions or procedures | Description                                                                   | Page |
|-------------------------|-------------------------------------------------------------------------------|------|
| InputQuery              | Ask the User to enter some string value                                       | 796  |
| InputSelect             | Ask the User to select one item from an array of strings                      | 796  |
| InputSelectEnum         | Ask the User to select one enumerate item                                     | 796  |
| SetStyle                | Set the style for a form and a its buttons                                    | 796  |
| ShowMessage             | Show an (error) message, using a Vista-Style dialog box                       | 796  |
| ShowMessage             | Show an (error) message, using a Vista-Style dialog box                       | 796  |
| YesNo                   | Ask the User to choose Yes or No [and Cancel], using a Vista-Style dialog box | 796  |

**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** 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** CreateTempForm(const aCaption: string; aPanelReference: PTPanel=nil; ScreenCursorHourGlass: boolean=false; aCaptionColor: integer=c1Navy; aCaptionSize: integer=12): TForm;

*Popup a temporary form with a message over all forms*

**procedure** EnsureSingleInstance;

*Ensure that the program is launched once*

- the main project .dpr source file must contain:

**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 a its buttons*

- set the Default Font for all components, i.e. Calibri if available

```
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*

#### 1.4.7.30. SQLite3UIOptions unit

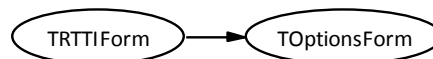
*Purpose:* General Options setting dialog

- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17

**Units used in the *SQLite3UIOptions* unit:**

| Unit Name             | Description                                                                                                                                                                 | Page |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                       | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 736  |
| <i>SQLite3ToolBar</i> | Database-driven Office 2007 Toolbar<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17              | 768  |

| Unit Name             | Description                                                                                                                                                                                    | Page |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                          | 781  |
| <i>SQLite3UIEdit</i>  | Record edition dialog, used to edit record content on the screen<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17    | 790  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                    | 793  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                      | 229  |
| <i>SynTaskDialog</i>  | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 553  |



*SQLite3UIOptions class hierarchy*

#### Objects implemented in the *SQLite3UIOptions* unit:

| Objects      | Description            | Page |
|--------------|------------------------|------|
| TOptionsForm | Options setting dialog | 797  |

**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: TScrollBar; 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

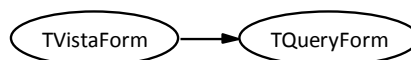
#### 1.4.7.31. SQLite3UIQuery unit

*Purpose:* Form handling queries to a User Interface Grid

- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.15

#### Units used in the *SQLite3UIQuery* unit:

| Unit Name             | Description                                                                                                                                                                                    | Page |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                          | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                    | 736  |
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                          | 781  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                    | 793  |
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                      | 229  |
| <i>SynTaskDialog</i>  | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 553  |



*SQLite3UIQuery class hierarchy*

## Objects implemented in the *SQLite3UIQuery* unit:

| Objects    | Description                                       | Page |
|------------|---------------------------------------------------|------|
| TQueryForm | This Form perform simple Visual queries to a Grid | 799  |

**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 bandwidth 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

## 1.5. Main SynFile Demo

### 1.5.1. 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-256/SHA-256* 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 45);
- Database design - see *Object-relational mapping* (page 46);
- User Interface generation.

We hope this part of the *Software Architecture Design (SAD)* document would be able to be a reliable guideline for using our framework for your own projects.

### 1.5.2. 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.

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, and the `ForceBlobTransfert := true` line in `TFileClient`. Create method.

### 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.

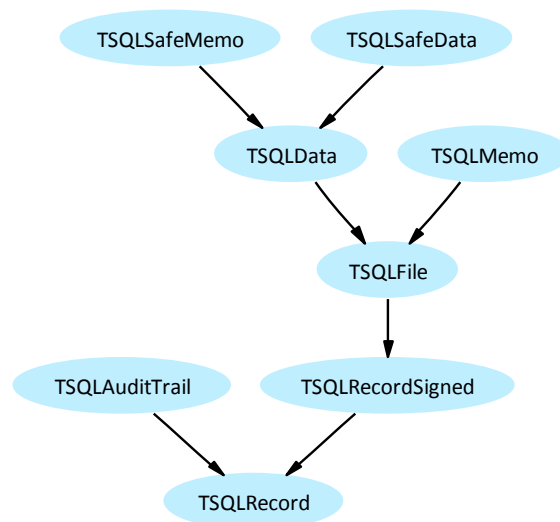
You'll discover how the ORM plays its role here: you change the data, just like changing any class instance properties.

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.

### 1.5.3. 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* (page 46). The following class hierarchy was designed:





*SynFile TSQLRecord classes hierarchy*

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 defined 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:

```

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:

```

TSQLMemo = class(TSQLFile)
  
```

```
public
  fContent: RawUTF8;
published
  property Content: RawUTF8 read fContent write fContent;
end;
```

|                         |
|-------------------------|
| ID : integer            |
| Content : RawUTF8       |
| Created : TTimeLog      |
| KeyWords : RawUTF8      |
| Modified : TTimeLog     |
| Name : RawUTF8          |
| Picture : TSQLRawBlob   |
| Signature : RawUTF8     |
| SignatureTime: TTimeLog |

*Memo Record Layout*

```
TSQLData = class(TSQLFile)
public
  fData: TSQLRawBlob;
published
  property Data: TSQLRawBlob read fData write fData;
end;
```

|                         |
|-------------------------|
| ID : integer            |
| Data : TSQLRawBlob      |
| Created : TTimeLog      |
| KeyWords : RawUTF8      |
| Modified : TTimeLog     |
| Name : RawUTF8          |
| Picture : TSQLRawBlob   |
| Signature : RawUTF8     |
| SignatureTime: TTimeLog |

*Data Record Layout*

```
TSQLSafeMemo = class(TSQLData);
```

|                         |
|-------------------------|
| ID : integer            |
| Data : TSQLRawBlob      |
| Created : TTimeLog      |
| KeyWords : RawUTF8      |
| Modified : TTimeLog     |
| Name : RawUTF8          |
| Picture : TSQLRawBlob   |
| Signature : RawUTF8     |
| SignatureTime: TTimeLog |

*SafeMemo Record Layout*

```
TSQLSafeData = class(TSQLData);
```

|                         |
|-------------------------|
| ID : integer            |
| Data : TSQLRawBlob      |
| Created : TTimeLog      |
| KeyWords : RawUTF8      |
| Modified : TTimeLog     |
| Name : RawUTF8          |
| Picture : TSQLRawBlob   |
| Signature : RawUTF8     |
| SignatureTime: TTimeLog |

*SafeData Record Layout*

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*:

```

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;

```

|                                     |
|-------------------------------------|
| ID : integer                        |
| AssociatedRecord : TRecordReference |
| Status : TFileEvent                 |
| StatusMessage : RawUTF8             |
| Time : TTimeLog                     |

*AuditTrail Record Layout*

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 will not 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(AssociatedRecord) 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 44) - the framework expects a common database model to be shared between client and server. A common function has been defined in the `FileTables.pas` unit, as such:

```
function CreateFileModel(Owner: TSQLRest): 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 the *SynProject*, which is able to create custom graphs from the application source code it parsed.

#### 1.5.4. 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 127) - 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 generate a Ribbon-based application, in which each table is available via a Ribbon tab, and some actions performed to it.

So the framework would 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 `SQLite3UIEdit.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 219).

##### 1.5.4.1. 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:

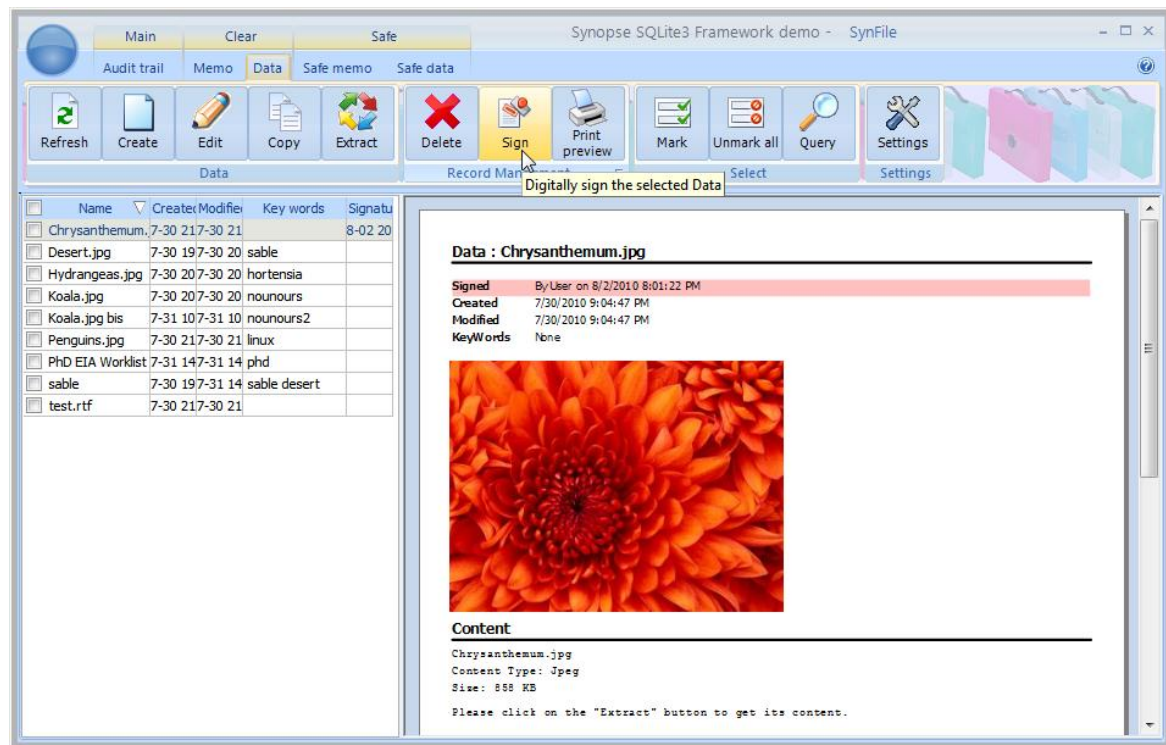
| Criteria                   | VCL             | TMS           |
|----------------------------|-----------------|---------------|
| Rendering                  | Basic           | Sophisticated |
| OS version                 | Variant         | Constant      |
| Ribbon look                | Unusual         | 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 `SQLite3ToolBar` (i.e. the `TSynForm`, `TSynToolBar`, `TSynToolButton`, `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 will require acceptance of the Office UI License terms as defined at <http://msdn.microsoft.com/en-us/office/aa973809.aspx..>

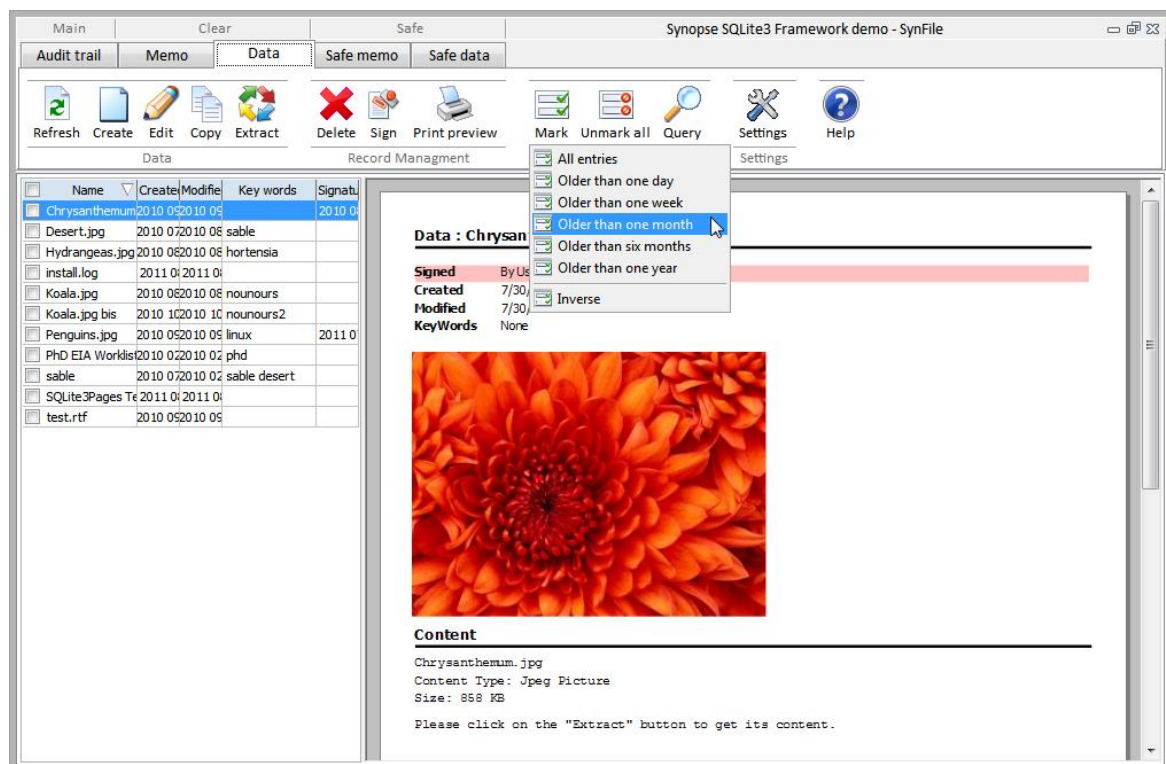
If you want to design your user interface using a Office 2007/2010 ribbon look, please take a look at those official guidelines: <http://msdn.microsoft.com/en-us/library/cc872782.aspx..>

Here is the screen content, using the TMS components:



*User Interface generated using TMS components*

And here is the same application compiled using only VCL components, available from Delphi 6 up to XE2:



*User Interface generated using VCL components*

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.

#### 1.5.4.2. Enumeration types

A list of available actions should be defined, as an enumeration type:

```
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 `SQLite3i18n.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 806) and *User Interface generated using VCL components* (page 806).

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`):

```
TFileEvent = (  
  feUnknownState, feServerStarted, feServerShutdown,  
  feRecordCreated, feRecordModified, feRecordDeleted,  
  feRecordDigitallySigned, feRecordImported, feRecordExported );
```

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.

#### 1.5.4.3. 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

```
TFileRibbonTabParameters = object(TSQLRibbonTabParameters)  
  /// the SynFile actions  
  Actions: TFileActions;  
end;
```

Then a constant array of such objects is defined:

```
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: TSQLData;  
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.

#### 1.5.4.4. Report generation

The following CreateReport method is overridden in FileClient.pas:

```
/// class used to create the User interface  
TFileRibbon = class(TSQLRibbon)  
public  
  /// overridden method used customize the report content  
  procedure CreateReport(aTable: TSQLRecordClass; aID: integer; aReport: TGDIPages;  
    AlreadyBegan: boolean=false); override;  
end;
```

The reporting engine in the framework is implemented via the TGDIPages class, defined in the SQLite3Pages.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:

```
procedure TFileRibbon.CreateReport(aTable: TSQLRecordClass; aID: integer; 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
begin
  inherited; // default handler
  exit;
end;
Rec := TSQLFile(P.CurrentRecord);
Caption := U2S(Rec.fName);
```

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.

```
// prepare page footer
SaveLayout;
Font.Size := 9;
AddPagesToFooterAt(sPageN, LeftMargin);
TextAlign := taRight;
AddTextToFooterAt('SynFile http://synopse.info - '+Caption, RightMarginPos);
RestoreSavedLayout;
```

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.

```
// write global header at the beginning of the report
DrawTitle(P.Table.CaptionName+ ' : '+Caption, 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
  s := sNone else
begin
  s := U2S(Rec.fKeywords);
  ExportPDFKeywords := s;
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;
```

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 SQLite3i18n.pas, are used for conversion of some text or TTimeLog into a text formatted with the current language settings (i18n).

```
// 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 http://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 SQLite3i18n.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.

#### 1.5.4.5. 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 SQLite3i18n.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 87).

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.

##### 1.5.4.5.1. Creating the reference file

In order to begin a translation task, the SQLite3i18n.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:

```
procedure TMainForm.FormShow(Sender: TObject);
begin
{$ifdef EXTRACTALLRESOURCES}
  ExtractAllResources(
    // first, all enumerations to be translated
    [TypeInfo(TFileEvent),TypeInfo(TFileAction),TypeInfo(TPreviewAction)],
    // then some class instances (including the TSQLModel will handle all TSQLRecord)
    [Client.Model],
    // some custom classes or captions
    [],[]);
  Close;
{$else}
  //i18nLanguageToRegistry(LngFrench);
{$endif}
  Ribbon.ToolBar.ActivePageIndex := 1;
end;
```

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 SQLite3 Framework demo - SynFile

[Messages]
2784453965=Memo
2751226180=Data
744738530=Safe memo
895337940=Safe data
2817614158=Name
1741937413=&User name:
4235002365=&Password:
16479868= Synopse SQLite3 Framework 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 NumericalKey=EnglishText pairs. Note this will reflect the exact content of resourcestring 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 refers 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.

#### 1.5.4.5.2. Adding a new language

In order to translate the whole application into French, the following SynFile.FR file could be made available in the SynFile.exe folder:

```
[Messages]  
2784453965=Texte  
2751226180=Données  
744738530=Texte sécurisé  
895337940=Données sécurisées  
2817614158=Nom  
1741937413=&Nom utilisateur:  
4235002365=&Mot de passe:  
16479868= Synopse mORMot Framework demo - SynFile  
940170664=Contenu  
3153227598=Vide  
3708724895=Page %d / %d  
2767358349=Taille: %s  
contenu: %s  
protégé par un mot de passe.|Choisissez "Editer" pour le visualiser.  
3011148197=Choisissez "Extraire" pour enregistrer le contenu.  
388288630=Signé,Par %s le %s  
(....)
```

Since no form-level custom captions have been defined in this SynFile.FR file, the default numerical values will be used. In our case, Name.EditLabel1.Caption will be displayed using the text specified by 2817614158, i.e. 'Nom'.

Note that the special characters %s %d , | markup was preserved: only the plain English text has been translated to the corresponding French.

#### 1.5.4.5.3. Language selection

User Interface language can be specified at execution.

By default, it will use the registry to set the language. It will need an application restart, but it will also allow easier translation of all forms, using a low-level hook of the TForm.Create constructor.

For instance, if you set in FileMain.pas, for the framework main demo:

```
procedure TMainForm.FormShow(Sender: TObject);  
begin  
  (...)  
  i18nLanguageToRegistry(lngFrench);
```

```
Ribbon.ToolBar.ActivePageIndex := 1;
end;
```

Above code will set the main application language as French. At next startup, the content of a supplied `SynFileFR.msg` file 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.

#### 1.5.4.5.4. 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 `SQLite3i18n` unit will register itself to some methods of `SQLite3Commons.pas`, in order to translate the RTTI-level text into the current selected language. See for instance `i18nDateText`.

### 1.5.5. Source code implementation

#### 1.5.5.1. Main SynFile Demo used Units

The Main `SynFile` Demo makes use of the following units.

**Units located in the "Lib\" directory:**

| Source File Name     | Description                                                         | Page |
|----------------------|---------------------------------------------------------------------|------|
| <i>SynCommons</i>    | Common functions used by most Synopse projects                      | 229  |
| <i>SynCrypto</i>     | Fast cryptographic routines (hashing and cypher)                    | 383  |
| <i>SynGdiPlus</i>    | GDI+ library API access                                             | 438  |
| <i>SynTaskDialog</i> | Implement TaskDialog window (native on Vista/Seven, emulated on XP) | 553  |
| <i>SynZip</i>        | Low-level access to ZLib compression (1.2.5 engine version)         | 560  |



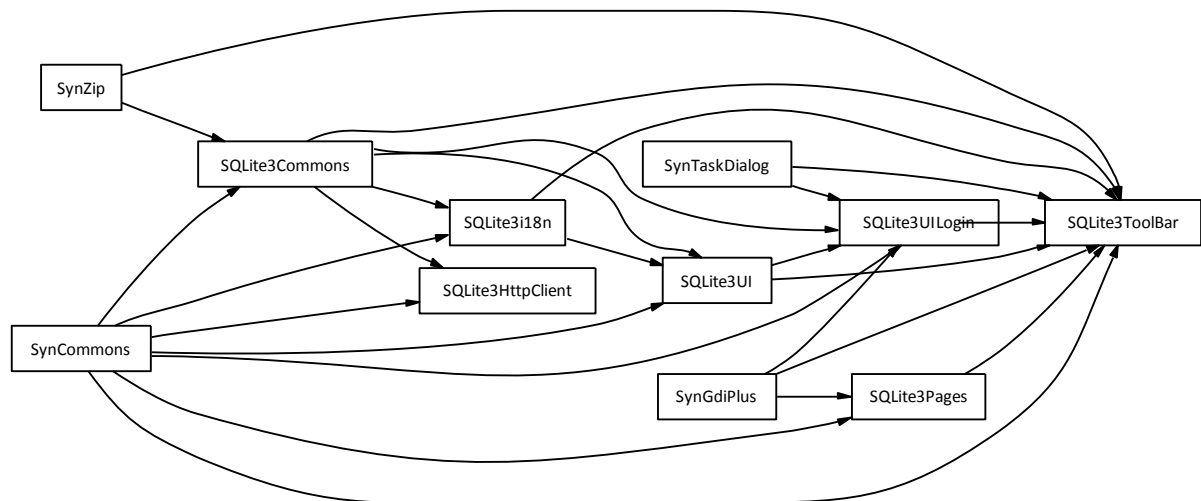
*Unit dependencies in the "Lib" directory*

**Units located in the "Lib\SQLite3\" directory:**

| Source File Name         | Description                                      | Page |
|--------------------------|--------------------------------------------------|------|
| <i>SQLite3Commons</i>    | Common ORM and SOA classes                       | 575  |
| <i>SQLite3HttpClient</i> | HTTP/1.1 RESTFUL JSON mORMot Client classes      | 731  |
| <i>SQLite3i18n</i>       | Internationalization (i18n) routines and classes | 736  |



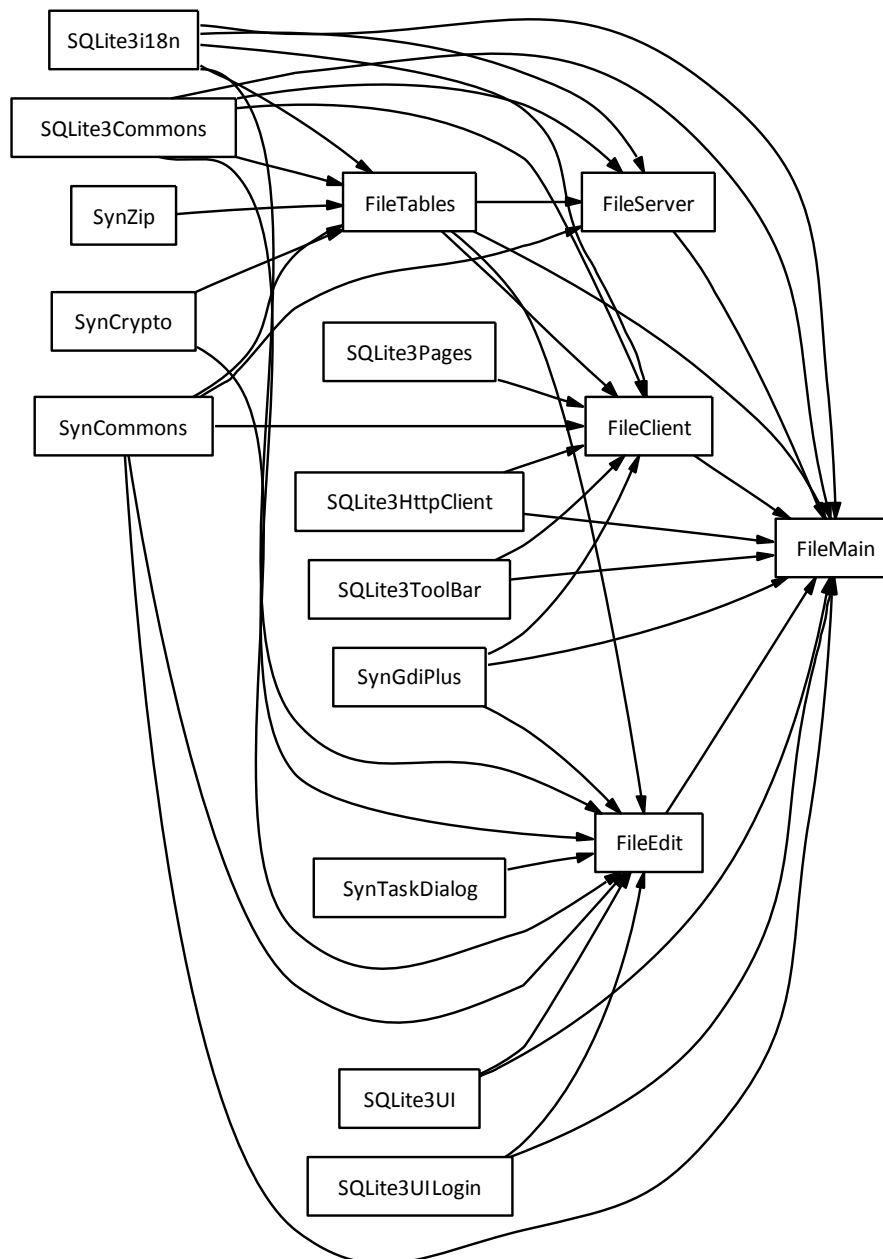
| Source File Name      | Description                                      | Page |
|-----------------------|--------------------------------------------------|------|
| <i>SQLite3Pages</i>   | Reporting unit                                   | 745  |
| <i>SQLite3ToolBar</i> | Database-driven Office 2007 Toolbar              | 768  |
| <i>SQLite3UI</i>      | Grid to display Database content                 | 781  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs | 793  |



*Unit dependencies in the "Lib\SQLite3" directory*

**Units located in the "Lib\SQLite3\Samples\MainDemo\" directory:**

| Source File Name  | Description                                              | Page |
|-------------------|----------------------------------------------------------|------|
| <i>FileClient</i> | SynFile client handling                                  | 816  |
| <i>FileEdit</i>   | SynFile Edit window                                      | 818  |
| <i>FileMain</i>   | SynFile main Window                                      | 820  |
| <i>FileServer</i> | SynFile server handling                                  | 821  |
| <i>FileTables</i> | SynFile ORM definitions shared by both client and server | 822  |



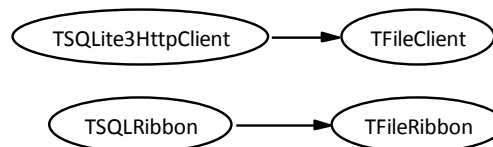
*Unit dependencies in the "Lib\SQLite3\Samples\MainDemo" directory*

#### 1.5.5.2. FileClient unit

*Purpose:* SynFile client handling

**Units used in the *FileClient* unit:**

| Unit Name                | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>FileTables</i>        | SynFile ORM definitions shared by both client and server                                                                                                                                                                                                                                                                                       | 822  |
| <i>SQLite3Commons</i>    | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                          | 575  |
| <i>SQLite3HttpClient</i> | HTTP/1.1 RESTFUL JSON mORMot Client classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                         | 731  |
| <i>SQLite3i18n</i>       | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                    | 736  |
| <i>SQLite3Pages</i>      | Reporting unit<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                                             | 745  |
| <i>SQLite3ToolBar</i>    | Database-driven Office 2007 Toolbar<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                                 | 768  |
| <i>SynCommons</i>        | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                      | 229  |
| <i>SynGdiPlus</i>        | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |



*FileClient class hierarchy*

#### Objects implemented in the *FileClient* unit:

| Objects     | Description                             | Page |
|-------------|-----------------------------------------|------|
| TFileClient | A HTTP/1.1 client to access SynFile     | 817  |
| TFileRibbon | Class used to create the User interface | 818  |

**TFileClient = class(TSQLite3HttpClient)**

*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*

TFileRibbon = **class**(TSQLRibbon)

*Class used to create the User interface*

**procedure** CreateReport(aTable: TSQLRecordClass; aID: integer; aReport:  
TGDIPages; AlreadyBegan: boolean=false); **override**;

*Overriden method used customize the report content*

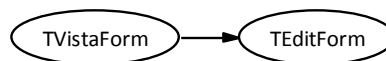
### 1.5.5.3. FileEdit unit

*Purpose: SynFile Edit window*

**Units used in the *FileEdit* unit:**

| Unit Name             | Description                                                                                                                                                                                                                                                                                                                   | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>FileTables</i>     | SynFile ORM definitions shared by both client and server                                                                                                                                                                                                                                                                      | 822  |
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                                      | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                | 736  |
| <i>SQLite3UI</i>      | Grid to display Database content<br>- this unit is a part of the freeware Synapse SQLite3 database<br>framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                      | 781  |
| <i>SQLite3UILogin</i> | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                | 793  |
| <i>SynCommons</i>     | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                  | 229  |
| <i>SynCrypto</i>      | Fast cryptographic routines (hashing and cypher)<br>- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms<br>- optimized for speed (tuned assembler and VIA PADLOCK optional<br>support)<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 383  |

| Unit Name            | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynGdiPlus</i>    | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |
| <i>SynTaskDialog</i> | Implement TaskDialog window (native on Vista/Seven, emulated on XP)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                                 | 553  |



*FileEdit class hierarchy*

#### Objects implemented in the *FileEdit* unit:

| Objects   | Description         | Page |
|-----------|---------------------|------|
| TEditForm | SynFile Edit window | 819  |

**TEditForm = class(TVistaForm)**

*SynFile Edit window*

- we don't use the standard Window generation (from SQLite3UIEdit), 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:

| Functions or procedures | Description                                                                                | Page |
|-------------------------|--------------------------------------------------------------------------------------------|------|
| Cypher                  | Will display a modal form asking for a password, then encrypt or uncrypt some BLOB content | 820  |

```
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:

```
EditForm: TEditForm;
```

*SynFile Edit window instance*

#### 1.5.5.4. FileMain unit

*Purpose:* SynFile main Window

#### Units used in the *FileMain* unit:

| Unit Name                | Description                                                                                                                                                                    | Page |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>FileClient</i>        | SynFile client handling                                                                                                                                                        | 816  |
| <i>FileEdit</i>          | SynFile Edit window                                                                                                                                                            | 818  |
| <i>FileServer</i>        | SynFile server handling                                                                                                                                                        | 821  |
| <i>FileTables</i>        | SynFile ORM definitions shared by both client and server                                                                                                                       | 822  |
| <i>SQLite3Commons</i>    | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17                       | 575  |
| <i>SQLite3HttpClient</i> | HTTP/1.1 RESTFUL JSON mORMot Client classes<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17      | 731  |
| <i>SQLite3i18n</i>       | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 736  |
| <i>SQLite3ToolBar</i>    | Database-driven Office 2007 Toolbar<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17              | 768  |
| <i>SQLite3UI</i>         | Grid to display Database content<br>- this unit is a part of the freeware Synopse SQLite3 database<br>framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17       | 781  |
| <i>SQLite3UILogin</i>    | Some common User Interface functions and dialogs<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 793  |
| <i>SynCommons</i>        | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17   | 229  |

| Unit Name         | Description                                                                                                                                                                                                                                                                                                                                    | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynGdiPlus</i> | GDI+ library API access<br>- adds GIF, TIF, PNG and JPG pictures read/write support as standard TGraphic<br>- make available most useful GDI+ drawing methods<br>- allows Antialiased rendering of any EMF file using GDI+<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 438  |



*FileMain class hierarchy*

#### Objects implemented in the *FileMain* unit:

| Objects   | Description         | Page |
|-----------|---------------------|------|
| TMainForm | SynFile main Window | 821  |

**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*

#### 1.5.5.5. FileServer unit

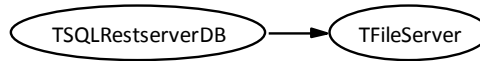
*Purpose:* SynFile server handling

#### Units used in the *FileServer* unit:

| Unit Name             | Description                                                                                                                                                                 | Page |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>FileTables</i>     | SynFile ORM definitions shared by both client and server                                                                                                                    | 822  |
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                       | 575  |
| <i>SQLite3i18n</i>    | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 736  |



| Unit Name         | Description                                                                                                                                                                  | Page |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i> | Common functions used by most Synapse projects<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 229  |



*FileServer class hierarchy*

#### Objects implemented in the *FileServer* unit:

| Objects     | Description                             | Page |
|-------------|-----------------------------------------|------|
| TFileServer | A server to access SynFile data content | 822  |

**TFileServer = class**(TSQLRestserverDB)

*A server to access SynFile data content*

**Server:** TSQLite3HttpServer;

*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** Event(**var** aParams: TSQLRestServerCallBackParams): Integer;

*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

**function** OnDatabaseUpdateEvent(Sender: TSQLRestServer; Event: TSQLiteEvent; aTable: TSQLRecordClass; aID: integer): boolean;

*Database server-side trigger which will add an event to the TSQLAuditTrail table*

**procedure** AddAuditTrail(aEvent: TFileEvent; **const** aMessage: RawUTF8='';  
aAssociatedRecord: TRecordReference=0);

*Add a row to the TSQLAuditTrail table*

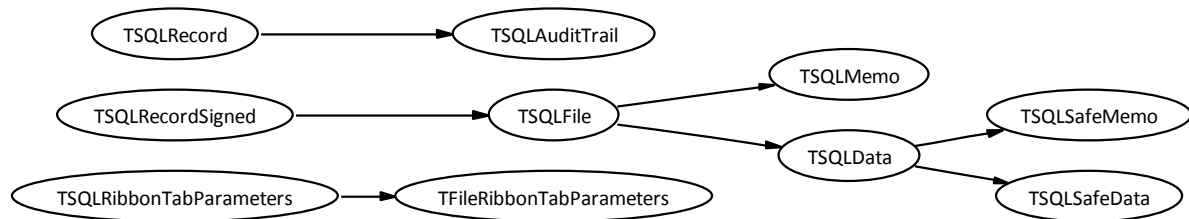
#### 1.5.5.6. FileTables unit

*Purpose:* SynFile ORM definitions shared by both client and server

#### Units used in the *FileTables* unit:

| Unit Name             | Description                                                                                                                                              | Page |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br>- this unit is a part of the freeware Synapse mORMot framework,<br>licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 575  |

| Unit Name          | Description                                                                                                                                                                                                                                                                                                             | Page |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3i18n</i> | Internationalization (i18n) routines and classes<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                             | 736  |
| <i>SynCommons</i>  | Common functions used by most Synopse projects<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                               | 229  |
| <i>SynCrypto</i>   | Fast cryptographic routines (hashing and cypher)<br>- implements AES, XOR, ADLER32, MD5, SHA1, SHA256 algorithms<br>- optimized for speed (tuned assembler and VIA PADLOCK optional support)<br>- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17 | 383  |
| <i>SynZip</i>      | Low-level access to ZLib compression (1.2.5 engine version)<br>- this unit is a part of the freeware Synopse framework, licensed under a MPL/GPL/LGPL tri-license; version 1.17                                                                                                                                         | 560  |



*FileTables class hierarchy*

#### Objects implemented in the *FileTables* unit:

| Objects                  | Description                                                   | Page |
|--------------------------|---------------------------------------------------------------|------|
| TFileRibbonTabParameters | The type of custom main User Interface description of SynFile | 824  |
| TSQLAuditTrail           | An AuditTrail table, used to track events and status          | 824  |
| TSQLData                 | An unencrypted Data table                                     | 824  |
| TSQLFile                 | An abstract class, with common fields                         | 823  |
| TSQLMemo                 | An unencrypted Memo table                                     | 824  |
| TSQLSafeData             | A crypted SafeData table                                      | 824  |
| TSQLSafeMemo             | A crypted SafeMemo table                                      | 824  |

```
TSQLFile = class(TSQLRecordSigned)
```

*An abstract class, with common fields*

**TSQLMemo = class(TSQLFile)**

*An unencrypted Memo table*  
 - will contain some text

**TSQLData = class(TSQLFile)**

*An unencrypted Data table*  
 - can contain any binary file content  
 - is also used 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, 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: 'IddId'; Actions:  
DEF_ACTIONS), (Table: TSQLData; 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) );
```

*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. uncrpyted tables*

```
GROUP_MAIN = 0;
```

*Will define the first User Interface ribbon group, i.e. main tables*

```
GROUP_SAFE = 2;
```

*Will define the 3d User Interface ribbon group, i.e. crypted 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:

| Functions or procedures | Description                          | Page |
|-------------------------|--------------------------------------|------|
| CreateFileModel         | Create the database model to be used | 826  |

```
function CreateFileModel(Owner: TSQLRest): TSQLModel;
```

*Create the database model to be used*

- shared by both client and server sides

## 2. SWRS implications

### Software Architecture Design Reference Table

The following table is a quick-reference guide to all the *Software Requirements Specifications* (SWRS) document items.

| SWRS #       | Description                                                                                                                                                                                                    | Page |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| DI-2.1.1     | The framework shall be Client-Server oriented                                                                                                                                                                  | 828  |
| DI-2.1.1.1   | A RESTful mechanism shall be implemented                                                                                                                                                                       | 828  |
| DI-2.1.1.2.1 | Client-Server Direct communication shall be available inside the same process                                                                                                                                  | 829  |
| DI-2.1.1.2.2 | Client-Server Named Pipe communication shall be made available by some dedicated classes                                                                                                                       | 829  |
| DI-2.1.1.2.3 | Client-Server Windows GDI Messages communication shall be made available by some dedicated classes                                                                                                             | 829  |
| 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) | 830  |
| DI-2.1.2     | UTF-8 JSON format shall be used to communicate                                                                                                                                                                 | 830  |
| DI-2.1.3     | The framework shall use an innovative ORM (Object-relational mapping) approach, based on classes RTTI (Runtime Type Information)                                                                               | 831  |
| DI-2.2.1     | The <i>SQLite3</i> engine shall be embedded to the framework                                                                                                                                                   | 832  |
| DI-2.2.2     | The framework libraries, including all its <i>SQLite3</i> related features, shall be tested using Unitary testing                                                                                              | 833  |
| 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       | 833  |
| DI-2.3.1.2   | Toolbars shall be able to be created from code, using RTTI and enumerations types for defining the action                                                                                                      | 834  |

| SWRS #     | Description                                                                                                                                                                                                                                                                                                                        | Page |
|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| 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 | 834  |
| DI-2.3.2   | A reporting feature, with full preview and export as PDF or TXT files, shall be integrated                                                                                                                                                                                                                                         | 835  |

## 2.1. Client Server JSON framework

### 2.1.1. 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 *Synapse mORMot Framework* shall implement such a Client-Server model by a set of dedicated classes, over various communication protocols, but in a 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:**

| Unit Name             | Description                                                                                                                       | Page |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><i>See in particular</i> TSQLRecord, TSQLRest, TSQLRestServer, TSQLRestClientURI and TSQLTableJSON. | 575  |

### 2.1.2. 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 *Synapse 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:**

| Unit Name             | Description                                                                                            | Page |
|-----------------------|--------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><i>See in particular TSQLRest, TSQLRestServer and TSQLRestClientURI.</i> | 575  |

### 2.1.3. 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:**

| Unit Name             | Description                                                                                                                         | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><i>See in particular TSQLRestServer, URIRequest, USEFASTMM4ALLOC and TSQLRestClientURID11.Create.</i> | 575  |

### 2.1.4. 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:**

| Unit Name             | Description                                                                                                                                           | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><i>See in particular TSQLRestServer.ExportServerNamedPipe, TSQLRestClientURINamedPipe.Create and TSQLRestClientURI.</i> | 575  |

### 2.1.5. SWRS # DI-2.1.1.2.3

**Client-Server Windows GDI Messages communication shall be made available by some dedicated classes**

Client-Server Windows GDI Messages communication shall be made available by some dedicated classes.

**This specification is implemented by the following units:**

| Unit Name             | Description                                                                                                         | Page |
|-----------------------|---------------------------------------------------------------------------------------------------------------------|------|
|                       | Common ORM and SOA classes                                                                                          |      |
| <i>SQLite3Commons</i> | <i>See in particular TSQLRestClientURI, TSQLRestServer.ExportServerMessage and TSQLRestClientURIMessage.Create.</i> | 575  |

#### 2.1.6. 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:**

| Unit Name                | Description                                                                                                                      | Page |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------|------|
|                          | Common ORM and SOA classes                                                                                                       |      |
| <i>SQLite3Commons</i>    | <i>See in particular TSQLRestClientURI and TSQLRestServer.URI.</i>                                                               | 575  |
|                          | HTTP/1.1 RESTFUL JSON mORMot Server classes                                                                                      |      |
| <i>SQLite3HttpServer</i> | <i>See in particular TSQLite3HttpServer.Create, TSQLite3HttpServer.DBServer and TSQLite3HttpServer.AddServer.</i>                | 733  |
|                          | Classes implementing HTTP/1.1 client and server protocol                                                                         |      |
| <i>SynCrtSock</i>        | <i>See in particular THttpServer.Create, THttpApiServer.Create, THttpServerGeneric.Request and THttpServerGeneric.OnRequest.</i> | 369  |
|                          | HTTP/1.1 RESTFUL JSON mORMot Client classes                                                                                      |      |
| <i>SQLite3HttpClient</i> | <i>See in particular TSQLite3HttpClient.Create.</i>                                                                              | 731  |

#### 2.1.7. 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 *Synapse 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 formating 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:**

| Unit Name             | Description                                                                                                                                                                                                                                                                                                                                                            | Page |
|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i>     | Common functions used by most Synopse projects<br><br><i>See in particular</i> <code>TTextWriter.Create</code> , <code>TTextWriter.AddJSONEscape</code> , <code>IsJSONString</code> , <code>JSONDecode</code> , <code>JSONEncode</code> , <code>JSONEncodeArray</code> , <code>GetJSONField</code> and <code>JSON_CONTENT_TYPE</code> .                                | 229  |
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><br><i>See in particular</i> <code>TSQLTable.GetJSONValues</code> , <code>TSQLTableJSON.Create</code> , <code>TSQLTableJSON.UpdateFrom</code> , <code>TJSONWriter.Create</code> , <code>TSQLRecord.CreateJSONWriter</code> , <code>TSQLRecord.GetJSONValues</code> , <code>GetJSONObjectAsSQL</code> and <code>UnJSONFirstField</code> . | 575  |

### 2.1.8. 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:**

| Unit Name             | Description                                                                                                                                                                                                                                                                                                                                            | Page |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><br><i>See in particular</i> <code>TClassProp</code> , <code>TClassType</code> , <code>TEnumType</code> , <code>TTypeInfo</code> , <code>TSQLRecord.ClassProp</code> , <code>TSQLRecord.GetJSONValues</code> , <code>TPropInfo.GetValue</code> , <code>TPropInfo.SetValue</code> and <code>TSQLRecordProperties</code> . | 575  |

## 2.2. SQLite3 engine

### 2.2.1. 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 *Synopse 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 *Synopse mORMot Framework*, and interact directly from the Delphi application process.

The *Synopse 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:**

| Unit Name             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3Commons</i> | Common ORM and SOA classes<br><br><i>See in particular</i> <code>TSQLRestServer</code> .                                                                                                                                                                                                                                                                                                                                                                                                        | 575  |
| <i>SynSQLite3</i>     | SQLite3 embedded Database engine direct access<br><br><i>See in particular</i> <code>TSQLRequest.Execute</code> , <code>TSQLDataBase</code> , <code>TSQLTableDB.Create</code> , <code>TSQLite3Blob</code> , <code>TSQLite3DB</code> , <code>TSQLite3FunctionContext</code> , <code>TSQLite3Statement</code> , <code>TSQLite3Value</code> , <code>TSQLite3ValueArray</code> , <code>TSQLTableDB</code> , <code>TSQLRequest</code> , <code>TSQLBlobStream</code> and <code>ESQLException</code> . | 502  |

| Unit Name      | Description                                                                                                                      | Page |
|----------------|----------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3</i> | SQLite3 embedded Database engine used as the kernel of mORMot<br><i>See in particular TSQLRestServerDB and TSQLRestClientDB.</i> | 567  |

### 2.2.2. 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 *Synapse mORMot Framework* shall use all integrated Unitary testing features provided by a common testing framework integrated to all Synapse 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 bug.

**This specification is implemented by the following units:**

| Unit Name                | Description                                                                                                                                                           | Page |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SynCommons</i>        | Common functions used by most Synapse projects<br><i>See in particular TTestLowLevelCommon.</i>                                                                       | 229  |
| <i>SQLite3Commons</i>    | Common ORM and SOA classes<br><i>See in particular TTestLowLevelTypes and TTestBasicClasses.</i>                                                                      | 575  |
| <i>SQLite3</i>           | SQLite3 embedded Database engine used as the kernel of mORMot<br><i>See in particular TTestSQLite3Engine, TTestFileBased, TTestMemoryBased and TTestFileBasedWAL.</i> | 567  |
| <i>SQLite3HttpServer</i> | HTTP/1.1 RESTFUL JSON mORMot Server classes<br><i>See in particular TTestClientServerAccess.</i>                                                                      | 733  |

## 2.3. User interface

### 2.3.1. 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:**

| Unit Name             | Description                                                                                        | Page |
|-----------------------|----------------------------------------------------------------------------------------------------|------|
| <i>SQLite3UI</i>      | Grid to display Database content<br><i>See in particular</i> <code>TSQLTableToGrid.Create</code> . | 781  |
| <i>SQLite3ToolBar</i> | Database-driven Office 2007 Toolbar<br><i>See in particular</i> <code>TSQLLister.Create</code> .   | 768  |

### 2.3.2. 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:**

| Unit Name             | Description                                                                                                                                                                                     | Page |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3ToolBar</i> | Database-driven Office 2007 Toolbar<br><i>See in particular</i> <code>TSQLRibbon.Create</code> , <code>TSQLRibbonTab</code> , <code>TSQLLister</code> and <code>TSQLCustomToolBar.Init</code> . | 768  |

### 2.3.3. 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:**

| Unit Name          | Description                                                                                                                                                                                                                                                                                                                                                                                                                                 | Page |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <i>SQLite3i18n</i> | Internationalization (i18n) routines and classes<br><i>See in particular</i> <code>TLanguage</code> , <code>TLanguageFile.Create</code> , <code>S2U</code> , <code>U2S</code> , <code>TLanguageFile.StringToUTF8</code> , <code>TLanguageFile.TimeToText</code> , <code>TLanguageFile.DateToText</code> , <code>TLanguageFile.DateTimeToText</code> , <code>TLanguageFile.UTF8ToString</code> , <code>TLanguageFile.Translate</code> and —. | 736  |

#### 2.3.4. 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 *Synapse 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:**

| Unit Name           | Description                                                                                 | Page |
|---------------------|---------------------------------------------------------------------------------------------|------|
| <i>SQLite3Pages</i> | Reporting unit<br><i>See in particular TGDIPages.</i>                                       | 745  |
| <i>SynGdiPlus</i>   | GDI+ library API access<br><i>See in particular TGDIPPlus.DrawAntiAliased.</i>              | 438  |
| <i>SynPdf</i>       | PDF file generation<br><i>See in particular TPdfDocument and TPdfCanvas.RenderMetaFile.</i> | 459  |