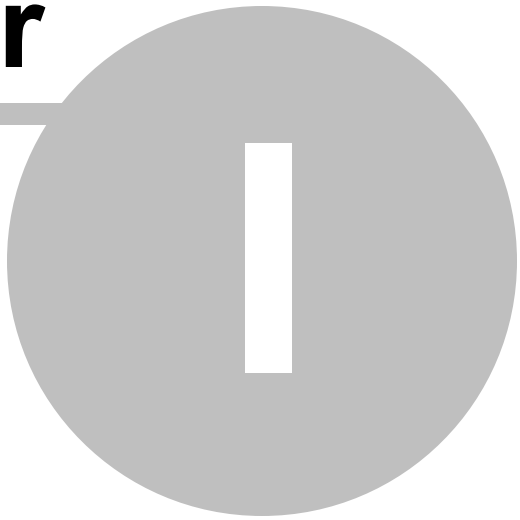
TMS Aurelius Manual

October, 2013

Web: http://www.tmssoftware.com
E-mail: info@tmssoftware.com

Table of Contents

Chapter



Introduction

1 Introduction

TMS Aurelius is an Object-Relational Mapping (ORM) framework. Its purpose is to be the definitive ORM framewok for the Delphi/C++ Builder environment, with full support for data manipulation, complex and advanced queries, inheritance, polymorphism, among others. This manual covers all topics needed for you to know about Aurelius and start using it.

TMS Aurelius product page: http://www.tmssoftware.com/site/aurelius.asp
TMS Software site: http://www.tmssoftware.com

Manual Topics

1.1 Benefits

Aurelius brings all benefits an application can obtain from using an ORM framework. Main ones are:

- Productivity: Avoid complex SQL statements that can only be verified at runtime. Code directly with objects.

Instead of this code:

```
Query1.Sql.Text := 'SELECT I.ID AS INVOICE_ID, I.INVOICE_TYPE,
I.INVOICENO, I.ISSUE_DATE, I.PRINT_DATE, ' +
  'C.ID AS CUSTOMER_ID, C.CUSTOMER_NAME, C.SEX, C.BIRTHDAY, N.ID AS
COUNTRY_ID, N.COUNTRY_NAME' +
  'FROM INVOICE AS I INNER JOIN CUSTOMER AS C ON (C.ID = I.CUSTOMER_ID) '
+
  'LEFT JOIN COUNTRY AS N ON (N.ID = C.COUNTRY_ID)' +
  'WHERE I.ID = :INVOICE_ID;'
Query1.ParamByName('INVOICE_ID').AsInteger := 1;
Query1.Open;
ShowMessage(Format('Invoice No: %d, Customer: %s, Country: %s',
  [Query1.FieldByName('INVOICE_ID').AsInteger,
  Query1.FieldByName('CUSTOMER_NAME').AsString,
  Query1.FieldByName('COUNTRY_NAME').AsString]));
```

Write this code:

```
Invoice := Manager1.Find<TInvoice>(1);
ShowMessage(Format('Invoice No: %d, Customer: %s, Country: %s',
  [Invoice.InvoiceNo, Invoice.Customer.Name,
Invoice.Customer.Country.Name]));
```

- Maintainability: Clearer business logic by dealing with objects, hiding all the database-access layer.
- Portability: Easily change the underlying database - all your business code stays the same since they are just pure objects.

1.2 Features

Here is a list of main features of TMS Aurelius framework:

- Several database servers supported (MS SQL Server, Firebird, MySQL)
- Several database-access components supported (dbExpress, AnyDac, SQLDirect, UniDac, ADO)
- Multi-platform solution - Win32, Win64, Mac OS X, VCL, FireMonkey
- Saving, updating and loading of entity objects in an object-oriented way
- Queries - Powerful query API using criteria expressions, projections, grouping, conditions and even logical operators in a LINQ-like approach
- Inheritance mapping and polymorphism - map a full class hierarchy into the database
- Visual data binding with data-aware controls using full-featured TAureliusDataset component
- Cross-database development - use a single Delphi code to target multiple databases in a transparent way
- Choose from classes-to-database approach (creating the database structure from classes) or database-to-classes approach (creating classes source code from database, using TMS Data Modeler)
- Mapping directly in classes using custom attributes
- Association mapping
- Lifetime management of objects using object manager
- Cached and identity-mapped objects
- Automatic database structure generation
- Nullable types support
- Lazy loading for associations and blob fields
- Allows logging of SQL commands
- Allows mapping enumerated types to database values
- Open architecture - easy extendable to use different component sets or database servers
- Available for Delphi 2010 and up.

1.3 What's New

version 2.2 (Oct-2013)

- New: Increased querying capabilities with newTExpression/TLinq methods that allow comparing a projection to any other projection (in addition to comparing to values only)

- New: Support for Rad Studio XE5
- New: Connection driver for XData RemoteDB
- New: TCriteria.AutoDestroy property allows keeping TCriteria in memory after objects are retrieved
- Changed: Packages structure. See breaking changes.
- Fixed: Error when deserializing a Json array representing an existing object list, when class member was a proxy
- Fixed: Exception not being raised when calling TClassHierarchyExplorer.GetAllSubClasses
- Fixed: Wrong default values when inserting a record in XE4 with TAureliusDataset
- Fixed: IBObjects driver now correctly performing statements using IB_Session object specified in the TIBODatabase

version 2.1 (May-2013)
- New: Full iOS support, including native access to SQLite database
- New: Support for Rad Studio XE4
- Fixed: Not possible to create unique keys referencing columns declared using ForeignJoinColumn attributes
- Fixed: Merge cascades not being applied correctly
- Fixed: Access violation when loading package multiple times in TAureliusDataset design-time editor
- Fixed: Wrong example in documentation about lazy-loading associations in distributed applications (proxy loader)
- Fixed: Schema validation example code in manual
- Fixed: Error using transactions with IBExpress, IBObjects and DirectOracleAccess components
- Changed: Live bindings disabled by default

version 2.0 (Apr-2013)
- New: Update Database Schema feature (TDatabaseManager.UpdateDatabase method)
- New: Database Schema validation feature (TDatabaseManager.ValidateDatabase method)
- New: Detailed Database Schema analysis when updating/validating/creating (TDatabaseManager properties: Actions, Warnings Errors)
- New: TMappingSetup.MappedClasses property allows defining different class entities for different setups (and thus databases/connections)
- New: TDatabaseManager.SQLExecutionEnabled property allows generating scripts to update/create/drop database schema without effectively execute statements
- New: TSQLiteNativeConnectionAdapter.EnableForeignKeys and DisableForeignKeys methods allow control when foreign keys are enforced in SQLite connections
- Improved: TGlobalConfig.AutoSearchMappedClasses property removed
- Fixed: Conversion error in TAureliusDataset entity fields when using live bindings

version 1.9 (Feb-2013)
- New: Support for Unified Interbase (UIB) components
- Improved: Statements to generate MS SQL Server database structure now explicitly declare NULL constraint when creating fields
- Improved: Auto mapping now automatically includes TColumnProp.NoUpdate in ID column properties
- Improved: Retrieving objects (Find) with null id in database now raises an exception instead of just returning a nil instance
- Fixed: Error when flushing objects with many-valued-association declared before id fields and which foreign key field had same name as id field
- Fixed: Cascade not being applied when flushing objects with single-valued associations

pointing to unmanaged (transient) instances
- Fixed: Exception when setting TAureliusDataset.Filtered := true when dataset is active
- Fixed: Specific conversion issue when retrieving TGuid value from UNIQUEIDENTIFIER fields, using SQL-Direct with server type set to stSQLServer
- Fixed: Error when deserializing Nullable<double> types using JSON deserializer
- Fixed: Uses clause in Direct Oracle Access driver included a wrong unit name

version 1.8 (Jan-2013)
- New: Support for Direct Oracle Access components
- Improved: Updated source code to work correctly When recompiling with Assertions off
- Fixed: Error using TAureliusDataset.Locate with nullable string fields when there were null fields in dataset
- Fixed: Rare memory leak when using some specific compiler settings (Optimizations=On)
- Fixed: Memory leak in "Getting Started" demo

version 1.7 (Dec-2012)
- New: Full JSON support makes it easy to build distributed applications
- New: Enumeration field as string now possible in TAureliusDataset by using field name sufix ".EnumName"
- Improved: IdEq method in TLinq
- Improved: TGlobalConfigs.AutoMappingDefaultCascade now split in two different properties for Association and ManyValuedAssociation (breaking change)
- Fixed: TGuid properties and fields were causing occasional errors in Flush method calls

version 1.6 (Sep-2012)
- New: Delphi XE3 support
- New: Support for FIBPlus components
- New: TCriteria.RemovingDuplicatedEntities allows removing duplicated objects from result list
- New: Properties Count and PropNames in TCriteriaResult object provides additional info about retrieved projections
- Improved: Better support for other date types (string and julian) in SQLite database
- Improved: Possibility to use descendants of TList<T>/TObjectList<T> for many-valued associations
- Improved: Non-generic TObjectManager.Find method overload accepting a class type as parameter
- Fixed: Memory leak when creating a default TMappingExplorer
- Fixed: Error when saving collection items belonging to a joined-tables class hierarchy
- Fixed: Cascade removal was not removing lazy-loaded associations if the associations were not loaded

version 1.5 (Jun-2012)
- New: Guid, Uuid38, Uuid36 and Uuid32 identifier generators allow client-side automatic generation of GUID and/or string identifiers
- New: TExpression.Sql and TProjections.Sql methods for adding custom SQL syntax to a query, increasing flexibility in query construction
- New: Support for properties/fields of type TGuid, which are now mapped to database Guid/Uniqueidentifier fields (if supported by database) or database string fields
- New: Support for Absolute Database

version 1.4 (May-2012)
- New: Dynamic properties allows mapping to database columns at runtime
- Improved: TCriteriaResult object can retrieved projected values by projection alias

- Improved: TCriteriaResult objects supported in TAureliusDataset
- Improved: Better validation of MappedBy parameter in ManyValuedAssociation attribute
- Improved: TAureliusDataset.Post method now saves object if it's not persisted, even in edit mode
- Fixed: Issue with association as part of composite id when multiple associations are used in cascaded objects
- Fixed: Manual Quick Start example updated with correct code
- Fixed: Automapping was not correctly defining table name in some situations with inherited classes

- **version 1.3 (Mar-2012)**
- New: Paged fetch-on-demand using TAureliusDataset.SetSourceCriteria allows fetching TDataset records on demand without keeping an open database connection
- New: Fetch-on-demand support on TAureliusDataset, by using SetSourceCursor method
- New: Support for ElevateDB database server
- New: Paging query results now supported by using new TCriteria methods Skip and Take
- New: TCriteria.Open method allows returning a cursor for fetching objects on demand
- New: TBlob.LoadFromStream and SaveToStream methods for improved blob manipulation
- New: "Not" operator supported in TLinq expressions and "Not_" method in TExpression
- New: TAureliusDataset.InternalList property allows access to the internal object list
- Improved: TObjectManager.Find<T> method introduced as an alias for CreateCriteria<T> method for query creation
- Improved: TCriteria.UniqueResult now returns nil if no objects are returned
- Improved: TCriteria.UniqueResult returns the unique object even if the object is returned in more than one row (duplicated rows of same object)
- Improved: NexusDB through UniDac components now supported

version 1.2 (Mar-2012)
- New: Fully documented TAureliusDataset component for visual binding objects to data-aware controls.
- New: Support for UniDac components
- Improved: Better error handling with more detailed and typed exceptions being raised at key points, especially value conversion routines
- Improved: IBObjects adapter now can adapt any TIB_Connection component, not only TIBODatabase ones
- Improved: Better exception messages for convert error when load entity property values from database
- Fixed: issue with SQL statement when using more than 26 eager-loading associations
- Fixed: Issue when selecting objects with non-required associations and required sub-associations
- Fixed: Issue with lazy-loaded proxies using non-id columns as foreign keys
- Fixed: adding Automapping attribute was not requiring Entity attribute to be declared
- Fixed: Automapping in a subclass in a single-table hierarchy caused issues when creating database schema
- Fixed: Memory leak in MusicLibrary demo

version 1.1 (Feb-2012)
- New: TObjectDataset preview (for registered users only)
- New: Support for IBObjects components
- Improved: MusicLibrary demo refactored to use best-designed controllers
- Improved: Access Violation replaced by descriptive error message when SQL dialect was not found for connection
- Fixed: Registered version installer sometimes not correctly detecting XE/XE2 installation

- Fixed: Memory leak is some specific situations with automapped associations
- Fixed: Default value of OwnsObjects property in TObjectManager changed from false to true (as stated by documentation)
- Fixed: Memory leak in MusicLibrary demo
- Fixed: Component adapter was ignoring explicitly specified SQL dialect
- Fixed: Issue with automapping self-referenced associations

version 1.0 (Jan-2012)
- First public release

1.4 Copyright Notice

TMS Aurelius framework trial version is free for use in non-commercial applications, that is any software that is not being sold in one or another way or that does not generate income in any way by the use of the application.

For use in commercial applications, you must purchase a single license, a small team license or a site license. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates for a full version cycle and priority email support. A single developer license allows ONE developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A small team license allows TWO developers within a company to use the components for commercial application development, to obtain free updates and priority email support. Single developer and small team licenses are NOT transferable to another developer within the company or to a developer from another company. All licenses allow royalty free use of the components when used in binary compiled applications.

The component cannot be distributed in any other way except through TMS Software web site. Any other way of distribution must have written authorization of the author.

Online registration for TMS Aurelius is available at http://www.tmssoftware.com. Source code & license is sent immediately upon receipt of check or registration by email.

1.5 Getting Support

General notes
Before contacting support:
- Make sure to read this whole manual and any readme.txt or install.txt files in component distributions, if available.
- Search TMS support forum and TMS newsgroups to see if you question hasn't been already answer.
- Make sure you have the latest version of the component(s).

When contacting support:
- Specify with which component is causing the problem.
- Specify which Delphi or C++Builder version you're using and preferably also on which OS.

- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.

- Send email from an email account that
1) allows to receive replies sent from our server
2) allows to receive ZIP file attachments
3) has a properly specified & working reply address

Getting support
For general information: info@tmssoftware.com
Fax: +32-56-359696
For all questions, comments, problems and feature request for VCL components: help@tmssoftware.com

Important note:
All topics covered by this manual are officially supported and it's unlikely that future versions will break backward compatibility. If this ever happens, all breaking changes will be covered in this manual and guidelines to update to a new version will be described. However, it's important to note that parts of TMS Aurelius code that are undocumented are not officially supported and are **subject to change**, which includes breaking backward compatibility. In case you are using an unsupported/undocumented feature we will not provide support for upgrading and will not officially support it.

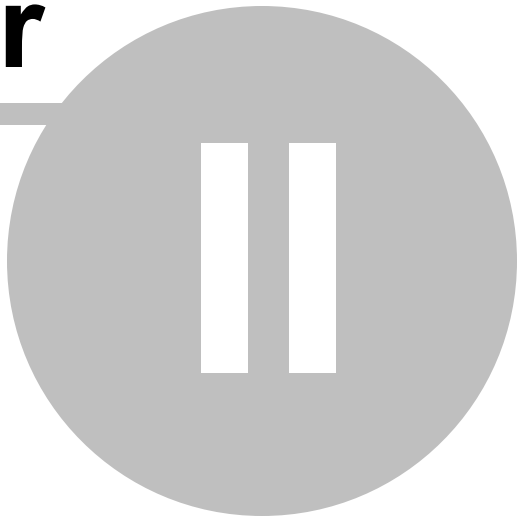
1.6 Breaking Changes

List of changes in each version that breaks backward compatibility from a previous version.

Version 2.2

Packages were restructured to use LIBSUFIX, which means dcp (Delphi Compiled Package) files won't have the a suffix indicating Delphi version. For example, in previous versions, the compiled package file for Delphi XE3 would be aureliusxe3.dcp. From version 2.2 and on, file name will be simply aurelius.dcp. Your application might be affected by this if you have packages that requires Aurelius packages. You will have you update your package files to require package "aurelius" instead of requiring package "aureliusxe3" (or whatever Delphi version you use). BPL files are unchanged, still keeping delphi version suffix (aureliusxe3.bpl).

Chapter



Getting Started

2 Getting Started

In this chapter we will provide you basic info about how to get started using TMS Aurelius. They are simple examples, but shows you how quickly you can start use it, and how simple is that. The intention is to explain the macro structure of the framework and what are the major steps to setup it. For a full usage of the framework and full flexibility, see other chapters in this manual.

The following topics are covered in this chapter:

2.1 Quick Start

Here we describe minimal steps to get started using TMS Aurelius framework.

1. Create the class model

Create a new class to be saved in the database (you can also use an existing class in your application):

```
type
  TPerson = class
  private
    FLastName: string;
    FFirstName: string;
    FEmail: string;
  public
    property LastName: string read FLastName write FLastName;
    property FirstName: string read FFirstName write FFirstName;
    property Email: string read FEmail write FEmail;
  end;
```

Your class can descend from any other Delphi class.

2. Define and map persistent entity class

Add Entity and Automapping attributes to the class, and an integer FId field. This will do automatic mapping.
(All attributes you need are declared in unit Aurelius.Mapping.Attributes so you must add it to your unit)

```
uses
  {...}, Aurelius.Mapping.Attributes;

type
  [Entity]
  [Automapping]
```

```
  TPerson = class
  private
    FId: integer;
    FLastName: string;
    FFirstName: string;
    FEmail: string;
  public
    property Id: integer read FId;
    property LastName: string read FLastName write FLastName;
    property FirstName: string read FFirstName write FFirstName;
    property Email: string read FEmail write FEmail;
  end;
```

You can also fully customize mapping - there is no need to use automatic one. Even including an FId is not required if you don't use automatic mapping.

3. Obtain an IDBConnection interface

Get the component you use in your application to connect to the database (dbExpress, ADO) and obtain an IDBConnection interface from it.
(The IDBConnection interface is declared in Aurelius.Drivers.Interfaces unit. Each adapter is declared in a different unit, you must check which unit you must use for each available adapter).

```
uses
  {...}, Aurelius.Drivers.Interfaces, Aurelius.Drivers.dbExpress;

var
  MyConnection: IDBConnection
begin
  // SQLConnection1 is a dbExpress TSQLConnection component
  // You can use several different data-access component libraries
  MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
false);
```

4. Specify the SQL dialect

Let Aurelius know which SQL dialects wlll be available to the application. You do that by adding a unit named Aurelius.SQL.XXX (where XXX is the name of SQL dialect) to any unit of your application, or the project itself.

```
uses
  {...}, Aurelius.SQL.MySQL, Aurelius.SQL.MSSQL;
```

In the example above, we make Aurelius aware of MySQL and Microsoft SQL Server dialects. The correct dialect will be chosen by Aurelius depending on the connection you specified in step 3. In that step (3) you can even specify which dialect you are using. There are plenty of SQL dialects you can use in Aurelius.

5. Create the database

Use the Database Manager to create the underlying database tables and fields where the objects will be saved.
(TDatabaseManager is declared in unit Aurelius.Engine.DatabaseManager):

```
uses
  {...}, Aurelius.Engine.DatabaseManager;

DBManager := TDatabaseManager.Create(MyConnection);
DBManager.BuildDatabase;
```

If you have an existing database with specific fields and tables you want to use, just skip this step.

6. Instantiate and save objects

Now you can instantiate a new TPerson instance and save it in the database, using the object manager:

```
uses
  {...}, Aurelius.Engine.ObjectManager;

Person := TPerson.Create;
Person.LastName := 'Lennon';
Person.FirstName := 'John';
Person.Email := 'lennon@beatles.com';
Manager := TObjectManager.Create(MyConnection);
try
  Manager.Save(Person);
  PersonId := Person.Id;
finally
  Manager.Free;
end;
```

A new record will be created in the database. Person.Id will be generated automatically.

7. Retrieve and update objects

```
Manager := TObjectManager.Create(MyConnection);
Person := Manager.Find<TPerson>(PersonId);
Person.Email := 'john.lennon@beatles.org';
Manager.Flush;
Manager.Free;
```

This way you can retrieve object data, update values and save it back to the database

8. Perform queries

What if you want to retrieve all persons which e-mail belongs to domain "beatles.org" or "beatles.com"?

(There are several units you can use to build queries. Aurelius.Criteria.Base must be always used, then for filter expressions you can use Aurelius.Criteria.Expression or Aurelius.Criteria.Linq if you prefer using linq-like operators. To use projections, use Aurelius.Criteria.Projections unit)

```
uses
  {...}, Aurelius.Criteria.Base, Aurelius.Criteria.Linq;

Manager := TObjectManager.Create(MyConnection);
Results := Manager.Find<TPerson>
  .Where(
     TLinq.Like('Email', '%beatles.org%')
     or TLinq.Like('Email', '%beatles.com%')
     )
  .List;

// Iterate through Results here, which is a TList<TPerson> list.
for person in Results do
  // use person variable here, it's a TPerson object

Manager.Free;
```

9. What's Next?

With just the above steps you are able to create the database and SAVE your classes in there, being able to save, delete, update and query objects. But what if you want:

a) Create a new class TCompany descending from TPerson and also save it?
Aurelius supports inheritance strategies using the Inheritance attribute.

b) Fine-tune the mapping to define names and types of the table columns where the class properties will be saved to?
You can do manual mapping using several attributes like Table and Column to define the database table and columns. You can even use Nullable<T> types to specify fields that can receive null values.

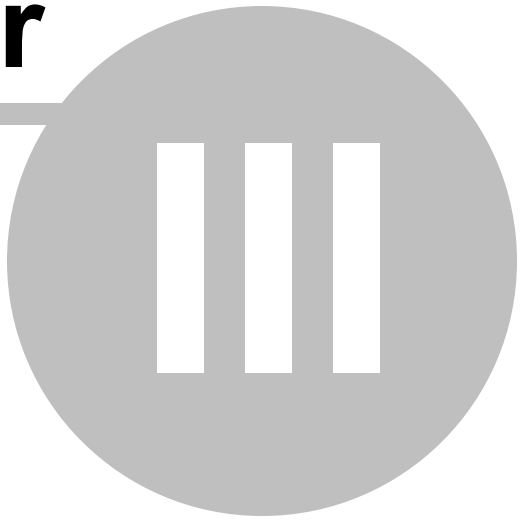
c) Create properties that are also objects or list of objects (e.g., a property Country: TCountry in my TPerson class), and also save them?
You can do it, using associations that can be fetched in a lazy or eager mode. You do that using Association and ManyValuedAssociation attributes.

d) Define different identifier strategies, define sequences, unique indexes, etc., in my database?
Just use the several mapping attributes available.

e) Perform complex queries using different conditional expressions, grouping, ordering, aggregated functions, condition expression in associated objects, etc.?
Aurelius allow you to create complex queries using all the mentioned features and more, all at object-level. You don't need to use SQL statements for that.

f) Send/receive Aurelius objects in JSON format through REST servers or any other multi-tier architecture? You can build distributed applications with Aurelius.

Chapter



Database Connectivity

3 Database Connectivity

This chapter explains how you properly configure Aurelius to access the database where objects will be saved to. The following topics describe all ways you can connect to, which dialects are supported and even how to automatically create/destroy the database schema.

IDBConnection Interface
Component Adapters
SQL Dialects
Components and Databases Homologation
Database Manager - Creating the Schema

3.1 IDBConnection Interface

The IDBConnection interface represents a connection to a database in Aurelius. Every object that needs to connect to a database just uses this interface to send and receive data from/to the database. As an example, when you create a TObjectManager object, you need to pass a IDBConnection interface to it so it can connect to the database.

IDBConnection wraps the data access component you are using, making it transparent for the framework. Thus, regardless if you connect to the database using dbExpress, ADO, IBX, etc., you just need IDBConnection.

To obtain an IDBConnection interface you use the adapters in Aurelius. The adapters just take an existing data access component (TSQLConnection, TADOConnection, etc.) and give you back the IDBConnection interface you need to use. To create database connections it's important to know the available:

Component Adapters
SQL Dialects

In summary:

To obtain an IDBConnection interface

1. Create and configure (or even use an existing one) component that makes a connection to your database

If you use dbExpress, for example, just drop a TSQLConnection component on the form and configure it. Or you can just use the existing one you have in your application. Suppose this component is named SQLConnection1.

```
SQLConnection1: TSQLConnection;
```

2. Instantiate an adapter passing the connection component

```
var
  MyConnection: IDBConnection;
begin
  MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
False);
  // Use your connection now
  Manager := TObjectManager.Create(MyConnection);
  ...
end;
```

Memory Management

Note the second boolean parameter in the Create constructor of the adapter. It indicates if the underlying connection component will be destroyed when the IDBConnection interface is destroyed. In the example above, the SQLConnection1 component will remain in memory, even after MyConnection interface is out of scope and released. If you want the component to be destroyed, just pass the second parameter as true. You will usually use this option when you create a connection component just for Aurelius usage. If you are using an existing component from your application, use false. Quick examples below:

```
var
  MyConnection: IDBConnection;
begin
  MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
False);
  // ...
  MyConnection := nil;
  { MyConection is nil, the TDBExpressConnectionAdapter component is
destroyed,
    but SQLconnection1 component remains in memory}
end;
```

```
var
  MyConnection: IDBConnection;
  SQLConnection1: TSQLConnection;
begin
  SQLConnection1 := TSQLConnection.Create(nil);
  // Set SQLConnection1 properties here in code
  MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
True);
  // ...
  MyConnection := nil;
  { MyConection is nil, the TDBExpressConnectionAdapter component is
destroyed,
    and SQLConnection1 is also destroyed }
end;
```

3.2 Component Adapters

There is an adapter for each data-access component. For dbExpress, for example, you have TDBExpressConnectionAdapter, which is declared in unit Aurelius.Drivers.dbExpress. All adapters are declared in unit Aurelius.Drivers.XXX where XXX is the name of data-access technology you're using. You can create your own adapter by implementing IDBConnection interfaces, but Aurelius already has the following adapters available:

Technology	Adapter class	Declared in unit	Adapted Component	Vendor Site
Absolute Database	TAbsoluteDBConnectionAdapter	Aurelius.Drivers.AbsoluteDB	TABSDatabase	http://www.componentace.com
AnyDac	TAnyDacConnectionAdapter	Aurelius.Drivers.AnyDac	TADDatabase	http://www.da-soft.com/anydac
dbExpress	TDBExpressConnectionAdapter	Aurelius.Drivers.dbExpress	TSQLConnection	Delphi Native
dbGo (ADO)	TDbGoConnectionAdapter	Aurelius.Drivers.dbGo	TADOConnection	Delphi Native
Direct Oracle Access (DOA)	TDoaConnectionAdapter	Aurelius.Drivers.Doa	TOracleSession	http://www.allroundautomations.com
ElevateDB	TElevateDBConnectionAdapter	Aurelius.Drivers.ElevateDB	TEDBDatabase	http://elevatesoftware.com/
FIBPlus	TFIBPlusConnectionAdapter	Aurelius.Drivers.FIBPlus	TFIBDatabase	http://www.devrace.com/
IBObjects (IBO)	TIBObjectsConnectionAdapter	Aurelius.Drivers.IBObjects	TIBODatabase	http://www.ibobjects.com/
Interbase Express (IBX)	TIBExpressConnectionAdapter	Aurelius.Drivers.IBExpress	TIBDatabase	Delphi Native
NexusDB	TNexusDBConnectionAdapter	Aurelius.Drivers.NexusDB	TnxDatabase	http://www.nexusdb.com
SQL-Direct	TSQLDirectConnectionAdapter	Aurelius.Drivers.SqlDirect	TSDDatabase	http://www.sqldirect-soft.com
SQLite	TSQLiteNativeConnectionAdapter	Aurelius.Drivers.SQLite	(not applicable)	TMS Aurelius Native
UniDac	TUniDacConnectionAdapter	Aurelius.Drivers.UniDac	TUniConnection	http://www.devart.com/unidac
Unified Interbase (UIB)	TUIBConnectionAdapter	Aurelius.Drivers.UIB	TUIBDatabase	http://sourceforge.net/projects/uib/
XData RemoteDB Server	TRemoteDBConnectionAdapter	Aurelius.Drivers.RemoteDB	TXDatabase	http://www.tmssoftware.com/site/xdata.asp

Creating the adapter

To create the adapter, you just need to instantiate it, passing an instance of the component to be adapted. In the example below, a dbExpress adapter constructor receives a TSQLConnection component.

```
MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
False);
```

The adapter detect the SQL Dialect automatically, but you can force the adapter to use a specific dialect. See SQL Dialects section for more details.

Native SQLite Adapter
Aurelius provides native SQLite database adapter. You just need to have sqlite3.dll in a path Windows/Mac can find. Creating SQLite adapter is a little different than other adapters, since you don't need to pass a component to be adapter. With the SQLite adapter, you just pass the name of the database file to be open (or created if it doesn't exist):

```
 MySQLiteConnection := TSQLiteNativeConnectionAdapter.Create('C:
\Database\SQLite\MyDatabase.sdb');
```

TSQLiteNativeConnectionAdapter class also has two additional methods that you can use to manually disable or enable foreign keys in SQLite (foreign keys are enforced at connection level, not database level in SQLite!).

```
  procedure EnableForeignKeys;
  procedure DisableForeignKeys;
```

So if you want to use SQLite with foreign keys, do this to retrieve your connection:

```
var
  SQLiteAdapter: TSQLiteNativeConnectionAdapter;
  MySQLiteConnection: IDBConnection;
begin
  SQLiteAdapter := TSQLiteNativeConnectionAdapter.Create('C:
\Database\SQLite\MyDatabase.sdb');
  SQLiteAdapter.EnableForeignKeys;
  MySQLiteConnection := SQLiteAdapter;
  // Use MySQLiteConnection interface from now on
```

dbGo (ADO) Adapter
Currently dbGo (ADO) is only officially supported when connecting to Microsoft SQL Server databases. Drivers for other databases might work but were not tested.

3.3 SQL Dialects

To save and manipulate objects in the database, TMS Aurelius internally build and execute SQL statements. The SQL statements are automatically adjusted to use the correct dialect, according to the database server being used by the programmer.

When you create an IDBConnection interface using a component adapter, usually the

adapter will automatically specify to Aurelius the SQL dialect to use. For example, if you are using dbExpress components, the adapter will look to the DriverName property and tell which db server you are using, and then define the correct SQL dialect name that should be used.

However, the SQL dialect must be explicity registered in the global settings for Aurelius. This is by design so you don't need to load units for SQL dialects you won't use. To register an SQL dialect, just use a unit named Aurelius.SQL.XXX where XXX is the name of the SQL dialect. The following table lists all current SQL dialects supported, the exact string identifier, and the unit you must add to your project in order for the dialect to be registered.

SQL dialect	String identifier	Declared in unit	Database Web Site
Absolute Database	AbsoluteDB	Aurelius.Sql.AbsoluteDB	http://www.componentace.com
DB2	DB2	Aurelius.Sql.DB2	http://www.ibm.com
ElevateDB	ElevateDB	Aurelius.Sql.ElevateDB	http://www.elevatesoftware.com
Firebird	Firebird	Aurelius.Sql.Firebird	http://www.firebirdsql.org
Interbase	Interbase	Aurelius.Sql.Interbase	http://www.embarcadero.com
Microsoft SQL Server	MSSQL	Aurelius.Sql.MSSQL	http://www.microsoft.com/sqlserver
MySQL	MySQL	Aurelius.Sql.MySql	http://www.mysql.com
NexusDB	NexusDB	Aurelius.Sql.NexusDB	http://www.nexusdb.com
Oracle	Oracle	Aurelius.Sql.Oracle	http://www.oracle.com
PostgreSQL	PostgreSQL	Aurelius.Sql.PostgreSQL	http://www.postgresql.org
SQLite	SQLite	Aurelius.Sql.SQLite	http://www.sqlite.org

Note that in some situations, the adapter is not able to identify the correct dialect. It can happen, for example, when you are using ODBC or just another data access component in which is not possible to tell which database server the component is trying to access. In this case, when creating the adapter, you can use an overloaded constructor that allows you to specify the SQL dialect to use:

```
MyConnection := TDBExpressConnectionAdapter.Create(SQLConnection1,
'MSSQL', False);
```

3.4 Schema Importers

To be able to update and validate database schema, Aurelius needs to perform reverse engineering in the database. This is accomplished by using schema importers that execute specific SQL statements to retrieve the database schema, depending on the database

server being used. To find the correct importer, Aurelius searches for a list of registered schema importers, using the same SQL Dialect used by the current connection. So, for example, if the current SQL Dialect is "MySQL", Aurelius will try to find a schema importer named "MySQL".

By default, no schema importers are registered. You must be explicity register a schema importer in the global settings for Aurelius. This is by design so you don't need to load units for schema importers you won't use. To register an schema importer, just use a unit named Aurelius.Schema.XXX where XXX is the name of the SQL dialect associated with the schema importer. The following table lists all current schema importers supported, the exact string identifier, and the unit you must add to your project in order for the dialect to be registered.

Schema Importer for	String identifier (associated SQL Dialect	Declared in unit
Absolute Database	AbsoluteDB	Aurelius.Schema.AbsoluteDB
DB2	DB2	Aurelius.Schema.DB2
ElevateDB	ElevateDB	Aurelius.Schema.ElevateDB
Firebird	Firebird	Aurelius.Schema.Firebird
Interbase	Interbase	Aurelius.Schema.Interbase
Microsoft SQL Server	MSSQL	Aurelius.Schema.MSSQL
MySQL	MySQL	Aurelius.Schema.MySql
NexusDB	NexusDB	Aurelius.Schema.NexusDB
Oracle	Oracle	Aurelius.Schema.Oracle
PostgreSQL	PostgreSQL	Aurelius.Schema.PostgreSQL
SQLite	SQLite	Aurelius.Schema.SQLite

3.5 Components and Databases Homologation

The following table presents which data-access component can be used to access each relational database server. Note that some components can access more databases than what's described here (especially dbGo (ADO) which can access several databases through OleDB drivers). However, the table below shows what has been tested and is officially supported by TMS Aurelius.

	Native	Absolute	AnyDac	dbExpress	dbGo	DOA	ElevateDB	FIBPlus	IBO	IBX	Nexus	SQLDirect	UniDac	UIB
AbsoluteDB		x												
DB2			x		x							x	x	
ElevateDB							x							
Firebird			x	x				x	x				x	x
Interbase			x	x					x	x		x	x	x

MS SQL Server			x	x	x							x	x	
MySQL			x	x								x	x	
NexusDB											x		x	
Oracle			x	x		x						x	x	
PostgreSQL			x										x	
SQLite	x		x										x	

Database versions used for homologation are listed below. TMS Aurelius tries to use no syntax or features of an specific version, its internal code uses the most generic approach as possible. Thus, other versions will most likely work, especially newer ones, but the list below is provided for your reference.

Database	Version
AbsoluteDB	7.05
DB2	9.7.500
ElevateDB	2.08
Firebird	2.5.1
Interbase	XE (10.0.3)
MS SQL Server	2008 R2 (10.50.1600)
MySQL	5.5.17 (Server) 5.1.60 (Client)
NexusDB	3.0900
Oracle	10g Express (10.2.0.1.0)
PostgreSQL	9.1
SQLite	3.7.9

Analog to databases above, in table below we list data-access components used for homologation and respective versions. Newer versions should work with not problems.

Component Library	Versions
AbsoluteDB	7.05
AnyDac	5.0.3.1917
dbExpress	16.0
dbGo	Delphi 2010 up to XE3
Direct Oracle Access	4.1.3.3
ElevateDB	2.08
FIBPlus	7.2
IBObjects	4.9.14
IBX	Delphi 2010 up to XE2

NexusDB	3.0900, 3.1003
SQL-Direct	6.3
UniDac	4.1.4 4.1.5 (for NexusDB)
Unified Interbase (UIB)	2.5 revision 428 (01-Feb-2013)

3.6 Database Manager - Creating/Updating Schema

If you have an existing database, you can use Aurelius on it. You can map your existing or new classes to the tables and fields of existing databases, and that's it. But for new applications, you might consider just modeling the classes, and let Aurelius build/update the database structure for you, creating all database objects needed to persist the objects. To do that, just create a TDatabaseManager object (declared in unit Aurelius.Engine.DatabaseManager) the same way you create a TObjectManager, and use one of the methods available to manager the schema (database structure). Common usage is as following:

```
uses
  Aurelius.Engine.DatabaseManager;
{...}
var
  DBManager: TDatabaseManager;
begin
  DBManager := TDatabaseManager.Create(MyConnection); // use default
mapping explorer
  // operate on database schema using DBManager
  DBManager.Free;
end;
```

alternatively, you can also pass a TMappingExplorer instance, which holds a custom mapping setup.

```
DBManager := TDatabaseManager.Create(MyConnection, MyMappingExplorer);
```

The following topics explain how to use the database manager object.

3.6.1 Creating New Schema

You can create a new schema from an empty database using method BuildDatabase:

```
uses
```

```
  Aurelius.Engine.DatabaseManager;
{...}
var
  DBManager: TDatabaseManager;
begin
  DBManager := TDatabaseManager.Create(MyConnection);
  DBManager.BuildDatabase;
  DBManager.Free;
end;
```

This method will execute all SQL statements that create the whole database structure needed to persist the mapped entity classes. It **does not take into account** the existing database schema, so if tables already exist, an "object already exists" error will happen in database server when executing the statement. You can alternatively just generate the SQL script without executing it.

Even though this method does not perform any reverse engineering to check existing database structure, a schema validation result is available. Results are provided as if the existing database is empty.

3.6.2 Updating Existing Schema

You can update the existing database structure using method UpdateDatabase:

```
uses
  Aurelius.Engine.DatabaseManager;
{...}
var
  DBManager: TDatabaseManager;
begin
  DBManager := TDatabaseManager.Create(MyConnection);
  DBManager.UpdateDatabase;
  DBManager.Free;
end;
```

This method will:

1. Perform a schema validation, which consists of:
a) Execute SQL statements to perform a reverse engineering in the database, retrieving the existing database schema (*).
b) Compare the existing schema with the target schema (all database objects - table, columns, etc. - need to persist the mapped entity classes.
c) Provide info about the differences between the two schema (see schema validation for details).
d) Generate the SQL Script needed to update the database schema

2. Execute the SQL Script in the database, unless command execution is disabled (see Generating SQL Script)

(*) **Note:** for Aurelius to properly import database schema, you need to register a schema importer according to the database server you are connecting to. For example, to import

MySQL schema, just use the unit "Aurelius.Schema.MySQL" anywhere in your project.

If command execution is disabled, this method behaves exactly as the ValidateDatabase method.

Since this method performs on a database that has existing object and **data**, it has some limitations. First, if you are unsure of the effects of schema update, it's strongly recommended that you check schema validation results before updating. Errors might occur when updating the schema, for example, if new schema requires a foreign key creating but existing data doesn't fit into this new constraint. See schema validation for a list of current valid operations and limitations.

Note that UpdateDatabase is a **non-destructive** method. This means that even if the validation reports that a data-holding object (table or column) needs to be dropped, the SQL statement for it will **not be performed.**

3.6.3 Dropping Existing Schema

You can drop the whole database structure from an existing database using method DestroyDatabase:

```
uses
  Aurelius.Engine.DatabaseManager;
{...}
var
  DBManager: TDatabaseManager;
begin
  DBManager := TDatabaseManager.Create(MyConnection);
  DBManager.DestroyDatabase;
  DBManager.Free;
end;
```

This method will execute all SQL statements that destroy the whole database structure needed to persist the mapped entity classes. It **does not take into account** the existing database schema, so if tables were already dropped, an "object does not exist" error will happen in database server when executing the statement. You can alternatively just generate the SQL script without executing it.

Even though this method does not perform any reverse engineering to check existing database structure, a schema validation result is available. Results are provided as if the existing database is complete, with all objects, and target database structure is empty.

3.6.4 Schema Validation

Schema validation is a process that gives you the differences between the existing database schema and the needed schema to make the current application to work. You can validate the existing database structure using method ValidateDatabase. The method returns true if there are no differences in that comparison (meaning that the existing database structure has all database objects needed by the application):

```
uses
  Aurelius.Engine.DatabaseManager,
  Aurelius.Schema.Messages;
{...}
```

```
var
  DBManager: TDatabaseManager;
  SchemaMessage: TSchemaMessage;
begin
  DBManager := TDatabaseManager.Create(MyConnection);
  if DBManager.ValidateDatabase then
    WriteLn('Database strucuture is valid.')
  else
  begin
    WriteLn(Format('Invalid database structure. %d Errors, %d Warnings, %
d Actions',
      [DBManager.ErrorCount, DBManager.WarningCount,
DBManager.ActionCount]));
    for SchemaMessage in DBManager.Warnings do
      WriteLn('Warning: ' + SchemaMessage.Text);
    for SchemaMessage in DBManager.Errors do
      WriteLn('Error: ' + SchemaMessage.Text);
    for SchemaMessage in DBManager.Actions do
      WriteLn('Action: ' + SchemaMessage.Text);
  end;
  DBManager.Free;
end;
```

This method will:

a) Execute SQL statements to perform a reverse engineering in the database, retrieving the existing database schema (*).
b) Compare the existing schema with the target schema (all database objects - table, columns, etc. - need to persist the mapped entity classes.
c) Provide info about the differences between the two schema (see schema validation for details).
d) Generate the SQL Script needed to update the database schema

(*) **Note:** for Aurelius to properly import database schema, you need to register a schema importer according to the database server you are connecting to. For example, to import MySQL schema, just use the unit "Aurelius.Schema.MySQL" anywhere in your project.

If command execution is disabled, this method behaves exactly as the UpdateDatabase method.

The comparison result is provided through properties Actions, Warnings and Errors and also ActionCount, WarningCount and ErrorCount, defined as following:

```
  property Actions: TEnumerable<TSchemaAction>;
  property Warnings: TEnumerable<TSchemaWarning>;
  property Errors: TEnumerable<TSchemaError>;
  property ActionCount: integer;
  property WarningCount: integer;
  property ErrorCount: integer;
```

TSchemaAction, TSchemaWarning and TSchemaError classes inherit from TSchemaMessage class, which just has a public Text property with the information about the difference. The concept of each message type (actio, warning, error) is described as

follows:

Actions
Actions are reported differences between the two schemas which associated SQL update statements **can be safely executed** by the database manager. Examples of differences that generate actions:

- A new table
- A new nullable column in an existing table
- A new sequence
- Foreign key removal (if supported by database)
- Unique key removal (if supported by database)

Warnings
Warnings are reported differences between the two schemas which associated SQL update statements **can be executed** by the database manager, but it **might cause runtime errors depending on the existing database data.** Examples of differences that generate actions:

- A new not null column in an existing table (to be safe, when updating existing schema, try to always create new columns as nullable)
- A new foreign key (usually you will create a new association, which will generate actions for new foreign key **and** new columns, which will not cause problem, unless the association is required. It's a warning if supported by database)

Errors
Errors are reported differences between the two schemas which associated SQL update statements **cannot be executed** by the database manager. This means that updating the schema will not make those differences disappear, and you would have to change the schema manually. The fact it is reported as "Error" **does not mean the application will not work**. It just means that the manager cannot update such differences. Examples of differences that generate errors:

- Column data type change
- Column Null/Not Null constraint change
- Column length, precision or scale change
- A new foreign key (if database does not support such statement)
- Foreign key removal (if database does not support such statement)
- Unique key removal (if database does not support such statement)
- Changes in primary key (id fields)
- Column removal
- Table removal
- Sequence removal
- A new unique key

3.6.5 Generating SQL Script

All TDatabaseManager methods that perform some operation in the database schema generate an SQL script, available in the **SQLStatements property**. Most methods also **execute** such statements (like BuildDatabase, UpdateDatabase and DropDatabase). Some methods do not execute, like ValidateDatabase. But in all cases, the associated SQL script is available.

In TDatabaseManager you have the option to disable execution of SQL statements. This way you have the freedom to execute the statements as you want, using you our error handling system, your own graphical user interface to execute them, etc. To do that, just set **SQLExecutionEnabled property to false**. Examples:

```
uses
  Aurelius.Engine.DatabaseManager;
{...}
var
  DBManager: TDatabaseManager;

  procedure OutputSQLScript;
  var
    SQLStatement: string;
  begin
    for SQLStatement in DBManager.SQLStatements do
      WriteLn(SQLStatement);
  end;

begin
  DBManager := TDatabaseManager.Create(MyConnection);
  DBManager.SQLExecutionEnabled := false;

  // Output an SQL Script to build a new database
  DBManager.BuildDatabase;
  OutputSQLScript;

  // Output an SQL to drop the full database
  DBManager.DropDatabase;
  OutputSQLScript;

  // Output an SQL script to update the existing database
  DBManager.UpdateDatabase;
  OutputSQLScript;

  DBManager.Free;
end;
```

Note that when SQLExecutionEnabled property is false, calling UpdateDatabase is equivalent to calling ValidateDatabase, so this code:

```
  // Output an SQL script to update the existing database
  DBManager.SQLExecutionEnabled := false;
  DBManager.UpdateDatabase;
  OutputSQLScript;
```

Could also be written just as:

```
  // Output an SQL script to update the existing database
  // Regardless of value of SQLExecutionEnabled property
  DBManager.ValidateDatabase;
  OutputSQLScript;
```

Chapter

Mapping

4 Mapping

This chapter provides you information about how to map your classes to the database. While a mapping can be made so simple using a single automapping attribute, it can be fully configurable and might need lots of concepts to be done the way you need. Several mapping attributes are available, you can also create your classes using special types like Nullable<T> and TBlob, and so on. The topics below describe all the mapping mechanism in TMS Aurelius.

4.1 Attributes

Object-Relational Mapping in Aurelius is done by using attributes. With this approach you can do your mapping directly when coding the classes, and by browsing the source code you can easily tell how the class is being mapped to the database.

Basically you just add attributes to the class itself, or to a field or property:

```
[Table('Customer')]
TMyCustomer = class
private
  [Column('Customer_Name')]
  FCustomerName: string;
...
```

For column and associations mapping Aurelius accepts mapping attributes in either class field or class property (but not both of course). We recommend using mapping attributes in fields whenever it's possible, for several reasons:

1. Attributes are kept in private section of your class, leaving the public section clean and easily readable
2. Fields represent better the current state of the object. Properties can have getter and setters based on other data that it's not exactly the object state for persistance.
3. Some Aurelius features are better suited for fields. For example, lazy-loaded associations requires the use of a Proxy type, which makes more sense to be uses in fields (although you can use it in properties)

Still, there are situations where creating mapping attributes in properties are interesting, when for example you want to save the result of a runtime calculation in database.

Available attributes (declared in unit Aurelius.Mapping.Attributes):

Basic Mapping
Entity
Id
Table
Column
Sequence
UniqueKey
Enumeration

Association Mapping
Association
JoinColumn

Many-Valued Association Mapping
ManyValuedAssociation
ForeignJoinColumn

Inheritance Mapping
Inheritance
DiscriminatorColumn
DiscriminatorValue
PrimaryJoinColumn

Automapping
Automapping
Transient

Other attributes
Description

4.1.1 Entity

Indicates that the class is an entity class, which means it can be persisted.

Level: Class Attribute

Description
Every class that you want to be persisted in database must have this attribute. It's also used by Aurelius for automatic class registration. When automatic registration is active in global configuration, every class marked with Entity attribute will be automatically registered as an entity class.

Constructor
```
constructor Create;
```

Parametes
None.

Usage

```
[Entity]
TCustomer = class(TObject)
```

4.1.2 Id

Specify the Identifier of the class

Level: Class Attribute

Description
Every object must be uniquely identified by Aurelius so that it can properly save and manage it. The concept is similar to a primary key in database. This attribute allows you to specify which field (or property) in the class will be used to uniquely identify the class. The value of that field/property must be unique for every object, and you can specify how that value will be generated for each object.
In addition, if you are creating the database structure from the mapped classes, Aurelius will create a primary key in the database corresponding to the field/column mapping.
If you are using inheritance, you must only declare the Id attribute in the base class of the hierarchy (the ancestor class). The inherited child classes can't have their own Id attribute.

For composite id's, specify as many Id attributes as you need to build the composite identifier.

Constructor

```
constructor Create(AMemberName: string; AGenerator: TIdGenerator);
```

Parameters

AMemberName	Contains the name of field or property that identifies the object
AGenerator	Indicates how the Id value will be generated. Valid values are (prefixed by TIdGenerator): None — Id value will not be automatically generated. Your application must assign a value to it and be sure it's unique IdentityOrSequence — Aurelius will ask the database to generate a new Id. If the database supports sequences and a sequence is defined, then Aurelius will use the sequence to generate the value. Otherwise, it will use identity (auto-numerated) fields. If no sequence is defined and database doesn't support identity fields, an exception will be raised. The name of the sequence to be created and used by Aurelius can be defined using the Sequence attribute. The type of the property that identifies the entity should be integer. Guid — Aurelius will generate a GUID (Globally Unique Identifier) value as the entity identifier. The type of the property that identifies the entity should be TGuid. Uuid38 — Aurelius will generate a 38-length UUID (Universally Unique

		Identifier) value as the entity identifier. An UUID is just a string representation of a GUID value, with the format "{550e8400-e29b-41d4-a716-446655440000}" (with hifens and curly brackets). The type of the property that identifies the entity should be string (with a minimum length of 38 characters)
	Uuid36	Aurelius will generate a 36-length UUID (Universally Unique Identifier) value as the entity identifier. An UUID is just a string representation of a GUID value, with the format "550e8400-e29b-41d4-a716-446655440000" (with hifens but no curly brackets). The type of the property that identifies the entity should be string (with a minimum length of 36 characters)
	Uuid32	Aurelius will generate a 32-length UUID (Universally Unique Identifier) value as the entity identifier. An UUID is just a string representation of a GUID value, with the format "550e8400e29b41d4a716446655440000" (no hifens and no curly brackets). The type of the property that identifies the entity should be string (with a minimum length of 32 characters)
	For composite id's this value is ignored and None is used.	

Usage

```
[Id('FId', TIdGenerator.IdentityOrSequence)]
TCustomer = class(TObject)
private
  [Column('CUSTOMER_ID')]
  FId: integer;
```

4.1.3 Table

Specify the database table where the objects will be saved to.

Level: Class Attribute

Description
Use the Table attribute to map the class to a database table. Every object instance saved will be a record in that table.
If you are using inheritance with single table strategy, you must use the Table attribute in the ancestor class only, since all classes will be saved in the same table.
If you are using inheritance with joined tables strategy, you must use Table attribute in all classes, since every class will be saved in a different table.

Constructor

```
constructor Create(Name: string); overload;
constructor Create(Name, Schema: string); overload;
```

Parameters

Name	The name of the table in database
Schema	Optionally you can specify the schema of the database

Usage

```
[Table('Customers')]
TCustomer = class(TObject)
private

[Table('Orders', 'dbo')]
TOrder = class(TObject)
private
```

4.1.4 Column

Specify the table column where the field/property value will be saved to.

Level: Field/Property Attribute

Description
Use Column attribute to map a field/property to a table column in the database. When saving an object, Aurelius will save and load the field/property value in the specified table column. Only fields/properties mapped using a Column attribute will be saved in the database (unless class is automapped using Automapping attribute).
Aurelius will define the table column data type automatically based on type of field/property being mapped.

Constructor

```
constructor Create(Name: string); overload;
constructor Create(Name: string; Properties: TColumnProps); overload;
constructor Create(Name: string; Properties: TColumnProps; Length:
Integer); overload;
constructor Create(Name: string; Properties: TColumnProps; Precision,
Scale: Integer); overload;
```

Parameters

Name	Contains the name of table column in the database where the field/property will be mapped to

Properties	A set containing zero or more options for the column. TColumnProps and TColumnProp are declared as follow:
	
	`TColumnProp = (Unique, Required, NoInsert, NoUpdate);`
	`TColumnProps = set of TColumnProp;`
	
	Unique — Values of this column must be unique. Aurelius will create an unique key (index) in the database to ensure unique values for this column. The index name will be the same as the column name. If you want to define a different name, do not set this flag and use UniqueKey attribute instead.

	Required	Column must be NOT NULL. Values are required for this field/property
	NoInsert	When inserting a record in the database, do not include this column in the INSERT command. The value of this field/property will not be saved in the database in INSERT commands. Note that for Id fields using identity (autogenerated), Aurelius will automatically not include the field in the INSERT statement, regardless if cpDontInsert is specified or not
	NoUpdate	When updating a record in the database, do not include this column in the UPDATE command. The value of this field/property will not be saved in the database in UPDATE commands. This flag is usually used for Id fields which once inserted should not be changed anymore
	Lazy	Used for blob fields only. Indicates that lazy-loading will be used for the blob, i.e., the content of the blob will only be retrieved from the database when needed. If the property is not of type TBlob, this option will be ignored.
Length	Used for string field/property. It's the maximum length of the table column. Usually this is mapped to the VARCHAR type, i.e., if Length is 30, the data type of table column will be VARCHAR(30). It it's not specified, Aurelius will use the default length for string data types.	
Precision, Scale	Used for numeric field/property. Specifies the precision and scale of numeric columns in the database table. If not specified, default values will be used.	

Usage

```
[Column('MEDIA_NAME', [TColumnProp.Required], 100)]
property MediaName: string read FMediaName write FMediaName;

[Column('DURATION', [])]
property Duration: Nullable<integer> read FDuration write FDuration;
```

4.1.5 Association

Specifies a many-to-one association (relationship).

Level: Field/Property Attribute

Description
Use Association attribute to indicate that the field/property represents a many-to-one association with another class. For example, if you have property Customer of type TCustomer, it means that your object is associated with one (and only one) customer. Associations can only be defined for fields and properties of class types, and the associated class must also be an Entity class, so you can have a relationship between one class and another (between tables, at database level).
You must always use Association attribute together with JoinColumn attribute. While the

former is used to define generic, class-level meta-information about the association, the latter is used to define database-level relationships (fields that will be foreign keys)

Constructor

```
constructor Create; overload;
constructor Create(AProperties: TAssociationProps); overload;
constructor Create(AProperties: TAssociationProps; Cascade:
TCascadeTypes); overload;
```

Parameters

AProperties	Specifies some general properties for the association. Valid values are:
	
	`TAssociationProp = (Lazy, Required);`
	`TAssociationProps = set of TAssociationProp;`
	
	Lazy — The associated object is not loaded together with the current object. Lazy-Loading is used. In a SELECT operation, Aurelius will only retrieve the Id of the associated object. The object will only be loaded when the application effectively needs it (e.g., when user references property MyObject.AssociatedObject). When it happens, Aurelius will perform another SELECT in the database just to retrieve the associated object data. Only at this point the object is instantiated and data is filled.
	If Lazy is not specified, the default behavior is eager-mode loading. It means that when the object is loaded, the associated object is also fully loaded. Aurelius will perform a INNER (or LEFT) JOIN to the related tables, fetch all needed fields, create an instance of the associated object and set all its properties. This is the default value.
	
	Required — Associated object is required. When Aurelius executes a SELECT statement to load the object, it will use an INNER JOIN to retrieve data for the associated object. When setting this flag it's recommended to set the column as required in the JoinColumn attribute.
	If Required not specified, then is assumes by default that association is optional. It means that associated object is not required. When Aurelius executes a SELECT statement to load the object, it will use a LEFT JOIN to retrieve data for the associated object.

Cascade	Defines how Aurelius will behave on the association when the container object is saved, deleted or updated.
	
	`TCascadeType = (SaveUpdate, Merge, Remove, Refresh);`
	`TCascadeTypes = set of TCascadeType;`
	`CascadeTypeAll = [TCascadeType.SaveUpdate, TCascadeType.Merge, TCascadeType.Remove];`
	
	SaveUpdate — When object is saved (inserted), or updated, the associated object will be automatically saved/updated. The associated object is actually saved before the container object, because

		the Id of associated object might be needed to save the container object.
	Merge	When object is merged, the associated object will also be merged.
	Remove	When object is removed from database, the associated object will also be removed.
	Refresh	Reserved, not used for now.

Usage

```
[Association([], CascadeTypeAll)]
[JoinColumn('ID_SONG_FORMAT', [])]
property SongFormat: TSongFormat read FSongFormat write FSongFormat;

[Association([TAssociationProp.Lazy], [TCascadeType.SaveUpdate])]
[JoinColumn('ID_ARTIST', [])]
FArtist: Proxy<TArtist>;
```

Note

In the previous example, the Proxy<TArtist> type is used because association was declared as lazy (see Associations and Lazy-Loading). Alternatively you can declare FArtist field just as TArtist, and in this case association will not be lazy-loaded.

4.1.6 JoinColumn

Specifies the table column used as foreign key for one association.

Level: Field/Property Attribute

Description
Use JoinColumn attribute to map a field/property to a table column in the database. The field/property must also have an Association attribute defined for it.
The table column defined by JoinColumn will be created as a foreign key to the referenced association. By default, the relationship created by Aurelius will reference the Id of the associated object. But you can reference another value in the object, as long as the value is an unique value.
The data type of the table column defined by JoinColumn will be the same as the data type of the referenced column in the associated table.

When the association is a class with composite Id's, specify as many JoinColumn attributes as the number of columns in the primary key of association class. For example, if the associated class has three table columns in the primary key, you must specify three JoinColumn attributes, one for each column.

Constructor

```
constructor Create(Name: string); overload;
constructor Create(Name: string; Properties: TColumnProperties);
overload;
```

```
constructor Create(Name: string; Properties: TColumnProperties;
ReferencedColumnName: string); overload;
```

Parameters

Name	Contains the name of table column in the database used to hold the foreign key.

Properties	A set containing zero or more options for the column. TColumnProps and TColumnProp are declared as follow:
	
	`TColumnProp = (Unique, Required, NoInsert, NoUpdate);`
	`TColumnProps = set of TColumnProp;`
	
	Unique — Values of this column must be unique. Aurelius will create an unique key (index) in the database to ensure unique values for this column. In practice, if this flag is set the relationship will become a one-to-one relationship
	
	Required — Column must be NOT NULL. Values are required for this field/property. This flag must be set together with the orRequired flag in Association attribute.
	
	NoInsert — When inserting a record in the database, do not include this column in the INSERT command. The value of this field/property will not be saved in the database in INSERT commands.
	
	NoUpdate — When updating a record in the database, do not include this column in the UPDATE command. The value of this field/property will not be saved in the database in UPDATE commands.
	
	Lazy — Not used. This option is only used in Column attribute.

ReferencedColumnName	Indicates the column name in the associated table that will be referenced as foreign key. The referenced column must be unique in the associated table. This parameter is optional, if it's not specified (and usually it won't), the name of Id field will be used - in other words, the primary key of the associated table will be referenced by the foreign key.

Usage

```
[Association]
[JoinColumn('ID_SONG_FORMAT', [])]
property SongFormat: TSongFormat read FSongFormat write FSongFormat;
```

```
[Association([TAssociationProp.Lazy], [])]
```

```
[JoinColumn('ID_ARTIST', [])]
FArtist: Proxy<TArtist>;
```

Note

In the previous example, the Proxy<TArtist> type is used because association was declared as lazy (see Associations and Lazy-Loading). Alternatively you can declare FArtist field just as TArtist, and in this case association will not be lazy-loaded.

4.1.7 ManyValuedAssociation

Specifies an one-to-many association (relationship), or in other words, a collection of objects.

Level: Field/Property Attribute

Description
Use ManyValuedAssociation attribute to indicate that the field/property represents a one-to-many association - a collection of objects of the same class. For example, if you have property Addresses of type TList<TCustomer>, it means that each object in collection is associated with the container object. Many-valued associations can only be defined for fields and properties of type TList<*class*>, and the associated *class* must also be an Entity class, so you can have a relationship between one class and another (between tables, at database level).

Defining a collection of child objects like this will require that the table holding child objects records will have a foreign key column referencing the container object. This can be done in two ways.
1. Use ForeignJoinColumn attribute to define a foreign key in the child object class.
2. Create an Association in the child object class and then use *MappedBy* parameter to indicate the field/property that holds the association. This will become a bidirectional association, since you have the child object referencing the parent object though an Association, and the parent object holding a collection of child objects through a ManyValuedAssociation.

Constructor

```
constructor Create; overload;
constructor Create(AProperties: TAssociationProps); overload;
constructor Create(AProperties: TAssociationProps; Cascade:
TCascadeTypes); overload;
constructor Create(AProperties: TAssociationProps; Cascade:
TCascadeTypes; MappedBy: string); overload;
```

Parameters

FetchType	Specifies some general properties for the association. Valid values are:
	
	`TAssociationProp = (Lazy, Required);`
	`TAssociationProps = set of TAssociationProp;`
	
	Lazy — The associated object is not loaded together with the current object. Lazy-Loading is used. In a SELECT operation, Aurelius will only retrieve the Id of the associated object. The object will only

	be loaded when the application effectively needs it (e.g., when user references property MyObject.AssociatedObject). When it happens, Aurelius will perform another SELECT in the database just to retrieve the associated object data. Only at this point the object is instantiated and data is filled. If Lazy is not specified, the default behavior is eager-mode loading. It means that when the object is loaded, the associated object is also fully loaded. Aurelius will perform a INNER (or LEFT) JOIN to the related tables, fetch all needed fields, create an instance of the associated object and set all its properties. This is the default value. Required This option is ignored in Many-valued Associations

Cascade	Defines how Aurelius will behave on the association list when the container object is saved, deleted or updated.
	
	`TCascadeType = (SaveUpdate, Merge, Remove, Refresh);`
	`TCascadeTypes = set of TCascadeType;`
	`CascadeTypeAll = [TCascadeType.SaveUpdate, TCascadeType.Merge, TCascadeType.Remove];`
	
	SaveUpdaet When object is save (inserted) or updated, the associated object list will be automatically saved. First the parent object is saved, then all objects in the collection are also saved.
	
	Merge When object is merged, all the associated objects in the object list are also merged.
	
	Remove When object is removed from database, all objects in the list are also removed.
	
	Refresh Reserved, not used for now.

MappedBy	This parameter must be used when the association is bidirectional, i.e., the associated class referenced in the list has also an Association to the object containing the list, see Description above. This parameter must contain the name of field or property, in the child object class, that holds an Association referencing the container object.

Usage

Example using *MappedBy* parameter:

```
TMediaFile = class
private
  [Association([TAssociationProp.Lazy], [])]
  [JoinColumn('ID_ALBUM', [])]
  FAlbum: Proxy<TAlbum>;

TAlbum = class
public
  [ManyValuedAssociation([], CascadeTypeAll, 'FAlbum')]
  property MediaFiles: TList<TMediaFile> read FMediaFiles write
FMediaFiles;
```

Example using ForeignJoinColumn attribute (in this example, TTC_InvoiceItem class does not have an association to TTC_Invoice class, so "INVOICE_ID" field will be created in InvoiceItem table):

```
TTC_Invoice = class
private
  [ManyValuedAssociation([], CascadeTypeAll)]
  [ForeignJoinColumn('INVOICE_ID', [TColumnProp.Required])]
  FItems: TList<TTC_InvoiceItem>;
```

Note

In the previous example, the Proxy<TAlbum> type is used because association was declared as lazy (see Associations and Lazy-Loading). Alternatively you can declare FAlbum field just as TAlbum, and in this case association will not be lazy-loaded.

4.1.8 ForeignJoinColumn

Specifies the table column used as foreign key in the child object, for a many-valued-association.

Level: Field/Property Attribute

Description
Use ForeignJoinColumn attribute to map a field/property to a table column in the database. The field/property must also have an ManyValuedAssociation attribute defined for it.
The table column defined by ForeignJoinColumn will be created as a foreign key to the referenced association. Note that the column will be created in the **child table**, and it will reference the **parent table,** i.e, the "container" of the object list.
By default, the relationship created by Aurelius will reference the Id of the associated object. But you can reference another value in the object, as long as the value is an unique value.
The data type of the table column defined by ForeignJoinColumn will be the same as the data type of the referenced column in the associated table.
This attribute must only be used if the ManyValuedAssociation is unidirectional. If it's bidirectional, you should not use it, and just the MappedBy parameter when declaring the ManyValuedAssociation attribute.

When the association is a class with composite Id's, specify as many ForeignJoinColumn attributes as the number of columns in the primary key of association class. For example,

if the associated class has three table columns in the primary key, you must specify three ForeignJoinColumn attributes, one for each column.

Constructor

```
constructor Create(Name: string); overload;
constructor Create(Name: string; Properties: TColumnProperties);
overload;
constructor Create(Name: string; Properties: TColumnProperties;
ReferencedColumnName: string); overload;
```

Parameters

Name	Contains the name of table column in the database used to hold the foreign key.

Properties	A set containing zero or more options for the column. TColumnProps and TColumnProp are declared as follow:
	
	`TColumnProp = (Unique, Required, NoInsert, NoUpdate);`
	`TColumnProps = set of TColumnProp;`
	
	Unique — Values of this column must be unique. Aurelius will create an unique key (index) in the database to ensure unique values for this column.
	
	Required — Column must be NOT NULL. Values are required for this field/property.
	
	NoInsert — When inserting a record in the database, do not include this column in the INSERT command. The value of this field/property will not be saved in the database in INSERT commands.
	
	NoUpdate — When updating a record in the database, do not include this column in the UPDATE command. The value of this field/property will not be saved in the database in UPDATE commands.
	
	Lazy — Not used. This option is only used in Column attribute.

ReferencedColumnName	Indicates the column name in the associated table that will be referenced as foreign key. The referenced column must be unique in the associated table. This parameter is optional, if it's not specified (and usually it won't), the name of Id field will be used - in other words, the primary key of the associated table will be referenced by the foreign key.

Usage

```
TTC_Invoice = class
private
  [ManyValuedAssociation([], CascadeTypeAll)]
  [ForeignJoinColumn('INVOICE_ID', [TColumnProp.Required])]
  FItems: TList<TTC_InvoiceItem>;
```

4.1.9 Inheritance

Identifies the class as the ancestor for a hierarchy of entity classes.

Level: Class Attribute

Description
Use Inheritance attribute to allow persistence of the current class and all its descendants (if they are marked with Entity attribute).
If you have a class hierarchy and want Aurelius to save all of those classes, you must add the Inheritance attribute to the top level (parent) class of all the hierarchy in order to use a specific inheritance strategy. If you are using single table strategy, you also need to define a DiscriminatorColumn attribute in the base class, and a DiscriminatorValue attribute in each descendant class. If you are using joined tables strategy, you need to define a PrimaryJoinColumn attribute and a Table attribute in each descendant class.

Constructor

```
constructor Create(Strategy: TInheritanceStrategy);
```

Parameters

Strategy	Specifies the inheritance strategy to be used in the class hierarchy. Valid values are (prefixed by TInheritanceStrategy): SingleTable — Use single table strategy for the class hierarchy. You must also define a DiscriminatorColumn attribute in the class and a DiscriminatorValue attribute in each descendant class. JoinedTables — Use joined tables strategy for the class hierarchy. In this strategy for each descendant class you must define a PrimaryJoinColumn and Table attribute.

Usage

```
[Inheritance(TInheritanceStrategy.SingleTable)]
[DiscriminatorColumn('MEDIA_TYPE', TDiscriminatorType.dtString)]
TMediaFile = class
```

4.1.10 DiscriminatorColumn

Specifies the column table to be used as class discriminator in a single table inheritance strategy.

Level: Class Attribute

Description
Use DiscriminatorColumn attribute to specify the column in the table used as class discriminator. When you use Inheritance attribute and set strategy to single table, you must also define this attribute. In single table strategy, all classes are saved in the same table, and the value of discriminator column is the way Aurelius use to tell the class representing each record in the table. For example, if you have both classes TCar and TMotorcycle inheriting from TVehicle and all classes being saved in the same table, when Aurelius reads a record it must tell if it represents a TCar or TMotorcycle. It does that using the value specified in the discriminator column. Each descending class must declare the attribute DiscriminatorValue that will define what is the value to be saved in the discriminator column that will represent the specified class.

Constructor

```
constructor Create(Name: string; DiscriminatorType: TDiscriminatorType);
overload;
constructor Create(Name: string; DiscriminatorType: TDiscriminatorType;
Length: Integer); overload;
```

Parameters

Name	The name of the table column that will hold discriminator values which will identify the class. This column will be created by Aurelius if you create the database
DiscriminatorType	Specifies the column data type. Valid values are (prefixed by TDiscriminatorType) dtString — Discriminator column type will be string. Discriminator values must be strings. dtInteger — Discriminator column type will be integer. Discriminator values must be integer numbers.
Length	Specifies the length of column data type, only used when DiscriminatorType is string. If not specified, a default value is used.

Usage

```
[Inheritance(TInheritanceStrategy.SingleTable)]
[DiscriminatorColumn('MEDIA_TYPE', TDiscriminatorType.dtString)]
TMediaFile = class
```

4.1.11 DiscriminatorValue

Specifies the value that identifies a class in the discriminator column, when using single table inheritance strategy.

Level: Class Attribute

Description
Use DiscriminatorValue to define the value to be saved in the discriminator column when the class is saved. In a single table inheritance strategy, all classes are saved in the same table. Thus, when a subclass is saved, Aurelius updates an extra table column with a

value that indicates that the record contains that specific class. This value is specified in this DiscriminatorValue attribute. It's also used by Aurelius when the record is being read, so it knows which class needs to be instantiated when loading objects from database.

Constructor

```
constructor Create(Value: string); overload;
constructor Create(Value: Integer); overload;
```

Parameters

Value	The value to be used in the discriminator column. Value must be string or integer, depending on the type of the discriminator column declared in the DiscriminatorColumn attribute.

Usage

```
// Ancestor class:
[Inheritance(TInheritanceStrategy.SingleTable)]
[DiscriminatorColumn('MEDIA_TYPE', TDiscriminatorType.dtString)]
TMediaFile = class

// Child classes:
[DiscriminatorValue('SONG')]
TSong = class(TMediaFile)

[DiscriminatorValue('VIDEO')]
TVideo = class(TMediaFile)
```

4.1.12 PrimaryJoinColumn

Defines the primary key of a child table that will be referencing the primary key of a parent table, in a joined tables inheritance strategy.

Level: Class Attribute

Description
Use PrimaryJoinColumn attribute to specify the column that will be used as primary key of the child table. If you specified a joined tables inheritance strategy using the Inheritance attribute in the base class, then each descendant class will be saved in a different table in the database, and it will be linked to the table containing the data of the parent class. This relationship is one-to-one, so the child table will have a primary key of the same data type of the parent table's primary key. The child table's primary key will also be a foreign key referencing the parent table. So PrimaryJoinColumn attribute is used to define the name of the primary key column. Data type doesn't need to be defined since it will be the same as the parent primary key.
You can omit the PrimaryJoinColumn attribute. In this case, the name of table column used will be the same as the name of table column in the base class/table.

When the ancestor is a class with composite Id's, you can specify one PrimaryJoinColumn attribute for each table column in the ancestor class primary key. If you specify less PrimaryJoinColumn attributes than the number of columns in the primary key, the missing ones will be considered default, i.e, the name of the table column in the primary key will be used.

Constructor

```
constructor Create(Name: string);
```

Parameters

Name	The name of the child table column used as primary key and foreign key. If an empty string is provided, it will use the same name as the table column in the parent's class/table primary key

Usage

```
// Ancestor class:
[TABLE('MEDIA_FILES')]
[Inheritance(TInheritanceStrategy.JoinedTables)]
[Id('FId', TIdGenerator.IdentityOrSequence)]
TMediaFile = class
private
  [Column('MEDIA_ID', [TColumnProp.Required])]
  FId: integer;

// Child classes:
[TABLE('SONGS')]
[PrimaryJoinColumn('MEDIAFILE_ID')]
TSong = class(TMediaFile)

// In this case, a field with name MEDIA_ID will be created in table
'VIDEOS'
[TABLE('VIDEOS')]
[PrimaryJoinColumn('')]
TVideo = class(TMediaFile)

// In this case, a field with name MEDIA_ID will be created in table
'LIST_SHOWS'
// Since PrimaryJoinColumn attribute is not present
[TABLE('LIVE_SHOWS')]
TLiveShow = class(TMediaFile)
```

4.1.13 Sequence

Defines the sequence (generator) used to generate Id values.

Level: Class Attribute

Description
Use the Sequence attribute to define the database sequence (generator) to be created (if requested) and used by Aurelius to retrieve new Id values. If the database does not support sequences, or the generator type specified in the Id attribute does not use a database sequence, this attribute is ignored.

Constructor

```
constructor Create(SequenceName: string); overload;
```

```
constructor Create(SequenceName: string; InitialValue, Increment:
Integer); overload;
```

Parameters

SequenceName	The name of the sequence/generator in the database
InitialValue	The initial value of the sequence. Default value: 1
Increment	The increment used to increment the value each time a new value is retrieved from the sequence. Default value: 1.

Usage

```
[Sequence('SEQ_MEDIA_FILES')]
[Id('FId', TIdGenerator.IdentityOrSequence)]
TMediaFile = class
```

4.1.14 UniqueKey

Defines an exclusive (unique) index for the table.

Level: Class Attribute

Description
Use UniqueKey if you want to define a database-level exclusive (unique) index in the table associated with the class. Note that you do not need to use this attribute to define unique keys for field defined in the Id attribute, nor for columns defined as unique in the Column attribute. Those are created automatically by Aurelius.

Constructor

```
constructor Create(Columns: string);
```

Parameters

Columns	The name of the table columns that compose the unique key. If two or more names are specified, they must be separated by comma.

Usage

```
[UniqueKey('INVOICE_TYPE, INVOICENO')]
TTC_Invoice = class
```

4.1.15 Enumeration

Specifies how to save an enumerated type in the database.

Level: Enumerator Attribute

Description
Use Enumeration attribute if you have fields or properties of enumerated types and you want to save them in the database. Using Enumerator you define how the enumerated values will be saved and loaded from the database. The Enumerator attribute must be declared right above the enumerated type.

Constructor

```
constructor Create(MappedType: TEnumMappingType); overload;
constructor Create(MappedType: TEnumMappingType; MappedValues: string);
overload;
```

Parameters

MappedType	Indicated the type of the enumerated value in the database. Valid values are (prefixed by TEnumMappingType): emChar — Enumerated values will be saved as single-chars in the database emInteger — Enumerated values will be saved as integer values. The value used is the ordinal value of the enumerated type, i.e, the first value in the enumerator will be saved as 0, the second as 1, etc.. emString — Enumerated values will be saved as strings in the database
MappedValues	If MappedType is char or string, then you must use this parameter to specify the char/string values corresponding to each enumerated value. The values must be comma-separated and must be in the same order as the values in the enumerated type.

Usage

```
[Enumeration(TEnumMappingType.emChar, 'M,F')]
TSex = (tsMale, tsFemale);
```

4.1.16 Automapping

Indicates that the class is an entity class, and all its attributes are automapped.

Level: Class Attribute

Description
When Automapping attribute is present in the class, all mapping is done automatically by Aurelius, based on the class declaration itself. For more information about how automapping works, see Automapping section.
If AutoMappingMode in global configuration is set to Full, then you don't need to define this attribute - every entity class is considered to be automapped.

Constructor

```
constructor Create;
```

Parametes
None.

Usage

```
[Entity]
[Automapping]
TCustomer = class(TObject)
```

4.1.17 Transient

Indicates a non-persistent field in an automapped class.

Level: Field Attribute

Description
When the class is being automapped using Automapping attribute, by default every field in the class is persisted. If you don't want an specific field to be persisted, declare a Transient attribute before it.

Constructor

```
constructor Create;
```

Parametes
None.

Usage

```
[Entity]
[Automapping]
TCustomer = class(TObject)
private
  [Transient]
  FTempCalculation: integer;
```

4.1.18 Description

Allows to associate a description to the class or field/property.

Level: Class, Field or Property attribute

Description
Use Description attribute to better document your classes, fields and properties, by adding a string description to it. Currently this information is not used by Aurelius but this Description attribute can be created when generating classes from database using TMS Data Modeler tool. You can later at runtime retrieve this information for any purposes.

Constructor

```
constructor Create(AText: string);
```

Parametes

AText	The text to be associated with class, field or property

Usage

```
[Entity]
```

```
[Automapping]
[Description('Customer data')]
TCustomer = class(TObject)
private
```

4.2 Automapping

Automapping is an Aurelius feature that allows you to save a class without needing to specify all needed mapping attributes. Usually in an entity class you need to define table where data will be saved using Table attribute, then for each field or property you want to save you need to specify the Column attribute to define the table column in the database where the field/property will be mapped to, etc..

By defining a class as automapped, a lot of this mapping is done automatically based on class information, if it's not explicity specified. For example, the table name is automatically defined as the class name, with the "T" prefix removed.

To define a class as automapped, you just need to add the Automapping attribute to the class.

Automapping is not an all-or-nothing feature. Aurelius only performs the automatic mapping if no attribute is specified. For example, you can define a class as automapped, but you can still declare the Table attribute to specify a different table name, or you can use Column attribute in some specific fields or properties to override the default automatic mapping.

Below we list some of rules that automapping use to perform the mapping.

Table mapping

The name of table is assumed to be the name of the class. If the first character of the class name is an upper case "T", it is removed. For example, class "TCustomer" will be mapped to table "Customer", and class "MyInvoice" will be mapped to table "MyInvoice"

Column mapping

Every field in the class is mapped to a table column. Properties are ignored and not saved. If you don't want a specific class field to be saved automatically, add a Transient attribute to that class field.

The name of the table column is assumed to be name of the field. If the first character of the field name is an upper case "F", it is removed. For example, field "FBirthday" is mapped to table column "Birthday".

If the class field type is a Nullable<T> type, then the table column will be optional (nullable). Otherwise, the table column will be required (NOT NULL).

For currency fields, scale and precision are mapped to 4 and 18. For float fields, scale and precision are mapped to 8 and 18, respectively. If field is a string, length used will be the default length specified in the global configuration.

If the field is an object instance instead of an scalar value/primitive type, then it will be

mapped as an association, see below.

Associations

If the class field in an object instance (except a list), it will be mapped as an association to that class. The column name for the foreign key will be the field name (without "F") followed by "_ID". For example, if the class has a field

```
FCustomer: TCustomer
```

Aurelius will create an association with TCustomer and the name of table column holding the foreign key will be "Customer_ID".

If the class field is a list type (TList<T>) it will be mapped as a many-valued association. A foreign key will be created in the class used for the list. The name of table column holding the foreign key is field name + table name + "_ID". For example, if class TInvoice has a field:

```
FItems: TList<TInvoiceItem>
```

Aurelius will create a many-valued association with TInvoiceItem, and a table column holding the foreign key will be created in table "InvoiceItem", with the name "Items_Invoice_ID".

If the field type is a Proxy<T> type, fetch type of the association will be defined as lazy, otherwise, it will be eager.

Identifier

If no Id attribute is specified in the class, Aurelius will use a field named "FID" in the class as the class identifier. If such class field does not exist and no Id attribute is defined, an error will be raised when the class is saved.

Enumerations

Enumerations are not automapped unless the auto mapping mode is configured to Full in global configuration. In this case, if an enumeration type does not have an Enumeration attribute defined, it will be automapped as string type, and the mapped value will be the name of the enumerated value. For example, the enumerated type:

```
TSex = (seFemale, seMale);
```

will be mapped as string with mapped values 'seFemale', 'seMale'.

Sequences

If not specified, the name of the sequence to be created/used (if needed) will be "SEQ_" + table name. Initial value and increment will defined as 1.

Inheritance

Inheritance is not automapped and if you want to use it you will need explicitly define Inheritance attribute and the additional attributes needed for complete inheritance

mapping.

4.3 Nullable<T> Type

Table columns in databases can be marked as optional (nullable) or required (not null). When you map a class property to a table column in the database, you can choose if the column will be required or not.
If the column is optional, the column value hold one valid value, or it can be null. Problem is that primitive types in Delphi cannot be nullable. Using Nullable<T> type which is declared in unit Aurelius.Types.Nullable, you can create a property in your class that can represent the exact value in the database, i.e., it can hold a value, or can be nullable.

For example, suppose you have the following class field mapped to the database:

```
[Column('BIRTHDAY', [])]
FBirthday: TDate;
```

The column BIRTHDAY in the database can be null. But the field FBirthday in the class cannot be null. You can set FBirthday to zero (null date), but this is different from the NULL value in the database.
Thus, you can use the Nullable<T> type to allow FBirthday field to receive null values:

```
[Column('BIRTHDAY', [])]
FBirthday: Nullable<TDate>;
```

You can use FBirthday directly in expressions and functions that need TDate, Delphi compiler will do the implicit conversion for you:

```
FBirthday := EncodeDate(2000, 1, 1);
```

If the compiler fails in any situation, you can read or write the TDate value using Value property:

```
FBirthday.Value := Encode(2000, 1, 1);
```

To check if the field has a null value, use HasValue or IsNull property:

```
IsBirthdayNull := not FBirthday.HasValue;
IsBirthdayNull := FBirthday.IsNull;
```

There is global Nullable variable names SNull which represents the null value, you can also use it to read or write null values:

```
if FBirthday <> SNull then // birthday is not null
  FBirthday := SNull; // Set to null
```

4.4 Binary Large Objects (Blobs)

You can map binary (or text) large objects (Blobs) table columns to properties in your class. As with other properties, Aurelius will properly save and load the content of the

property to the specified table column in the database. In order for it to know that the class member maps to a blob, you must declare the data type as an array of byte:

```
[Column('Document', [])]
FDocument: TArray<byte>;
```

or as the TBlob type (recommended):

```
[Column('Photo', [])]
FPhoto: TBlob;
```

In both examples above, Aurelius will check the field data type and create a blob field in the table to hold the content of the binary data. Each SQL dialect uses a different data type for holding the blobs. Aurelius will choose the most generic one, i.e, that can hold any data (binary) and the largest possible amount of data. If the blob field already exists in the database, Aurelius will just load the field content in binary format and set it in the property.

In theory, you could use the TBytes type as well (and any other type that is an array of byte), however Delphi doesn't provide RTTI type info for the TBytes specifically. It might be a bug or by design, but you just can't use it. Use TArray<byte> or any other dynamic byte array instead (or TBlob of course).

Using TBlob type you have more flexibility and features, as described in topics below.

Lazy-Loading Blobs
TBlob Type

4.4.1 Lazy-Loading Blobs

When declaring blob attributes in your class, you can configure them for lazy-loading. It means that whenever Aurelius tries to retrieve an object from the database, it will not include the blob field in the select, and thus the blob content will not be sent through network from server to client unless it's needed. If you access the blob content through the blob property, then Aurelius will execute an SQL statement on-the-fly only to retrieve the blob content.

To map the blob property/field as lazy, you just need two requirements:

1. Use the TBlob type as the field/property type.
2. Add TColumnProp.Lazy to the column properties in the Column attribute.

The code below indicates how to declare a lazy-loaded blob:

```
TTC_Customer = class
strict private
  // <snip>
  [Column('Photo', [TColumnProp.Lazy])]
  FPhoto: TBlob;
```

The TBlob type is implicitly converted to an array of byte but also have methods for retrieving the blob content as TBytes, string, etc.. Whenever you try to access the blob data through the TBlob type, the blob content will be retrieved from the database.

4.4.2 TBlob Type

The TBlob type is used to declare blob field/properties. It's not required that you use a TBlob type, but doing so will allow you to configure lazy-loading blobs and also provides you with helper methods for handling the binary content.

Usage

```
TCustomer = class
private
  [Column('Photo', [TColumnProp.Lazy])]
  FPhoto: TBlob;
public
  property Photo: TBlob read FPhoto write FPhoto;
```

Implicit conversion to TBytes

A TBlob implicitly converts to TBytes so you can directly use it in any method that uses it:

```
BytesStream := TBytesStream.Create(Customer1.Photo);
// Use BytesStream anywhere that needs a TStream
```

Explicitly using AsBytes property

Alternatively you can use AsBytes property to get or set the value of the blob:

```
Customer1.Photo.AsBytes := MyBytesContent; // MyBytesContent is a TBytes
variable
```

Use AsString property to read/set the blob content as string

If you want to work with the blob content as string, you can just use AsString property for that:

```
Customer1.Photo.AsString := 'Set string directly to the blob';
```

Raw access to the data using Data and Size properties

If you want to have directly access to data, for high performance operations, without having to copy a byte array or converting data to a string, you can use read-only properties Data and Size. Data is a pointer (PByte) to the first byte of the data, and Size contains the size of blob data.
The code below saves the blob content into a stream:

```
MyStream := TFileStream.Create('BlobContent.dat', fmCreate);
try
  MyStream.Write(Customer1.Photo.Data^, Customer1.Photo.Size);
finally
  MyStream.Free;
end;
```

Using streams to save/load the blob

You can also use TBlob.LoadFromStream and SaveToStream methods to directly load blob content from a stream, or save to a stream:

```
MyStream := TFileStream.Create('BlobContent.dat', fmCreate);
try
  Customer1.Photo.LoadFromStream(MyStream);
  Customer1.Photo.SaveToStream(AnotherStream);
finally
  MyStream.Free;
end;
```

IsNull property

Use IsNull property to check if a blob is empty (no bytes):

```
if not Customer1.Photo.IsNull then
  // Do something
```

Clearing the blob

You can clear the blob content (set blob content to zero bytes) by setting IsNull property to true, or by calling Clear method:

```
// Clear Photo and Description blobs content.
// Both statement are equivalent
Customer1.Photo.IsNull := true;
Customer1.Photo.Clear;
```

4.5 Associations and Lazy-Loading

Aurelius supports associations between objects, which are mapped to foreign keys in the database. Suppose you have the following TInvoice class:

```
TInvoice = class
private
  FCustomer: TCustomer;
  FInvoiceItems: TList<TInvoiceItem>;
```

the class TInvoice has an association to the class TCustomer. By using Association mapping attribute, you can define this association and Aurelius deals with it automatically - customer data will be saved in its own table, and in Invoice table only thing saved will

be a value in a foreign key field, referencing the primary key in Customer table.

Also, TInvoice has a list of invoice items, which is also a type of association. You can define such lists using ManyValuedAssociation mapping attribute. In this case, the TInvoiceItem objects in the list will have a foreign key referencing the primary key in InvoiceTable.

Eager Loading

When an object is retrieved from the database, its properties are retrieved and set. This is also true for associations. By default, eager-loading is performed, which means associated objects and lists are loaded and filled when object is loaded. In the TInvoice example above, when a TInvoice instance is loaded, Aurelius also creates a TCustomer instance, fill its data and set it to the FCustomer field. Aurelius uses a single SQL statement to retrieve data for all associations. FInvoiceItems list is also loaded. In this case, an extra SELECT statement is performed to load the list.

Lazy Loading

You can optionally define associations to be lazy-loaded. This means that Aurelius will not retrieve association data from database until it's really needed (when the property is accessed). You define lazy-loading associations this way:

1. Declare the class field as a Proxy<TMyClass> type, instead of TMyClass
2. Declare the Association (or ManyValuedAssociation) attribute above the field, and define fetch mode as lazy in attribute parameters
3. Declare a property of type TMyClass with getter and setter that read/write from/to the proxy value field.

Example:

```
TMediaFile = class
private
  [Association([TAssociationProp.Lazy], [])]
  [JoinColumn('ID_ALBUM', [])]
  FAlbum: Proxy<TAlbum>;

  function GetAlbum: TAlbum;
  procedure SetAlbum(const Value: TAlbum);
public
  property Album: TAlbum read GetAlbum write SetAlbum;

implementation

function TMediaFile.GetAlbum: TAlbum;
begin
  Result := FAlbum.Value;
end;

procedure TMediaFile.SetAlbum(const Value: TAlbum);
begin
  FAlbum.Value := Value;
end;
```

In the example above, Album will not be loaded when TMediaFile object is loaded. But if in Delphi code you do this:

```
TheAlbum := MyMediaFileObject.Album;
```

then Aurelius will perform an extra SELECT statement on the fly, instantiate a new TAlbum object and fill its data.

Lazy loading lists

Lists can be set as lazy as well, which means the list will only be filled when the list object is accessed. It works in a very similar way to lazy-loading in normal associations. The only difference is that since you might need an instance to the TList object to manipulate the collection, you must initialize it and then destroy it. Note that you **should not** access Value property directly when creating/destroying the list object. Use methods SetInitialValue and DestroyValue. The code below illustrates how to do that.

```
TInvoice = class
private
  [ManyValuedAssociation([TAssociationProp.Lazy], CascadeTypeAll)]
  [ForeignJoinColumn('INVOICE_ID', [TColumnProp.Required])]
  FItems: Proxy<TList<TInvoiceItem>>;
private
  function GetItems: TList<TInvoiceItem>;
public
  constructor Create; virtual;
  destructor Destroy; override;
  property Items: TList<TInvoiceItem> read GetItems;
end;

implementation

constructor TInvoice_Lazy.Create;
begin
  FItems.SetInitialValue(TList<TInvoiceItem>.Create);
end;

destructor TInvoice_Lazy.Destroy;
begin
  FItems.DestroyValue;
  inherited;
end;

function TInvoice_Lazy.GetItems: TList<TInvoiceItem>;
begin
  result := FItems.Value;
end;
```

4.6 Inheritange Strategies

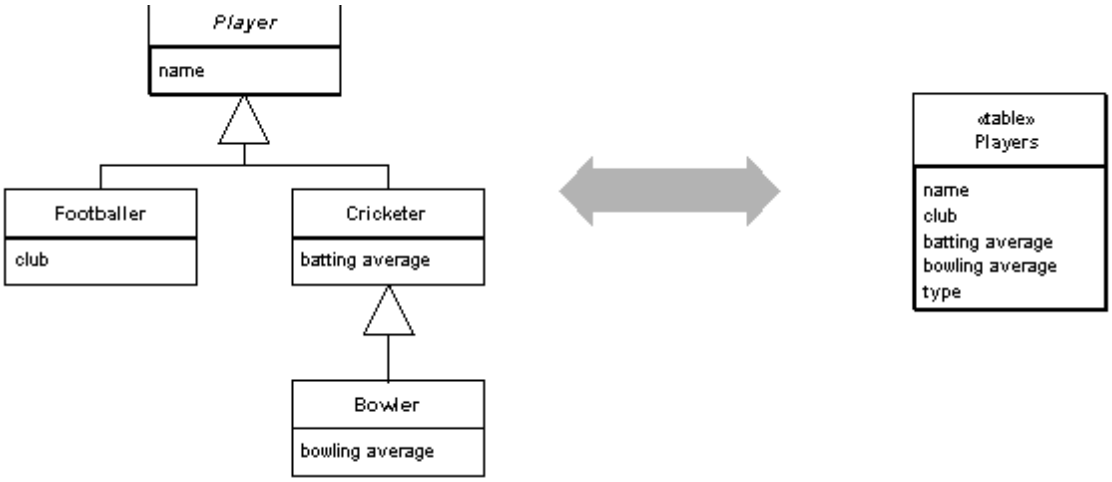
There are currently two strategies for you to map class inheritance into the relational database:

Single Table: All classes in the hierarchy are mapped to a single table in the database
Joined Tables**:** Each class is mapped to one different table, each one linked to the parent's table.

Inheritance is defined in Aurelius using the Inheritance attribute.

4.6.1 Single Table Strategy

With this strategy, all classes in the class hierarchy are mapped to a single table in relational database



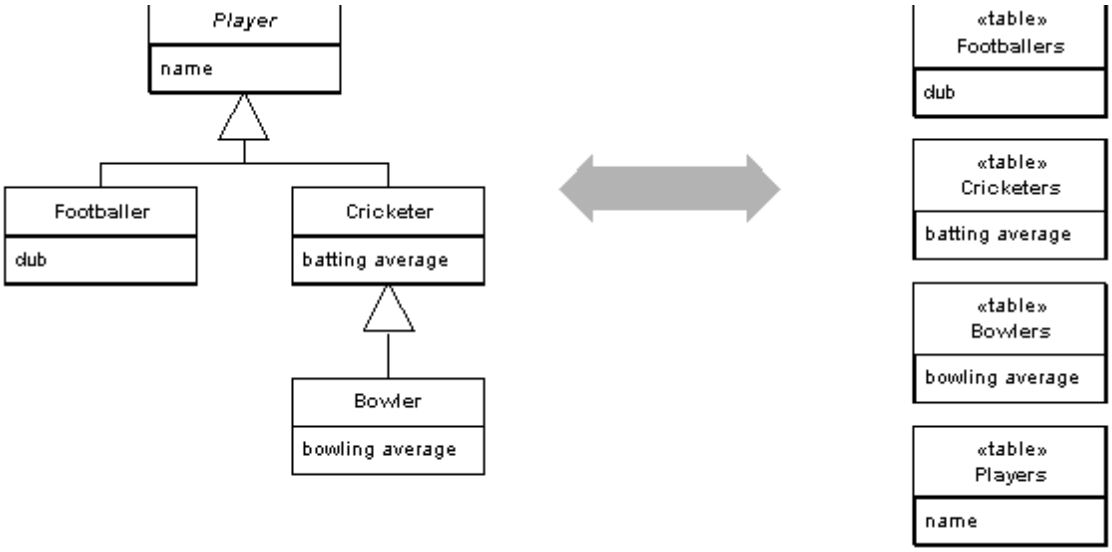
The concrete class of the object is indicated by the values in a special column in the table named *discriminator column.* This column is specified by the programmer and its content is used to identify the real class of the object. The discriminator column must be of string or integer type.

The advantage of this strategy is that the database is simple, and performance is optimized, since queries don't need to have too many joins or unions.

One disadvantage is that all columns belonging to child classes must be declared as not required, since they must be null if the row in the table corresponds to a super class.

4.6.2 Joined Tables Strategy

In this strategy there is one table for each class in the class hierarchy.



Each table represents a class in the hierarchy, and columns in the table are associated to the properties declared in the class itself. Even abstract classes have their own table, since they might have declared properties as well.

Tables are joined together using foreign keys. Each table representing a child class has a foreign key referencing the table representing the parent class. The foreign key is also the primary key, so the relationship cardinality between the tables is 1:1. In the previous illustration, the table Cricketer has a foreign key referencing the primary key in table Player.

The advantage of this strategy is that the database is normalized and the database model is very similar to the class model. Also, unlike the Single Table Strategy, all columns in tables are relevant to all table rows.

One disadvantage is performance. To retrieve a single object several inner or left joins might be required, becoming even worse when complex queries are used. Database refactoring is also more difficult - if you need to move a property to a different class in hierarchy, for example, more than one table needs to be updated.

4.7 Composite Id

You can use composite identifier in TMS Aurelius. Although possible, it's strongly recommended that you use single-attribute, single-column identifiers. The use of composite id should be used only for legacy applications where you already have a database schema that uses keys with multiple columns. Still in those cases you could try to add an auto-generated field in the table and use it as id.

Using composite Id's is straightforward: you just use the same attributes used for single Id: Id, JoinColumn, ForeignJoinColumn and PrimaryJoinColumn attributes. The only difference is that you add those attributes two or more times to the classes. For example, the following TAppointment class has a composite Id using the attributes AppointmentDate and Patient (you can use associations as well):

```
[Entity]
```

```
[Table('PERSON')]
[Id('FLastName', TIdGenerator.None)]
[Id('FFistName', TIdGenerator.None)]
TPerson = class
strict private
  [Column('LAST_NAME', [TColumnProp.Required], 50)]
  FLastName: string;
  [Column('FIRST_NAME', [TColumnProp.Required], 50)]
  FFirstName: string;
public
  property LastName: string read FLastName write FLastName;
  property FirstName: string read FFiratName write FFiratName;
end;

[Entity]
[Table('APPOINTMENT')]
[Id('FAppointmentDate', TIdGenerator.None)]
[Id('FPatient', TIdGenerator.None)]
TAppointment = class
strict private
  [Association([TAssociationProp.Lazy, TAssociationProp.Required],
[TCascadeType.Merge, TCascadeType.SaveUpdate])]
  [JoinColumn('PATIENT_LASTNAME', [TColumnProp.Required])]
  [JoinColumn('PATIENT_FIRSTNAME', [TColumnProp.Required])]
  FPatient: Proxy<TPerson>;
  [Column('APPOINTMENT_DATE', [TColumnProp.Required])]
  FAppointmentDate: TDateTime;
  function GetPatient: TPerson;
  procedure SetPatient(const Value: TPerson);
public
  property Patient: TPerson read GetPatient write SetPatient;
  property AppointmentDate: TDateTime read FAppointmentDate write
FAppointmentDate;
end;
```

Note that while TAppointment has a composite Id of two attributes, the number of underlying database table columns is three. This is because Patient attribute is part of Id, and the TPerson class itself has a composite Id. So primary key columns of table APPOINTMENT will be APPOINTMENT_DATE, PATIENT_LASTNAME and PATIENT_FIRSTNAME.

Also pay attention to the usage of JoinColumn attributes in field FPatient. Since TPerson has a composite Id, you must specify as many JoinColumn attributes as the number of table columns used for the referenced table. This is the same for ForeignJoinColumn and PrimaryJoinColumn attributes.

As illustrated in the previous example, you can have association attributes as part of a composite identifier. However, there is one limitation: you can't have lazy-loaded associations as part of the Id. All associations that are part of an Id are loaded in eager mode. In the previous example, although FPatient association was declared with TAssociationProp.Lazy, using a proxy, this settings will be ignored and the TPerson object will be fully loaded when a TAppointment object is loaded from the database.

When using composite Id, the generator specified in the Id attribute is ignored, and all are considered as TIdGenerator.None.

When using Id values for finding objects, for example when using Find method of object manager or using IdEq expression in a query, you are required to provide an Id value. The type of this value is Variant. For composite Id's, you must provide an array of variant (use VarArrayCreate method for that) where each item of the array refers to the value of a table column. For associations in Id's, you must provide a value for each id of association (in the example above, to find a class TAppointment you should provide a variant array of length = 3, with the values of appointment data, patient's last name and first name values.

4.8 Mapping Examples

This topic lists some code snippets that illustrates how to use attributes to build the object-relational mapping.

Examples:
Basic Mapping
Single-Table Inheritance and Associations
Joined-Tables Inheritance

4.8.1 Basic Mapping

```
unit Artist;

interface

uses
  Aurelius.Mapping.Attributes,
  Aurelius.Types.Nullable;

type
  [Entity]
  [Table('ARTISTS')]
  [Sequence('SEQ_ARTISTS')]
  [Id('FId', TIdGenerator.IdentityOrSequence)]
  TArtist = class
  private
    [Column('ID', [TColumnProp.Unique, TColumnProp.Required,
TColumnProp.NoUpdate])]
    FId: Integer;
    FArtistName: string;
    FGenre: Nullable<string>;
  public
    property Id: integer read FId;

    [Column('ARTIST_NAME', [TColumnProp.Required], 100)]
    property ArtistName: string read FArtistName write FArtistName;

    [Column('GENRE', [], 100)]
    property Genre: Nullable<string> read FGenre write FGenre;
  end;

implementation

end.
```

4.8.2 Single-Table Inheritance and Associations

In the example below, TSong and TVideo inherit from TMediaFile. The TMediaFile class has two associations: Album and Artist. Both are lazy associations

```
unit MediaFile;

interface

uses
  Generics.Collections,
  Artist, Album,
  Aurelius.Mapping.Attributes,
  Aurelius.Types.Nullable,
  Aurelius.Types.Proxy;

type
  [Entity]
  [Table('MEDIA_FILES')]
  [Sequence('SEQ_MEDIA_FILES')]
  [Inheritance(TInheritanceStrategy.SingleTable)]
  [DiscriminatorColumn('MEDIA_TYPE', TDiscriminatorType.dtString)]
  [Id('FId', TIdGenerator.IdentityOrSequence)]
  TMediaFile = class
  private
    [Column('ID', [TColumnProp.Unique, TColumnProp.Required,
TColumnProp.DontUpdate])]
    FId: Integer;
    FMediaName: string;
    FFileLocation: string;
    FDuration: Nullable<integer>;

    [Association([TAssociationProp.Lazy], [])]
    [JoinColumn('ID_ALBUM', [])]
    FAlbum: Proxy<TAlbum>;

    [Association([TAssociationProp.Lazy], [])]
    [JoinColumn('ID_ARTIST', [])]
    FArtist: Proxy<TArtist>;

    function GetAlbum: TAlbum;
    function GetArtist: TArtist;
    procedure SetAlbum(const Value: TAlbum);
    procedure SetArtist(const Value: TArtist);
  public
    property Id: integer read FId;

    [Column('MEDIA_NAME', [TColumnProp.Required], 100)]
    property MediaName: string read FMediaName write FMediaName;

    [Column('FILE_LOCATION', [], 300)]
    property FileLocation: string read FFileLocation write FFileLocation;

    [Column('DURATION', [])]
    property Duration: Nullable<integer> read FDuration write FDuration;

    property Album: TAlbum read GetAlbum write SetAlbum;
    property Artist: TArtist read GetArtist write SetArtist;
```

```
  end;

  [DiscriminatorValue('SONG')]
  TSong = class(TMediaFile)
  private
    FSongFormat: TSongFormat;
  public
    [Association]
    [JoinColumn('ID_SONG_FORMAT', [])]
    property SongFormat: TSongFormat read FSongFormat write FSongFormat;
  end;

  [DiscriminatorValue('VIDEO')]
  TVideo = class(TMediaFile)
  private
    FVideoFormat: TVideoFormat;
  public
    [Association]
    [JoinColumn('ID_VIDEO_FORMAT', [])]
    property VideoFormat: TVideoFormat read FVideoFormat write
FVideoFormat;
  end;

implementation

{ TMediaFile }

function TMediaFile.GetAlbum: TAlbum;
begin
  Result := FAlbum.Value;
end;

function TMediaFile.GetArtist: TArtist;
begin
  Result := FArtist.Value;
end;

procedure TMediaFile.SetAlbum(const Value: TAlbum);
begin
  FAlbum.Value := Value;
end;

procedure TMediaFile.SetArtist(const Value: TArtist);
begin
  FArtist.Value := Value;
end;

end.
```

4.8.3 Joined-Tables Inheritance

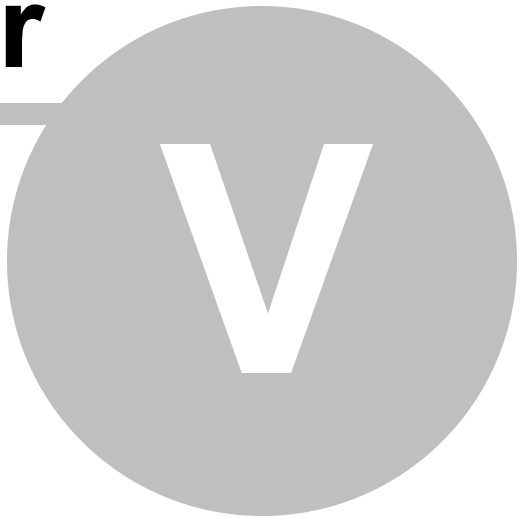
In this example, TBird and TMammal classes inherit from TAnimal. Each class has its own table. Specific bird data is saved in "BIRD" table, and common animal data is saved in

"ANIMAL" table.

```
unit Animals;

interface

uses
  Generics.Collections,
  Aurelius.Mapping.Attributes,
  Aurelius.Types.Nullable,
  Aurelius.Types.Proxy;

type
  [Entity]
  [Table('ANIMAL')]
  [Sequence('SEQ_ANIMAL')]
  [Inheritance(TInheritanceStrategy.JoinedTables)]
  [Id('FId', TIdGenerator.IdentityOrSequence)]
  TAnimal = class
  strict private
    [Column('ID', [TColumnProp.Unique, TColumnProp.Required,
TColumnProp.DontUpdate])]
    FId: Integer;
    [Column('ANIMAL_NAME', [TColumnProp.Required], 50)]
    FName: string;
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
  end;

  [Entity]
  [Table('BIRD')]
  [PrimaryJoinColumn('ANIMAL_ID')]
  TBird = class(TAnimal)
  strict private
    [Column('CAN_FLY', [], 0)]
    FCanFly: Nullable<boolean>;
    [Column('BIRD_BREED', [], 50)]
    FBirdBreed: Nullable<string>;
  public
    property CanFly: Nullable<boolean> read FCanFly write FCanFly;
    property BirdBreed: Nullable<string> read FBirdBreed write
FBirdBreed;
  end;

  [Entity]
  [Table('MAMMAL')]
  [PrimaryJoinColumn('ANIMAL_ID')]
  TMammal = class(TAnimal)
  strict private
    [Column('LAST_PREGNANCY_DAYS', [], 0)]
    FLastPregnancyDays: Nullable<integer>;
  public
    property LastPregnancyDays: Nullable<integer> read FLastPregnancyDays
write FLastPregnancyDays;
```

```
  end;

implementation

end.
```

Chapter



Mapping Setup

5 Mapping Setup

Aurelius uses the mapping you have done to manipulate the objects. You do the mapping at design-time (adding attributes to your classes and class members), but this information is of course retrieved at run-time by Aurelius and is cached for better performance. This cached information is kept in an object of class TMappingExplorer. Whenever a TObjectManager object is created to manipulate the objects, a TMappingExplorer object mas but provided to it, in order for the object manager to retrieve meta information about the mapping.
To create a TMappingExplorer object, you need to pass an instance of a TMappingSetup object.

So the order of "injection" of objects is illustrated below:

```
TMappingSetup -> TMappingExplorer -> TObjectManager
```

The following topics explain different ways of specifying the mapping setup and what custom settings you can do with mapping.

5.1 Defining a Mapping Setup

To have full control over the mapping setup, the overall behavior is the following.

1. Create and configure a TMappingSetup object
2. Create a TMappingExplorer object passing the TMappingSetup instance
3. Destroy the TMappingSetup object. **Keep** the TMappingExplorer instance
4. Create several TObjectManager instances passing the TMappingExplorer object
5. Destroy the TMappingExplorer object at the end of your application (or when all TObjectManager objects are destroyed and you have finished using Aurelius objects)

The concept is that you obtain a TMappingExplorer object that contains an immutable cache of the mapping scheme, using some initial settings defined in TMappingSetup. Then you keep the instance of that TMappingeExplorer during the lifetime of the application, using it to create several object manager instances.

Sample code:

```
uses
  Aurelius.Mapping.Setup,
  Aurelius.Mapping.Explorer,
  Aurelius.Engine.ObjectManager;

{...}

var
```

```
  MapSetup: TMappingSetup;
begin
  MapSetup := TMappingSetup.Create;
  try
    // Configure MapSetup object
    {..}

    // Now create exporer based on mapping setup
    FMappingExplorer := TMappingExplorer.Create(MapSetup);
  finally
    MapSetup.Free;
  end;

  // Now use FMappingExplorer to create instances of object manager
  FManager := TObjectManager.Create(MyConnection, FMappingExplorer);
  try
    // manipulate objects using the manager
  finally
    FManager.Free;
  end;

  // Don't forget to destroy FMappingExplorer at the end of application
end;
```

5.2 Default Mapping Setup Behavior

In most situations, you as a programmer don't need to worry about manually defining a mapping setup. This is because Aurelius provide some default settings and efault instances that makes it transparent for you (and also for backward compatibility).

There is a global TMappingExplorer object available in the following class function:

```
class function TMappingExplorer.DefaultInstance: TMappingExplorer;
```

that is lazily initialized that is used by Aurelius when you don't explicitly define a TMappingExplorer to use. That's what makes you possible to instantiate TObjectManager objects this way:

```
Manager := TObjectManager.Create(MyConnection);
```

the previous code is equivalent to this:

```
Manager := TObjectManager.Create(MyConnection,
TMappingExplorer.DefaultInstance);
```

Note that the TMappingSetup object is not specified here. It means that the TMappingExplorer object initially available in TMappingExplorer.DefaultInstance internally uses an empty TMappingSetup object. This just means that no customization in the setup was done, and the default mapping (and all the design-time mapping done by you) is used normally.

If you still want to define a custom mapping setup, but you don't want to create all your object manager instances passing a new explorer, you can alternatively change the TMappingExplorer.DefaultInstance. This way you can define a custom setup, and from that point, all TObjectManager objects to be created without an explicit TMappingExplorer parameter will use the new default instance. The following code illustrates how to change the default instance:

```
uses
  Aurelius.Mapping.Setup,
  Aurelius.Mapping.Explorer,
  Aurelius.Engine.ObjectManager;

{...}

var
  MapSetup: TMappingSetup;
begin
  MapSetup := TMappingSetup.Create;
  try
    // Configure the mapping setup

    // Replace default instance of TMappingExplorer
    // MAKE SURE that no TObjectManager instances are alive using the old
DefaultInstance
    TMappingExplorer.ReplaceDefaultInstance(TMappingExplorer.Create
(MapSetup));
  finally
    MapSetup.Free;
  end;

  FManager := TObjectManager.Create(MyConnection);

  try
    // manipulate objects using the manager
  finally
    FManager.Free;
  end;

  // No need to destroy the old or new default instances. Aurelius will
manager them.
end;
```

Please attention to the comment in the code above. Make sure you have no existing TObjectManager instances that uses the old TMappingExplorer instance being replaced. This is because when calling ReplaceDefaultInstance method, the old default instance of TMappingExplorer is destroyed, and if there are any TObjectManager instances referencing the destroyed explorer, unexpected behavior might occur.
Nevertheless, you would usually execute such example code above in the beginning of your application.eu

5.3 Mapped Classes

By default, TMS Aurelius maps all classes in the application marked with Entity attribute. Alternatively, you can manually define which class will be mapped in each mapping setup. This allows you to have a differents set of classes for each database connection in the same application. For example, you can have classes A, B and C mapped to a SQL Server connection, and classes D and E mapped to a local SQLite connection.

Defining mapped classes

Mapped classes are defined using TMappingSetup.MappedClasses property. This provides you a TMappedClasses class which several methods and properties to define the classes to be mapped.

```
uses
  Aurelius.Mapping.Setup,
  Aurelius.Mapping.Explorer,
  Aurelius.Mapping.MappedClasses,
  Aurelius.Engine.ObjectManager;

{...}

var
  MapSetup1: TMappingSetup;
  MapSetup2: TMappingSetup;
begin
  MapSetup1 := TMappingSetup.Create;
  MapSetup2 := TMappingSetup.Create;
  try
    MapSetup1.MappedClasses.RegisterClass(TCustomer);
    MapSetup1.MappedClasses.RegisterClass(TCountry);
    MapSetup2.MappedClasses.RegisterClass(TInvoice);
    FMappingExplorer1 := TMappingExplorer.Create(MapSetup1);
    FMappingExplorer2 := TMappingExplorer.Create(MapSetup2);
  finally
    MapSetup.Free;
  end;

  // FManager1 will connect to SQL Server and will only deal with entity
classes TCustomer and TCountry
  FManager1 := TObjectManager.Create(MySQLServerConnection,
FMappingExplorer1);
  // FManager2 will connect to SQLite and will only deal with entity
classe TInvoice
  FManager2 := TObjectManager.Create(MySQLiteConnection,
FMappingExplorer2);

  // Don't forget to destroy FMappingExplorer1 and FMappingExplorer2 at
the end of application
end;
```

Default behavior

You **don't** need to manually register classes in MappedClasses property. If it is empty, Aurelius will automatically register all classes in the application marked with the Entity attribute.

Methods and properties

The following methods and properties are available in TMappedClasses class.

```
procedure RegisterClass(Clazz: TClass);
```

Registers a class in the mapping setup.

```
procedure RegisterClasses(AClasses: TEnumerable<TClass>);
```

Register a set of classes in the mapping setup (you can pass a TList<TClass> or any other class descending from TEnumerable<TClass>.

```
procedure Clear;
```

Unregister all mapped classes. This returns to the default state, where all classes marked with Entity attribute will be registered.

```
function IsEmpty: boolean;
```

Indicates if there is any class registered as a mapped class. When IsEmpty returns true, it means that the default classes will be used (all classes marked with Entity attribute).

```
property Classes: TEnumerable<TClass> read GetClasses;
```

Lists all classes currently registered as mapped classes.

```
procedure UnregisterClass(Clazz: TClass);
```

Unregister a specified class. This method is useful when combined with GetEntityClasses. As an example, the following will register all classes marked with Entity attribute (the default classes), except TInternalConfig:

```
MapSetup.MappedClasses.RegisterClasses(TMappedClasses.GetEntityClasses);
MapSetup.MappedClasses.UnregisterClass(TInternalConfig);
```

```
class function GetEntityClasses: TEnumerable<TClass>;
```

Returns all classes in the application marked with Entity attribute. This is not a list of the currently mapped classes (use Classes property for that). This property is just a helper property in case you want to register all classes marked with Entity attribute and then

remove some classes. It's useful when used together with UnregisterClass method.

5.4 Dynamic Properties

Dynamic properties are a way to define mapping to database columns at runtime. Regular mapping is done as following:

```
[Column('MEDIA_NAME', [TColumnProp.Required], 100)]
property MediaName: string read FMediaName write FMediaName;
```

But what if don't know at design-time if the MEDIA_NAME column will be available in the database? What if your application runs in many different customers and the database schema in each customer is slightly different and columns are not known at design-time? To solve this problem, you can use dynamic properties, which allows you to manipulate the property this way:

```
MyAlbum.CustomProps['MediaName'] := 'My media name';
```

The following steps describe how to use them:

Preparing Class for Dynamic Properties
Registering Dynamic Properties
Using Dynamic Properties
Dynamic Properties in Queries and Datasets

5.4.1 Preparing Class for Dynamic Properties

To make your class ready for dynamic properties, you must add a new property that will be used as a container of all dynamic properties the object will have. This container must be managed (created and destroyed) by the class and is an object of type TDynamicProperties:

```
uses
  Aurelius.Mapping.Attributes,
  Aurelius.Types.DynamicProperties,

type
  [Entity]
  [Automapping]
  TPerson = class
  private
    FId: integer;
    FName: string;
    FProps: TDynamicProperties;
  public
    constructor Create;
    destructor Destroy; override;
    property Id: integer read FId write FId;
    property Name: string read FName write FName;
    property Props: TDynamicProperties read FProps;
  end;
```

```
constructor TPerson.Create;
begin
  FProps := TDynamicProperties.Create;
end;

destructor TPerson.Destroy;
begin
  FProps.Free;
  inherited;
end;
```

The Automapping attribute is being used in the example, but it's not required to use dynamic properties. You just need to declare the TDynamicProperties property, with no attributes associated to it.

5.4.2 Registering Dynamic Properties

Dynamic properties must be registered at run-time. To do that, you need to use a custom mapping setup. You need to create a TMappingSetup object, register the dynamic properties using DynamicProps property, and then create a TMappingExplorer object from this setup to be used when creating TObjectManager instances, or just change the TMappingExplorer.DefaultInstance.

The DynamicProps property is an indexed property which index is the class where the dynamic property will be registered. The property returns a TList<TDynamicProperty> which you can use to manipulate the registered dynamic properties. You don't need to create or destroy such list, it's managed by the TMappingSetup object. You just add TDynamicProperty instances to it, and you also don't need to manage such instances.

The following code illustrates how to create some dynamic properties in the class TPerson we created in the topic "Preparing Class for Dynamic Properties".

```
uses
   {...}, Aurelius.Mapping.Setup;

procedure TDataModule1.CreateDynamicProps(ASetup: TMappingSetup);
var
  PersonProps: TList<TDynamicProperty>;
begin
  PersonProps := ASetup.DynamicProps[TPerson];
  PersonProps.Add(
    TDynamicProperty.Create('Props', 'HairStyle', TypeInfo(THairStyle),
      TDynamicColumn.Create('HAIR_STYLE')));
  PersonProps.Add(
    TDynamicProperty.Create('Props', 'Photo', TypeInfo(TBlob),
      TDynamicColumn.Create('PHOTO')));
  PersonProps.Add(
    TDynamicProperty.Create('Props', 'Extra', TypeInfo(string),
      TDynamicColumn.Create('COL_EXTRA', [], 30)));
end;

procedure TDataModule1.DefineMappingSetup;
var
```

```
  MapSetup: TMappingSetup;
begin
  MapSetup := TMappingSetup.Create;
  try
    CreateDynamicProps(MapSetup);
    TMappingExplorer.ReplaceDefaultInstance(TMappingExplorer.Create
(MapSetup));
  finally
    MapSetup.Free;
  end;
end;
```

In the previous example, we have registered three dynamic properties in class TPerson:
- HairStyle, which is a property of type THairStyle (enumerated type) and will be saved in database column HAIR_STYLE
- Photo, a property of type TBlob, to be saved in column PHOTO
- Extra, a property of type string, to be saved in column COL_EXTRA, size 30.

Note that the type of dynamic property must be informed. It should be the type of the **property** (not the type of database column) as if the property was a real property in the class. You can create dynamic properties of any type supported by Aurelius, with two exceptions: associations are not supported (and such Proxy types are not allowed) and Nullable types are also not supported, but because they are not needed. All dynamic properties are nullable because they are in essence TValue types and you can always set them to TValue.Empty values (representing a null value).

The first parameter of TDynamicProperty.Create method must have the name of the TPerson property which will hold the dynamic property values (we have created a property Props of type TDynamicProperties in class TPerson).

Declaration of TDynamicProperty and TDynamicColumn objects are as following:

```
  TDynamicProperty = class
  public
    constructor Create(AContainerName, APropName: string; APropType:
PTypeInfo; ColumnDef: TDynamicColumn);
    destructor Destroy; override;
    function Clone: TDynamicProperty;
    property ContainerName: string read FContainerName write
FContainerName;
    property PropertyName: string read FPropertyName write FPropertyName;
    property PropertyType: PTypeInfo read FPropertyType write
FPropertyType;
    property Column: TDynamicColumn read FColumn write FColumn;
  end;

  TDynamicColumn = class
  public
    constructor Create(Name: string); overload;
    constructor Create(Name: string; Properties: TColumnProps); overload;
    constructor Create(Name: string; Properties: TColumnProps; Length:
Integer); overload;
```

```
    constructor Create(Name: string; Properties: TColumnProps; Precision,
Scale: Integer); overload;
    function Clone: TDynamicColumn;
    property Name: string read FName write FName;
    property Properties: TColumnProps read FProperties write FProperties;
    property Length: integer read FLength write FLength;
    property Precision: integer read FPrecision write FPrecision;
    property Scale: integer read FScale write FScale;
  end;
```

Note that the overloaded Create methods of TDynamicColumn are very similar to the ones used in Column attribute. The TDynamicColumn represents contains info about the physical table column in the database where the dynamic property will be mapped to, and its properties behave the same as the ones described in the documentation of Column attribute.

5.4.3 Using Dynamic Properties

Once you have prepared your class for dynamic properties, and registered the dynamic properties in the mapping setup, you can manipulate the properties as any other property of your object, using the TDynamicProperties container object. It's declared as following:

```
TDynamicProperties = class
public
  property Prop[const PropName: string]: TValue read GetItem write
SetItem; default;
  property Props: TEnumerable<TPair<string, TValue>> read GetProps;
end;
```

This is how you would use it:

```
Person := Manager.Find<TPerson>(PersonId);
Person.Props['Extra'] := 'Some value';
Manager.Flush;
ExtraValue := Person.Props['Extra'];
```

Note that in the example above, the dynamic property behave exactly as a regular property. The Flush method have detected that the "Extra" property was changed, and will update it in the database accordingly.
Be aware that Props type is TValue, which is a generic container. Some implicit conversions are possible, as illustrated in the previous example using the dynamic property "Extra". However, in some cases (and to be safe you can use this approach whenever you are not sure about using it or not) you will need to force the TValue to hold the correct type of the property. The following example shows how to define a value for the dynamic property HairStyle, which was registered as the type THairStyle (enumerated type):

```
Person := TPerson.Create;
Person.Props['HairStyle'] := TValue.From<THairStyle>(THairStyle.Long);
Manager.Save(Person);
PersonHairStyle := Person.Props['HairStyle'].AsType<THairStyle>;
```

The same applies to blob properties, which must be of type TBlob:

```
var
  Blob: TBlob;
begin
  // Saveing a blob
  Blob.LoadFromStream(SomeStream);
  Person.Props['Photo'] := TValue.From<TBlob>(Blob);
  Manager.SaveOrUpdate(Person);

  // Reading a blob
  Blob := Person.Props['Photo'].AsType<TBlob>;
  Blob.SaveToStream(MyStream);
```

Dynamic blob properties can also be lazy-loaded just as any regular blob property.

5.4.4 Dynamic Properties in Queries and Datasets

When it comes to queries and datasets, dynamic properties behave exactly as regular properties. In queries, they are accessed by name as any other query. So for example the following query:

```
Results := Manager.Find<TPerson>
  .Where(
    TLinq.Eq('HairStyle', THairStyle.Long) and
    TLinq.Like('Extra', '%value%')
  )
  .AddOrder(TOrder.Asc('Extra'))
  .List;
```

will list all people with HairStyle equals to Long and Extra containing "value", ordered by Extra. No special treatment is required, and the query doesn't care if HairStyle or Extra are dynamic or regular properties.

The same applies to the TAureliusDataset. The dynamic properties are initialized in fielddefs as any other property, and can be accessed through dataset fields:

```
//  DS: TAureliusDataset;
  DS.Manager := Manager;
  Person := TPerson.Create;
  DS.SetSourceObject(Person);
  DS.Open;
  DS.Edit;
  DS.FieldByName('Name').AsString := 'Jack';
  DS.FieldByName('Extra').AsString := 'extra value';
  // Enumerated types are treated by its ordinal value in dataset
```

```
  DS.FieldByName('HairStyle').AsInteger := Ord(THairStyle.Short);
  BlobField := DS.FieldByName('Photo') as TBlobField; // use BlobField as
usual
```

Chapter

Manipulating Objects

6 Manipulating Objects

This chapter explains how to manipulate objects. Once you have properly connected to the database and configure all mapping between the objects and the database, it's time for the real action. The following topics explain how to save, update, delete and other topics about dealing with objects. Querying objects using complex criteria and projections is explained in a specific chapter only for queries.

6.1 Object Manager

The Object Manager is implemented by the TObjectManager class which is declared in unit Aurelius.Engine.ObjectManager:

```
uses
  {...}, Aurelius.Engine.ObjectManager;
```

It's the layer between your application and the database, providing methods for saving, loading, updating, querying objects. It performs memory management, by controlling objects lifetime cycle, destroying them when they are not needed anymore, and caching objects by using identity mappings to ensure a single object is not loaded twice in the same manager.

The Object Manager also keeps tracking of changes in objects - you can update the content of objects (change properties, add associations, etc.) and then call *Flush* method to ask the object manager to update all object changes in the database at once.

The list below is a quick reference for the main methods and properties provided by TObjectManager object. A separate topic is provided for each method listed below:

Creating a new object manager

Directly create a TObjectManager instance, passing the IDBConnection interface that represents a database connection:

```
Manager := TObjectManager.Create(MyConnection);
try
  // perform operations with objects
finally
  Manager.Free;
end;
```

alternatively, you can also pass a TMappingExplorer instance, which holds a custom

mapping setup.

```
Manager := TObjectManager.Create(MyConnection, MyMappingExplorer);
```

Save method

Use it to save (insert into database) new entity objects:

```
Customer := TCustomer.Create;
Customer.Name := 'TMS Software';
Manager.Save(Customer);
```

Update method

Use it to update an existing object in the database:

```
Customer := TCustomer.Create;
Customer.Id := 10;
Customer.Email := 'customer@company.com';
Manager.Update(Customer);
```

SaveOrUpdate method

Use it to save or update an object depending on the Id specified in the object (update if there is an Id, save it otherwise):

```
Customer.LastName := 'Smith';
Manager.SaveOrUpdate(Customer);
```

Flush method

Commit to the database all changes made to the managed objects.

```
Customer1 := Manager.Find<TCustomer>(1);
Customer2 := Manager.Find<TCustomer>(2);
Customer1.Email := 'company@address.com';
Customer2.City := 'Miami';
Manager.Flush; // Update Customer1 e-mail and Customer2 city in database
```

Merge method

Use it to merge a transient object into the object manager and obtain the persistent object.

```
Customer := TCustomer.Create;
Customer.Id := 12;
Customer.Name := 'New name';
ManagedCustomer := Manager.Merge<TCustomer>(Customer);
```

In the example above, Merge will look in the cache or database for a TCustomer with id equals to 12. If it's not found, an exception is raised. If found, it will update the cached customer object with the new information and return a reference to the cached object in CachedCustomer. Customer reference will still point to an unmanaged object, so two

instances of TCustomer will be in memory.

Find method

Use Find method to retrieve an object given its Id:

```
Customer := Manager.Find<TCustomer>(CustomerId);
```

The id value is a variant type and must contain a value of the same type of the class Identifier (specified with the Id attribute). For example, if the identifier is a string type, id value must be a variant containing a string. For classes with composite id, a variant array of variant must be specified with all the values of the id fields.

You can alternatively use the non-generic overload of Find method. It might be useful for runtime/dynamic operations where you don't know the object class at the compile time:

```
Customer := Manager.Find(TCustomer, CustomerId);
```

FindAll method

Return all object instances of an specified class in the database. It's equivalent to perform an SELECT statement over a table without any filtering. A TObjectList<T> is returned.

```
var
  AllCustomers: TObjectList<TCustomer>;
begin
  AllCustomers := Manager.FindAll<TCustomer>;
  try
    {use AllCustomers}
  finally
    AllCustomers.Free;
  end;
```

Remove method

Use it to remove the object from the persistence (i.e., delete it from database and from object manager cache).

```
CustomerToRemove := Manager.Find<TCustomer>(CustomerId);
Manager.Remove(CustomerToRemove);
```

Find<T> method

Use Find<T> to create a new query to find objects based on the specified criteria.

```
Results := Manager.Find<TTC_Customer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

CreateCriteria<T> method

CreateCriteria is just an alias for Find<T> method. Both are equivalent:

```
Results := Manager.CreateCriteria<TTC_Customer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

IsAttached method

Checks if the specified object instance is already attached (persistent) in the object manager.

```
if not Manager.IsAttached(Customer) then
  Manager.Update(Customer);
```

OwnsObjects property

If true (default), all managed objects are destroyed when the TObjectManager object is destroyed. If false, the objects remain in memory.

```
Customer := Manager.Find<TCustomer>(CustomerId);
Manager.OwnsObjects := false;
Manager.Free;
// Customer object is still available after Manager is destroyed
```

ProxyLoad and BlobLoad methods

use to load a proxy object (or blob) based on meta information (see Lazy-Loading with JSON for more information)

```
function ProxyLoad(ProxyInfo: IProxyInfo): TObject;
function BlobLoad(BlobInfo: IBlobInfo): TArray<byte>;
```

6.2 Memory Management

Entity objects are saved and loaded to/from database using a TObjectManager object, which provides methods and properties for such operations. All entity objects cached in TObjectManager are managed by it, and you don't need to free such objects (unless you set OwnsObjects property to False). Also, entity objects retrieved from database, either loading by identifier or using queries, are also managed by the TObjectManager.

Concept of object state

In Aurelius when an object is told to be *persistent* (or *cached,* or *managed)* it means that the TObjectManager object is aware of that object and is "managing" it. When TObjectManager loads any object from the database, the object instances created in the loading process are persistent. You can also turn objects into persistent object when you for example call Save, Update or Merge methods.

When the TObjectManager is not aware of the object, the object is told to be *transient* (or *uncached,* or *unmanaged*).

Don't confuse a transient object with an object that is not saved into the database yet. You might have a TCustomer object which has been already saved in the database, but if the TCustomer instance you have is not being managed by the TObjectManager, it's transient.

Also, don't confuse persistent with saved. A persistent object means that TObjectManager is aware of it and it's managing it, but it might not be saved to the database yet.

Object lists
It's important to note that when retrieving object lists from queries, the list itself must be destroyed, although the objects in it are not. Note that when you use projections in queries, the objects returned are not entity objects, but result objects. In this case the objects are not managed by the object manager, but the list retrieved in result queries have their OwnsObjects set to true, so destroying the list will destroy the objects as well.

Unique instances
When dealing with entity objects (saving, loading, querying, etc.), object manager keeps an internal Identity Map to ensure that only one instance of each entity is loaded in the TObjectManager object. Each entity is identified by it's Id attribute. So for example, if you execute two different queries using the same object manager, and the query returns the same entity (same id) in the queries, the object instance in the both queries returned will be the same. The object manager will not create a different object instance every time you query the object. If you use a different TObjectManager object for each query, then you will have different instances of the same entity object

Examples
The code snippets below illustrates several the different situations mentioned above.

Saving objects

```
Customer := TTC_Customer.Create;
Customer.Name := 'Customer Name';
ObjectManager1.Save(Customer);
// From now on, you don't need to destroy Customer object anymore
// It will be destroyed when ObjectManager1 is destroyed
```

Loading objects

```
Customer := Manager1.Find<TCustomer>(CustomerId);
Customer2 := Manager1.Find<TCustomer>(CustomerId);
// Since CustomerId is the same for both queries, the same instance will
be
// returned in Customer and Customer2 (Customer = Customer2), and you
don't
// need to destroy such instance, it's manager by Manager1.
```

Retrieving entities from queries

```
Results := Manager.CreateCriteria<TCustomer>
  .Add(TExpression.Eq('Name', 'TMS Software'))
  .List;
Results.Free;
// Results is a TObjectList<TCustomer> object that needs to be destroyed
// However, the object instances it holds are not destroyed and are kept
// in Manager cache. The instances are also ensured to be unique in
Manager context
```

Retrieving projected query results

```
Results := Manager.CreateCriteria<TTC_Estimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo'))
    .Add(TProjections.Group('c.Name'))
    )
  .ListValues;
Results.Free;
// In this case the query does not return entity objects, but result
objects (TCriteriaResult)
// Such result objects are not managed by TObjectManager. However, in
this case,
// The Results object list is returned with its OwnsObjects property set
to true. Thus, when
// you destroy Results object, the TCriteriaResult objects it holds will
also be destroyed.
```

Using unmanaged objects

If for some reason you want to keep object instances available even after the object manager is destroyed (for example, after a query, you want to destroy object manager but keep the returned objects in memory), then just set the TObjectManager.OwnsObjects property to false:

```
Manager.OwnsObjects := false;
Results := Manager.CreateCriteria<TCustomer>
  .Add(TExpression.Eq('Name', 'TMS Software'))
  .List;
Manager.Free;
// Now although Manager object was destroyed, all objects in Results list
will be kept in memory,
// EVEN if you destroy Results list itself later.
```

6.3 Saving Objects

Using TObjectManager you can save (insert) objects using Save method. It is analog to SQL INSERT statement - it saves the object in database.

```
Customer1 := TCustomer.Create;
Customer1.Name := 'John Smith';
Customer1.Sex := tsMale;
Customer1.Birthday := EncodeDate(1986, 1, 1);
Manager1.Save(Customer1);
```

The identifier of the object (mapped using Id attribute) must not have a value, otherwise an exception will be raised - unless the generator defined in Id attribute is TIdGenerator.None. In this case, you must manually provide the id value of the object, and so of course Aurelius will accept an object with an id value. But you must be sure that there are no objects in the database with the same id value, to avoid duplicate values in the primary key.

When saving an object, associations and items in collections might be saved as well, depending on how cascade options are set when you defined the Association and ManyValuedAssociation attribute. In the example below, customer is defined to have SaveUpdate cascade. It means that when invoice is saveed, the customer is saved as well, before the invoice.

```
Customer := TTC_Customer.Create;
Customer.Name := 'Customer Name';
Invoice := TTC_Invoice.Create;
Invoice.InvoiceType := 999;
Invoice.InvoiceNo := 123456;
Invoice.Customer := Customer;
Invoice.IssueDate := Date;
Manager1.Save(Invoice);
```

You can also use SaveOrUpdate method to save objects. The difference from Save is that if the object has an id value set, SaveOrUpdate will internally call Update method instead of Save method. So, if you use TIdGenerator.None in the Id attribute of your object class, SaveOrUpdate will not work.

6.4 Updating Objects

You modify objects using the TObjectManager methods Flush or Update. If you are modifying objects that are already attached to the object manager, use Flush. For transient objects, use Update. Thus, if you need to load the object in the object manager using Find method, or using a query, or using Merge method (see below), use Flush method. Once the object is loaded, you can just update it and call Flush method:

```
Customer1 := Manager1.Find<TCustomer>(CustomerId);
Customer1.Email := 'newemail@domain.com';
Customer2 := Manager1.Find<TCustomer>(Customer2Id);
Customer2.Email := 'another@email.com';
Manager1.Flush;
```

The Flush method will detect all objects which content has been changed since they were loaded, and then update them all in the database. In the example above, both customers 1 and 2 will have their e-mail changed.

If you have an instance of the object but it's not attached to object manager, you can

use the Update method. This method will just take the passed transient instance and attach it to the TObjectManager. Then it will update the object in the database using the current object settings. If there was already another object attached to the TObjectManager with the same id, an exception will be raised (You could use Merge method in this case).

```
Customer2 := Manager1.Find<TCustomer>(Customer2Id);
Manager1.OwnsObjects := false;
Manager2.Free;
Customer2.Name := 'Mary';
Customer2.Sex := tsFemale;
Manager2.Update(Customer2);
```

In the example above, a TCustomer object was loaded in Manager1. It's not attached to Manager2. When Update method is called in Manager2, all data in Customer2 object will be updated to the database, and it will become persistent in Manager2.

The cascades defined in Association attributes in your class are applied here. Any associated object or collection item that has TCascadeType.SaveUpdate defined will also be updated in database.

6.5 Merging Objects

When you use Update method in a TObjectManager object, there should be no managed object with same Id in the object manager, otherwise an exception is raised. You can avoid such exception using the Merge method. This method will take a transient instance and merge it into the persistent instance. In other words, all the content of the transient object will be copied to the persistent object. Note that the transient object will continue to be transient. If there is not persistent object in the object manager with the same id, the object manager will load an object from the database with the same id of the transient object being merged. If no object is found in the database, an exception will be raised.

```
Customer2 := TCustomer.Create;
Customer2.Id := Customer2Id;
Customer2.Name := 'Mary';
Customer2.Sex := tsFemale;
MergedCustomer := Manager2.Merge<TCustomer>(Customer2);
```

In the example above, a TCustomer object was created and assigned an existing id. When calling Merge method, all data in Customer2 will be copied to the persistent object with same id in Manager2. If no persistent object exists in memory, it will be loaded from the database. Customer2 variable will still reference a transient object. The result value of Merge method is a reference to the persistent object in the object manager.

Note that Merge does nothing in the database - it justs update the persistent object in memory. To effectively update the object in the database you should then call Flush method.

The cascades defined in Association and ManyValuedAssociation attributes in your class are applied here. Any associated object or collection item that has TCascadeType.Merge

defined will also be updated in database.

6.6 Removing Objects

You can remove an object from the database using Remove method from a TObjectManager object. Just pass the object that you want to destroy. The object must be attached to the object manager.

```
Customer1 := Manager1.Find<TCustomer>(CustomerId);
Manager1.Remove(Customer1);
```

The cascades defined in Association and ManyValuedAssociation attributes in your class are applied here. Any associated object or collection item with delete cascade will also be removed from database.

6.7 Finding Objects

You can quickly find (load) objects using Find method of TObjectManager. You just need to pass the Id of the object, and object manager will retrieve the instance of the object loaded in memory. If the object is not attached to the object manager (not in memory), then it tried to load the object from database. If it's not found, an exception is raised.

```
Customer1 := Manager1.Find<TCustomer>(CustomerId);
// Customer1 has an instance to the loaded customer object.
```

The association will be loaded depending on how the fetch mode was defined in Association attribute. They will be loaded on the fly or on demand, depending if they are set as lazy-load associations or not.

If you want to retrieve all objects in database for a specified class, use the FindAll method. It will return a TList<T> of objects:

```
var
  Customers: TList<TCustomer>;
begin
  Customers := Manager1.FindAll<TCustomer>;
```

If you want to narrow the results by retrieving some objects given a specified criteria, then you should use queries instead.

Chapter

Queries

7 Queries

You can perform queries with Aurelius, just like you would do with SQL statements. The difference is that in Aurelius you perform queries at object level, filtering properties and associations. Most classes you need to use for querying are declared in unit Aurelius.Criteria.Base.

7.1 Creating Queries

Queries are represented by an instance of TCriteria object. To execute queries, you just create an instance of TCriteria object, use its methods to add filtering, ordering, projections, etc., and then call List method to execute the query and retrieve results.

Create a new query (TCriteria instance)

Use either Find<T>, CreateCriteria<T> or CreateCriteria method of a TObjectManager instance to create a new query instance. You must always define the class which you want to search objects for:

```
MyCriteria := Manager1.CreateCriteria(TCustomer);
```

or the recommended generic version, which will return a TCriteria<T> object:

```
MyCriteria := Manager1.Find<TCustomer>;
MyCriteria := Manager1.CreateCriteria<TCustomer>;
```

Memory management

One important thing you should know: the TCriteria object instance is automatically destroyed when you retrieve query results, either using List, ListValues, UniqueResult or UniqueValue methods. This is done this way so it's easier for you to use the fluent interface, so you don't need to keep instances to objects in variables and destroy them.

So be aware that you **don't need** to destroy the TCritera object you created using CreateCriteria or Find, unless for some reason you don't retrieve the query results.

If you don't want this behavior to apply and you want to take full control over the TCriteria lifecycle (for example, you want to keep TCriteria alive for some time to add more filters programatically), you can set TCriteria.AutoDestroy property to false (it's true by default). This way TCriteria will not be destroyed automatically and you **must** destroy

it at some point:

```
MyCriteria := Manager1.CreateCriteria(TCustomer);
MyCriteria.AutoDestroy := false;
// You MUST destroy MyCriteria eventually, even after retrieving results
```

7.2 Fluent Interface

The criteria objects you create implement a fluent interface. This means that most methods in the class will return an instance of the object itself. This is just a easier way to build your queries.

So instead of building the query like this:

```
var
  Results: TObjectList<TCustomer>;
  Criteria: TCriteria<TCustomer>;
  Filter: TCustomCriterion;
begin
  Criteria := Manager1.Find<TCustomer>;
  Filter := TExpression.Eq('Name', 'Mia Rosenbaum');
  Criteria.Add(Filter);
  Results := Criteria.List;
```

You can simply write it this way:

```
var
  Results: TObjectList<TCustomer>;
begin
  Results := Manager1.Find<TCustomer>
    .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
    .List;
```

Almost all the examples in this chapter uses the fluent interface so you can fully understand how to use it.

7.3 Retrieving Results

Usually query results are a list of objects of an specified class. You usually call List or List<T> methods to retrieve an object list, or Open to get a fetch-on-demand cursor. If you use a list, this will retrieve you a TList<T> object with all the queries objects. If you are sure your query will return a single value, use UniqueResult (or UniqueValue for projections), which will return a single instance of the object.

It's also important to know how memory management is performed with the queried objects, so you properly know when you need to destroy the retrieved results, and when you don't. Also, you don't need to destroy the query you created using CreateCriteria/ Find, it's automatically destroyed when you query the results.

The following topics describe different ways of retrieving the results of a query:

7.3.1 Retrieving an Object List

After building your query, you can use List method to retrieve filtered/ordered objects. The method to be used depends on how you created your TCriteria object, it could be List or List<T>. The result type will always be a TList<T> where T is the class you are filtering.

If you created the criteria using non-generic CreateCriteria method, you will need to call List<T> method.

```
var
  Results: TList<TCustomer>;
  MyCriteria: TCriteria;
begin
  MyCriteria := ObjectManager1.CreateCriteria(TCustomer);
  // <snip> Build the query
  // Retrieve results
  Results := MyCriteria.List<TCustomer>;
```

If you created the generic criteria using Find<T> or CreateCriteria<T> method, just call List method and it will return the correct object list:

```
var
  Results: TList<TCustomer>;
  MyCriteria: TCriteria<TCustomer>;
begin
  MyCriteria := ObjectManager1.CreateCriteria<TCustomer>;
  // <snip> Build the query
  // Retrieve results
  Results := MyCriteria.List;
```

Using this approach, a query will be executed, all objects will be fetched from the database, connection will be closed and a newly created TList<T> object will be returned with all fetched objects. You must later destroy the TList<T> object.

7.3.2 Unique Result

If you are sure your query will return a single value, use UniqueResult instead (or UniqueResult<T> for non-generic criteria). Instead of a TList<T>, it will just return an instance of T object:

```
var
  UniqueCustomer: TCustomer;
  MyCriteria: TCriteria<TCustomer>;
begin
  MyCriteria := ObjectManager1.Find<TCustomer>;
  // <snip> Build the query
  // Retrieve the single result
  UniqueCustomer := MyCriteria.UniqueResult;
```

If the query returns no objects, then UniqueResult will return nil. If the query returns more than one different object, an exception will be raised.
Note that if the query returns more than one record, but all records relate to the same object, then no exception will be raised, and the unique object will be returned.

7.3.3 Fetching Objects Using Cursor

Alternatively to retrieve an object list, you can get results by using a cursor. With this approach, Aurelius executes a query in the database and returns a cursor for you to fetch objects on demand. In this case, the query will remain open until you destroy the cursor. While this approach has the advantage to keeping a database connection alive, it takes advantage of fetch-on-demand features of the underlying component set you are using, allowing you to get initial results without having to fetch all the objects returned. You don't even need to fetch all results, you can close the cursor before it. Cursor can also be used in TAureliusDataset to make it more responsive to visual controls like DB Grids.

To obtain a cursor, use the Open method:

```
var
  Results: TList<TCustomer>;
  MyCriteria: TCriteria<TCustomer>;
  Cursor: TCriteriaCursor;
  FetchedCustomer: TCustomer;
begin
  MyCriteria := ObjectManager1.Find<TCustomer>;
  // <snip> Build the query
  // Retrieve results
  Cursor := MyCriteria.Open;
  try
    while Cursor.Next do
    begin
      FetchedCustomer := Cursor.Get<TCustomer>;
      // Do something with FetchedCustomer
    end;
  finally
    Cursor.Free;
  end;
```

The Open method returns a TCriteriaCursor object which must be later destroyed. The underlying TCriteria object (MyCriteria variable in the example above) is automatically destroyed when cursor is destroyed.
The TCriteriaCursor object is declared as following:

```
TCriteriaCursor = class abstract(TInterfacedObject, ICriteriaCursor)
public
  function Next: boolean; virtual; abstract;
  function Fetch: TObject; virtual; abstract;
  function BaseClass: TClass; virtual; abstract;
  function Get<T: class>: T;
end;
```

Next method increases cursor position. If result is true, then the new position is valid

and there is an object to fetch. If result is false, there are no more objects to be fetched, and cursor must be destroyed. It's important to note that when the cursor is open, it remains in an undefined position. You **must** call Next method first, before fetching any object. If the very Next call returns false, it means the cursor has no records.

Fetch method is used to retrieve the object in the current cursor position. If Next was never called, or if the result of last Next call was false, Fetch will return unpredictable values. Never call Fetch in such situation.

Get<T> method is just a strong-typed version of Fetch method.

BaseClass method returns the base class used in the criteria query. In the example above, base class would be TCustomer.

As you might have noticed in the TCriteriaCursor declaration above, it implements an ICriteriaCursor interface. That interface is defined as following:

```
ICriteriaCursor = interface
  function Next: boolean;
  function Fetch: TObject;
  function BaseClass: TClass;
end;
```

You can use such interface instead of using TCriteriaCursor, so you don't need to worry about destroying the cursor object - Delphi reference counting will take care of it. The only difference is that ICriteriaCursor doesn't declare the Get<T> method, so you would have to use Fetch method. The code below illustrates how to use ICriteriaCursor:

```
var
  Results: TList<TCustomer>;
  MyCriteria: TCriteria<TCustomer>;
  Cursor: ICriteriaCursor;
  FetchedCustomer: TCustomer;
begin
  MyCriteria := ObjectManager1.Find<TCustomer>;
  // <snip> Build the query
  // Retrieve results
  Cursor := MyCriteria.Open;
  while Cursor.Next do
  begin
    FetchedCustomer := TCustomer(Cursor.Fetch);
    // Do something with FetchedCustomer
  end;
  // No need to destroy cursor
```

7.3.4 Results with Projections

If you added projections to your query, the results will not be entity objects anymore, but instead an special object type that holds a list of values. For example, if you use sum and grouping in your orders, you will not receive a list of TOrder objects anymore, but instead a list of values for the sum results and grouping name.

If that's the case, you should use either:

- ListValues method (if you want to retrieve an object list. This is the equivalent of List method for entity objects).
- UniqueValue method (if you want to retrieve an unique value. This is the equivalent of UniqueResult method for entity objects).
- Open method to retrieve results using a cursor. In this case, the method is the same for either projected or non-projected queries. The only different is the type of object that will be returned.

When using queries with projections, the object returned is a TCriteriaResult object. The TCriteriaResult is an object that has a default property Values which you can use to retrieve the values using an index:

```
var
  Results: TObjectList<TCriteriaResult>;
  MyCriteria: TCriteria<TCustomer>;
  FirstValueInFirstRecord: Variant;
begin
  MyCriteria := ObjectManager1.CreateCriteria<TCustomer>;
  // <snip> Build the query and add projections to it
  // Retrieve projected results
  Results := MyCriteria.ListValues;
  FirstValueInFirstRecord := Results[0].Values[0];
```

Alternatively, you can find the value by name. The name is specified by the alias of projections. If no alias is specified, an internal autonumerated name is used.

```
var
  Results: TObjectList<TCriteriaResult>;
begin
  Results := Manager.CreateCriteria<TTC_Estimate>
    .CreateAlias('Customer', 'c')
    .Select(TProjections.ProjectionList
      .Add(TProjections.Sum('EstimateNo').As_('EstimateSum'))
      .Add(TProjections.Group('c.Name'))
     )
    .Add(TExpression.Like('c.Name', 'M%'))
    .AddOrder(TOrder.Asc('EstimateSum'))
    .ListValues;

  EstimateSum := Results[0].Values['EstimateSum'];
  CustomerName := Results[0].Values[1]; // no alias specified for c.Name
end;
```

If the property doesn't exist, an error is raised. TCriteriaResult also has an additional HasProp method for you to check if the specified value exists. The following code contains the TCriteriaResult public methods and properties.

```
TCriteriaResult = class
public
  function HasProp(PropName: string): boolean;
```

```
  function HasProp(PropName: string): boolean;
  property PropNames[Index: integer]: string read GetPropName;
  property Values[Index: integer]: Variant read GetValue; default;
  property Values[PropName: string]: Variant read GetPropValue; default;
  property Count: integer read GetCount;
end;
```

It's important to note that TCriteriaResult objects are not managed by the TObjectManager, so the retrieved objects **must** be destroyed. When using ListValues method to retrieve the results, the returned list is a TObjectList<T> object that already has its OwnsObjects property set to true. So destroyed the list should be enough. When using UniqueValue or Open methods, you must be sure to destroy the TCriteriaResult objects.

7.4 Filtering Results

You can narrow the result of your query by adding filter expressions to your query. This is similar to the WHERE clause in an SQL statement. Any expression object descends from TCustomCriterion, and you can use Add or Where methods to add such objects to the query:

```
Results := Manager1.Find<TCustomer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

You can add more than one expression to the query. The expression will be combined with an "and" operator, which means only objects which satisfies all conditions will be returned:

```
Results := Manager1.Find<TCustomer>
  .Add(TExpression.Eq('Country', 'US'))
  .Add(TExpression.Eq('Age', 30))
  .List;
```

The Where method is exactly the same as the method Add, it's just included as an option to Add method so it looks better in some queries:

```
Results := Manager1.Find<TCustomer>
  .Where(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

In all the examples above, the TCustomCriterion objects added to the criteria were created using the TExpression factory class. The TExpression is just a helper class with several class methods (Equal, GreaterThan, etc.) that you can use to easily create TCustomCriterion instances.

Creating Expressions Using TExpression
Logical Operators
TLinq instead of TExpression
Associations

7.4.1 Creating Expressions Using TExpression

To filter results you must add TCustomCriterion objects to the query object. The TCustomCriterion objects just represent a conditional expression that the object must satisfy to be included in the result. To create such objects, you can use the TExpression factory class. It's declared in Aurelius.Criteria.Expression unit:

```
uses Aurelius.Criteria.Expression
```

The TExpression is just a helper class with several class methods (Equal, GreaterThan, etc.) that you can use to easily create TCustomCriterion instances. For example, the following lines produce the same object:

```
Criterion := TSimpleExpression.Create(TPropertyProjection.Create('Age'),
30, eoGreater;)
Criterion := TExpression.GreaterThan('Age', 30);
```

Which is a restriction to list only objects which property Age value is greater than 30.

Note that in all the methods listed here, the method can receive a string (representing a property name) or a projection. See TProjections.Prop for more details.

You can use TExpression to create the following conditions:

Equals
Greater Than
Greater Than or Equals To
Less Than
Less Than Or Equals To
Like
IsNull
IsNotNull
Identifier Equals

7.4.1.1 Equals

Retrieves a condition where the specified property (or projection) value must be equals to the specified value or projection. You can use Equals or Eq method, they both do the same.

Example - Return customers where Name property is equal to "Mia Rosenbaum".

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;
```

7.4.1.2 Greater Than

Retrieves a condition where the specified property (or projection) value must be greater than the specified value. You can use either GreatherThan or Gt method, they both do the same.

Example - Return customers where Birthday property is greater than 10-10-1981 and less

than 02-02-1986.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.GreaterThan('Birthday', EncodeDate(1981, 10, 10)))
  .Add(TExpression.LessThan('Birthday', EncodeDate(1986, 2, 2)))
  .List;
```

7.4.1.3 Greater Than or Equals To

Retrieves a condition where the specified property (or projection) value must be greater than or equals to the specified value. You can use either GreaterOrEqual or Ge method, they both do the same.

Example - Return customers where Birthday property is greater than or equals to 10-10-1981 and less than or equals to 02-02-1986.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.GreaterOrEqual('Birthday', EncodeDate(1981, 10, 10)))
  .Add(TExpression.LessOrEqual('Birthday', EncodeDate(1986, 2, 2)))
  .List;
```

7.4.1.4 Less Than

Retrieves a condition where the specified property (or projection) value must be less than the specified value. You can use either LessThan or Lt method, they both do the same.

Example - Return customers where Birthday property is greater than 10-10-1981 and less than 02-02-1986.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.GreaterThan('Birthday', EncodeDate(1981, 10, 10)))
  .Add(TExpression.LessThan('Birthday', EncodeDate(1986, 2, 2)))
  .List;
```

7.4.1.5 Less Than Or Equals To

Retrieves a condition where the specified property (or projection) value must be less than or equals to the specified value. You can use either LessOrEqual or Le method, they both do the same.

Example - Return customers where Birthday property is greater than or equals to 10-10-1981 and less than or equals to 02-02-1986.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.GreaterOrEqual('Birthday', EncodeDate(1981, 10, 10)))
  .Add(TExpression.LessOrEqual('Birthday', EncodeDate(1986, 2, 2)))
  .List;
```

7.4.1.6 Like

Retrieves a condition where the specified property (or projection) value contains the text specified. It's equivalente to the LIKE operator in SQL statements. You must specify the wildchar % in the value condition.

Example - Return customers where Sex property is not null, and Name starts with "M".

```
Results := Manager.Find<TCustomer>
  .Where(TExpression
    .And_(
      TExpression.IsNotNull('Sex'),
      TExpression.Like('Name', 'M%')
      )
  )
  .List;
```

7.4.1.7 IsNull

Retrieves a condition where the specified property (or projection) contains a null value.

Example - Return customers where Sex property is female, or Sex property is null.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression
    .Or_(
      TExpression.Eq('Sex', tsFemale),
      TExpression.IsNull('Sex')
      )
  )
  .List;
```

7.4.1.8 IsNotNull

Retrieves a condition where the specified property (or projection) does not contain a null value.

Example - Return customers where Sex property is not null, and Name starts with "M".

```
Results := Manager.Find<TCustomer>
  .Where(TExpression
    .And_(
      TExpression.IsNotNull('Sex'),
      TExpression.Like('Name', 'M%')
      )
  )
  .List;
```

7.4.1.9 Identifier Equals

Retrieves a condition where the identifier of the specified class is equal to a value. This is very similar to using Equals, but in this case you don't need to specify the property name - Aurelius already knows that you are referring to the Id. Also, for composite id's, you can provide an array of variant for all the values of the composite id, the query will compare all table columns belonging to the composite id with all values provided in the array of variant.

Example - Return customer where identifier is equal to 1

```
Customer := Manager.Find<TCustomer>
  .Add(TExpression.IdEq(1))
  .UniqueResult;
```

Example - Using composite id: return patient where last name is "Smith" and first name is "John" (considering that the id of this class is made of properties LastName and FirstName:

```
var
  Id: Variant;
  Person: TPerson;
begin
  Id := VarArrayCreate([0, 1], varVariant);
  Id[0] := 'Smith'; // last name
  Id[1] := 'John'; // first name
  Person := Manager.CreateCriteria<TPerson>
    .Add(TExpression.IdEq(Id))
    .UniqueResult;
```

7.4.1.10 Sql Expression

Creates a custom SQL expression condition. Use this for total flexibility, if you might fall into a situation where regular query filters provided by Aurelius are not enough. The SQL you provide in this expression must conform with the underlying database syntax. Aurelius doesn't perform any syntax conversion (except aliases and parameters, see below).

Example - Return customer where database column NAME is equal to "Mia Rosenbaum"

```
Results := Manager.Find<TCustomer>
  .Add(TLinq.Sql('A.CUSTOMER_NAME = ''Mia Rosenbaum'''))
  .List;
```

Aliases
Note that since the SQL expression will be just injected in the SQL statement, you must be sure it will work. In the example above, the exact alias name ("A") and field name ("CUSTOMER_NAME") needed to be included.
In order to prevent you from knowing which alias to use (which is especially tricky when Aurelius need to use joins in SQL statement), you can use placeholders (aliases) between curly brackets. Write the name of the property inside curly brackets and Aurelius will translate it into the proper alias.fieldname format according to eh SQL. The following example does the same as the previous one, but instead of using the field name directly, you use the name of property TCustomer.Name.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.Sql('{Name} = ''Mia Rosenbaum'''))
  .List;
```

When querying associations, you can also prefix the property name with the alias of the association (see how to query Associations):

```
Results := Manager.Find<TCustomer>
  .CreateAlias('Country', 'c')
  .Add(TExpression.Sql('{c.Name} = ''United States'''))
  .List;
```

Note that when you use subcriteria, the context of the property in curly brackets will be the subcriteria class. The following query is equivalent to the previous one:

```
Results := Manager.Find<TCustomer>
  .SubCriteria('Country')
    .Add(TExpression.Sql('{Name} = ''United States'''))
  .List<TTC_Customer>;
```

Parameters
You can also use parameters in the Sql projection, to avoid having to use specific database syntax for literals. For example, if you want to compare a field with a date value, you would need to specify a date literal with a syntax that is compatible with the database SQL syntax. To avoid this, Aurelius allows you to use parameters in Sql expression. You can use up to two parameters in each expression. The parameters must be indicated by a question mark ("?") and the type of parameters must be provided in a generic parameter for the Sql method:

Example - using one parameter of type TSex

```
Results := Manager.Find<TCustomer>
  .Add(TLinq.Sql<TSex>('{Sex} IN (?)', TSex.tsFemale))
  .List;
```

Example - using two parameters of type TDate

```
Results := Manager.Find<TEstimate>
  .Where(
     TLinq.Sql<TDate, TDate>(
        '{IssueDate} IS NULL OR (({IssueDate} > ?) AND ({IssueDate}
< ?))',
       EncodeDate(1999, 2, 10), EncodeDate(2000, 8, 30))
    )
  .List;
```

7.4.1.11 Comparing Projections

You can also compare projections. The same methods used to compare values like Equals, GreaterThan, LessThan, etc, have overloaded versions that receives two TProjection objects that are to be compared.
This gives you great flexibility since you can create many different types of projections and compare them. This is often used to compare two fields.

Example - Return Orders where cancelation date is greater than shipping date:

```
Results := Manager.Find<TOrder>
  .Add(TExpression.GreaterThan(TProjections.Prop('CancelationDate'),
TProjections.Prop('ShippingDate')))
  .List;
```

7.4.2 Logical Operators

When adding expressions to a TCriteria object, all expressions will be combined with "AND" operator in the SQL statement. In case you want to use "OR" operator, you can use TExpression.Or_ method. You can combine multiple Or_ and And_ methods to create nested expressions. Each object created with Or_ or And_ methods are also TCustomCriterion descendants that can be combined together.

In addition, you can use Not_ method to use an inverse of current condition and filter objects that do not apply to the specified condition.

If you want to use several "and" and "or" operators, is strongly recommended that you use TLinq class (instead of TExpression) which allows easier way to build the nested expressions.

Some examples:

Or_ example: Retrieve customers where Sex is female or is null.

```
Results := Manager.Find<TCustomer>
  .Add(TExpression
    .Or_(
      TExpression.Eq('Sex', tsFemale),
      TExpression.IsNull('Sex')
      )
  )
  .List;
```

And_ example: Retrieve estimates where IssueDate is null, or IssueDate is between 10-Fev-1999 and 30-Aug-2000.

```
Results := Manager.Find<TEstimate>
  .Where(
    TExpression.Or_(
      TExpression.IsNull('IssueDate'),
      TExpression.And_(
        TExpression.GreaterThan('IssueDate', EncodeDate(1999, 2, 10)),
        TExpression.LessThan('IssueDate', EncodeDate(2000, 8, 30))
        )
      )
    )
```

Not_ example: Retrieve customers where Sex is not female (i.e, can be male or null):

```
Results := Manager.CreateCriteria<TCustomer>
  .Add(TExpression
    .Not_(TExpression.Eq('Sex', tsFemale))
      )
  )
  .List;
```

7.4.3 TLinq instead of TExpression

Using logical operators with TExpression.Or_ and TExpression.And_ method can make the query become very hard to understand if you use too many nested methods. As an alternative, you can use TLinq class instead of TExpression. It has exactly the same methods of TExpression, so creating expression with TLinq is done the same way as creating expression using TExpression. The only difference is that TLinq doesn't have methods And_ and Or_, instead you use pascal "and" and "or" operators directly in the expressions created by TLinq. You can also nest the expressions with parenthesis. TLinq class is declared in unit Aurelius.Criteria.Linq:

```
uses Aurelius.Criteria.Linq
```

So instead of writing the query like this using TExpression:

```
Results := Manager.Find<TEstimate>
  .Where(
    TExpression.Or_(
      TExpression.IsNull('IssueDate'),
      TExpression.And_(
        TExpression.GreaterThan('IssueDate', EncodeDate(1999, 2, 10)),
        TExpression.LessThan('IssueDate', EncodeDate(2000, 8, 30))
        )
      )
    )
  .List;
```

You can write it like this using TLinq:

```
Results := Manager.Find<TTC_Estimate>
  .Add(
    TLinq.IsNull('IssueDate') or
    (TLinq.GreaterThan('IssueDate', EncodeDate(1999, 2, 10))
     and TLinq.LessThan('IssueDate', EncodeDate(2000, 8, 30))
    )
  )
  .List;
```

Not operator is also supported:

```
Results := Manager.Find<TAnimal>
  .Add(not TLinq.Like('Name', 'A%'))
  .List;
```

7.4.4 Associations

You can add condition expressions to associations of the class being queried. For example, you can retrieve invoices filtered by the name of invoice customer.

To add a condition for an association, you have two options: use subcriteria or aliases.

Using SubCriteria

You can create a sub-criteria which related to the association being filtered, using SubCriteria method of the TCriteria object itself. It returns a new TCriteria object which context is the association class, not the main class being queried.

```
Results := Manager.Find<TInvoice>
  .SubCriteria('Customer')
    .Add(
      TExpression.Like('Name', 'M%')
    )
    .List<TInvoice>;
```

In the example above the class TInvoice has a property Customer which is an association to the TCustomer class. The filter "Name = 'M%'" is applied to the customer, not the invoice. SubCriteria method is being called and receives "Customer" parameter, which is the name of associated property. This returns a new TCriteria object. The expressions added to it related to TCustomer class, that's why 'Name' refers to the TCustomer.Name property, not TInvoice.Name (if that ever existed).

Note that SubCriteria method returns a TCriteria method (the non-generic version). That's why we need to call List<TInvoice> method (not just List).

You can have nested SubCriteria calls, there is not a level limit for it. In the example below, the query returns all estimates for which the country of the customer is "United States".

```
Results := Manager.CreateCriteria<TEstimate>
  .SubCriteria('Customer')
    .SubCriteria('Country')
      .Add(TExpression.Eq('Name', 'United States'))
  .List<TEstimate>;
```

Using aliases

You can alternatively give an alias for an association. This way you don't need to use SubCriteria to reference properties in associations, you can just prefix the property name with the alias.

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Add(TExpression.Like('c.Name', 'M%'))
  .List;
```

Calling CreateAlias does not return a new TCriteria instance, but instead it returns the original TCriteria. So the expression context is still the original class (in the example above, TEstimate). Thus, to reference a Customer property the "c" alias prefix was needed. Note that since the original TCriteria<TEstimate> object is being used, you can call List method (instead of TList<T>).

Just like SubCriteria calls, you can also use nested CreateAlias methods, by settings aliases for associations of associations. It's important to note that the context in the fluent interface is always the original TCriteria class:

```
Results := Manager.CreateCriteria<TEstimate>
  .CreateAlias('Customer', 'ct')
  .CreateAlias('ct.Country', 'cn')
  .Add(TExpression.Eq('cn.Name', 'United States'))
  .List;
```

Mixing SubCriteria and aliases

You can safely mix SubCriteria and CreateAlias calls in the same query:

```
Results := Manager.CreateCriteria<TEstimate>
  .SubCriteria('Customer')
    .CreateAlias('Country', 'cn')
    .Add(TExpression.Eq('cn.Name', 'United States'))
  .List<TEstimate>;
```

7.5 Ordering Results

You can order the results by any property of the class being query, or by a property of an association of the class. Just use AddOrder method of the TCriteria object. You must define name of the property (or projection) being ordered, and if the order is ascending or descending. See examples below:

Retrieve customers ordered by Name

```
Results := Manager.CreateCriteria<TCustomer>
  .Add(TExpression.Eq('Name', 'M%'))
  .AddOrder(TOrder.Asc('Name'))
  .List;
```

You can also use association aliases in orderings.

Retrieve all estimates which IssueDate is not null, ordered by customer name in descending order.

```
Results := Manager.CreateCriteria<TEstimate>
  .CreateAlias('Customer', 'c')
  .Add(TExpression.IsNotNull('IssueDate'))
  .AddOrder(TOrder.Desc('c.Name'))
  .List;
```

If you need to order by complex expressions, it's recommended that you use a Alias projection for it. In the example below, the order refers to the EstimateSum alias, which is just an alias for the sum expression

```
Results := Manager.CreateCriteria<TTC_Estimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo').As_('EstimateSum'))
    .Add(TProjections.Group('c.Name'))
    )
  .Add(TExpression.Like('c.Name', 'M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

7.6 Projections

You can make even more advanced queries in Aurelius by using projections. For example, instead of selecting pure object instances (TCustomer for example) you can perform grouping, select sum, average, among others. There is a formal definition for projection, but you can think of a projection just as an expression that returns a value, for example, a call to Sum function, a literal, or the value of a property. You can even use projections in expressions.

Any projection object descends from TProjection class. To specify a projection in the query, use the SetProjections or Select method.

The example below calculates the sum of all estimates where the customer name beings with "M".

```
Value := Manager.CreateCriteria<TEstimate>
  .SetProjections(TProjections.Sum('EstimateNo'))
  .CreateAlias('Customer', 'c')
  .Add(TExpression.Like('c.Name', 'M%'))
  .UniqueValue;
```

You can only have a single projection specified for the query. If you call SetProjections or Select method twice in a single query, it will replace the projection specified in the previous call. If you want to specify multiple projections, using a projection list:

Query over estimates, retrieving the sum of EstimateNo, grouped by customer name.

```
Results := Manager.CreateCriteria<TEstimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo'))
    .Add(TProjections.Group('c.Name'))
    )
  .ListValues;
```

Note that when using projections, the query does not return instances of the queried class (TEstimate in example above). Instead, it returns a list of TCriteriaResult objects, which you can use to retrieve the projection values. See more in Retrieving Results section.

The Select method is exactly the same as the method SetProjections, it's just included as an option so it looks better in some queries.

In all the examples above, the TProjection objects added to the criteria were created using the TProjections factory class. The TProjections is just a helper class with several class methods that you can use to easily create TProjection instances.

Creating Projections Using TProjections

7.6.1 Creating Projections Using TProjections

Any projection you want to use is a TProjection object. To create such objects, you can use the TProjections factory class. It's declared in Aurelius.Criteria.Projections unit.

```
uses Aurelius.Criteria.Projections
```

The TProjections class is just a helper class with several class methods (Sum, Group, etc.) that you can use to easily create TProjection instances. For example, the following lines produce the same object:

```
Projection := TAggregateProjection.Create('sum',
TPropertyProjection.Create('Total'));
Projection := TProjections.Sum('Total');
```

You can use TProjections to create the following projections:

Aggregated Functions
Prop
Group
Condition
Literal<T>
Value<T>
ProjectionList
Alias
Sql Projection

7.6.1.1 Aggregated Functions

There are several methods in TProjections class that create a projection that represents an aggregated function over a property value (or a projection). Available methods are:

Sum: Calculated the sum of values
Min: Retrieves the minimum value
Max: Retrieves the maximum value
Avg: Calculates the average of all values
Count: Retrieves the number of objects the satisfy the condition

Calculates the sum of all estimates where the customer name beings with "M".

```
Value := Manager.Find<TEstimate>
  .SetProjections(TProjections.Sum('EstimateNo'))
  .CreateAlias('Customer', 'c')
  .Add(TExpression.Like('c.Name', 'M%'))
  .UniqueValue;
```

7.6.1.2 Prop

Creates a projection that represents the value of a property. In most cases you won't use this projection because there are overloads for all methods in TExpression, TProjection and TLinq class that accepts a string instead of a projection. The string represents a property name and internally all it does is to create a property projection using Prop method.

The example below illustrates how Prop method can be used.

Retrieve the name of the customers ordered by the Name

```
Results := Manager.Find<TCustomer>
  .SetProjections(TProjections.Prop('Name'))
  .AddOrder(TOrder.Asc(TProjections.Prop('Name')))
  .ListValues;
```

As stated above, the following queries are equivalent:

```
Results := Manager.Find<TCustomer>
  .Add(TExpression.Eq('Name', 'Mia Rosenbaum'))
  .List;

Results := Manager.Find<TCustomer>
  .Add(TExpression.Eq(TProjections.Prop('Name'), 'Mia Rosenbaum'))
  .List;
```

7.6.1.3 Group

Creates a projection that represents a group. This is similar to the GROUP BY clause in an SQL statement, but the difference is that you don't need to set a Group By anywhere - you just add a grouped projection to the projection list and Aurelius groups is automatically.

The query below retrieves the sum of EstimateNo grouped by customer name. The projected values are the EstimateNo sum, and the customer name. Since the customer name is already one of the selected projections and it's grouped, that's all you need - you don't have to add the customer name in some sort of Group By section.

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo'))
    .Add(TProjections.Group('c.Name'))
    )
  .ListValues;
```

7.6.1.4 Condition

Creates a conditional projection. It works as an If..Then..Else clause, and it's equivalente to the "CASE..WHEN..ELSE" expression in SQL.

Retrieves the customer name and a string value representing the customer sex. If sex is

tsFemale, return "Female", if it's tsMale return "Male". If it's null, then return "Null".

```
Results := Manager.Find<TCustomer>
  .SetProjections(TProjections.ProjectionList
    .Add(TProjections.Prop('Name'))
    .Add(TProjections.Condition(
      TExpression.IsNull('Sex'),
      TProjections.Literal<string>('Null'),
      TProjections.Condition(
        TExpression.Eq('Sex', tsMale),
        TProjections.Literal<string>('Male'),
        TProjections.Literal<string>('Female')
        )
      )
    )
  )
  .ListValues;
```

7.6.1.5 Literal<T>

Creates a constant projection. It's just a literal value of scalar type T. Aurelius automatically translates the literal into the database syntax. The Literal<T> method is different from Value<T> in the sense that literals are declared directly in the SQL statement, while values are declared as parameters and the value is set in the parameter value.

Retrieves some literal values

```
Results := Manager.Find<TCustomer>
  .SetProjections(TProjections.ProjectionList
    .Add(TProjections.Literal<string>('Test'))
    .Add(TProjections.Literal<Currency>(1.53))
    .Add(TProjections.Literal<double>(3.14e-2))
    .Add(TProjections.Literal<integer>(100))
    .Add(TProjections.Literal<TDateTime>(Date1))
  )
  .ListValues;
```

Another example using Condition projection:

```
Results := Manager.CreateCriteria<TCustomer>
  .SetProjections(TProjections.ProjectionList
    .Add(TProjections.Prop('Name'))
    .Add(TProjections.Condition(
      TExpression.IsNull('Sex'),
      TProjections.Literal<string>('Null'),
      TProjections.Condition(
        TExpression.Eq('Sex', tsMale),
        TProjections.Literal<string>('Male'),
        TProjections.Literal<string>('Female')
        )
      )
    )
  )
  .ListValues;
```

7.6.1.6 Value<T>

Creates a constant projection. It's just a value of scalar type T. It works similar to Literal<T> method, the difference is that literals are declared directly in the SQL statement, while values are declared as parameters and the value is set in the parameter value.

7.6.1.7 ProjectionList

Retrieves a list of projections. It's used when setting the projection of a query using Select or SetProjections method. Since only one projection is allowed per query, you define more than one projections by adding a projection list. This method returns a TProjectionList object which defines the Add method that you use to add projections to the list.

Creates a projection list with two proejctions: Sum of EstimateNo and Customer Name.

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo'))
    .Add(TProjections.Group('c.Name'))
    )
  .ListValues;
```

7.6.1.8 Alias

Associates an alias to a projection so it can be referenced in other parts of criteria. Currently only orderings can refer to aliased projections. It's useful when you need to use complex expressions in the order by clause - some databases do not accept such expressions, so you can just reference an existing projection in the query, as illustrated below.

Retrieve all estimates grouped by customer name, ordered by the sum of estimates for each customer.

```
Results := Manager.Find<TTC_Estimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Alias(TProjections.Sum('EstimateNo'),
'EstimateSum'))
    .Add(TProjections.Group('c.Name'))
    )
  .Add(TExpression.Like('c.Name', 'M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

Alternatively you can create aliased projections using the As_ method of any simple projection. This query does the same as the previous query:

```
Results := Manager.Find<TTC_Estimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo').As_('EstimateSum'))
    .Add(TProjections.Group('c.Name'))
    )
  .Add(TExpression.Like('c.Name', 'M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

7.6.1.9 Sql Projection

Creates a projection using a custom SQL expression. Use this for total flexibility, if you might fall into a situation where regular projections provided by Aurelius are not enough. The SQL you provide in this expression must conform with the underlying database syntax. Aurelius doesn't perform any syntax conversion (except aliases, see below).

Example - Return specific projections

```
Results := Manager.CreateCriteria<TCustomer>
  .CreateAlias('Country', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Prop('Id').As_('Id'))
    .Add(TProjections.Sql<string>('A.CUSTOMER_NAME').As_('CustName'))
    .Add(TProjections.Sql<double>('{id} * 2').As_('DoubleId'))
    .Add(TProjections.Sql<integer>('{c.id} * 2').As_('DoubleCountryId'))
  )
  .ListValues;
```

Note that since the SQL expression will be just injected in the SQL statement, you must be sure it will work. In the example above, the exact alias name ("A") and field name ("CUSTOMER_NAME") needed to be included in projection "CustName".
In order to prevent you from knowing which alias to use (which is especially tricky when Aurelius need to use joins in SQL statement), you can use placeholders (aliases) between curly brackets. Write the name of the property inside curly brackets and Aurelius will translate it into the proper alias.fieldname format according to eh SQL. In the previous example, projections "DoubleId" and "DoubleCountryId" use placeholderrs that will be replaced by the proper "Alias.ColumnName" syntax corresponding to the referenced property. "{id}" refers to property TCustomer.Id, while "{c.Id}" refers to property

TCustomer.Country.Id".

The generic parameter in the Sql method must indicate the type returned by the Sql projection.

7.7 Polymorphism

Since Aurelius supports inheritance using different inheritance strategies, queries are also polymorphic. It means that if you query over a specified class, you might receive objects of that class, or even descendants of that class.

For example, suppose you have a class hierarchy this way:

```
TAnimal = class
TBird = class(TAnimal);
TMammal = class(TAnimal);
TDog = class(TMammal);
TCat = class(TMammal);
```

when you perform a query like this:

```
Results := Manager.Find<TMammal>
  .Add(TExpression.Like('Name', 'T%'))
  .List;
```

You are asking for all mammals which Name begins with "T". This means all mammals, dogs and cats. So in the resulted object list, you might receive instances of TMammal, TDog or TCat classes. Aurelius does it automatically for you, regardless on the inheritance strategy, i.e. if all classes are being saved in the same table or each class is being saved in a different table. Aurelius will be sure to filter out records representing animals and birds, and retrieve only the mammals (including dogs and cats).

You can safely rely on polymorphism with Aurelius in every query, and also of course, when saving and updating objects.

7.8 Paging Results

Aurelius provides methods the allows you to limit query results at server level. It's the equivalent of "SELECT TOP" or "SELECT..LIMIT" that some databases use (note this is just an analogy, TMS Aurelius will make sure to build the proper SQL statement for each database according to the supported syntax).

You can limit the number of objects retrieved by using the Take method of TCriteria object:

```
Results := Manager.Find<TCustomer>
  .AddOrder(TOrder.Asc('Name'))
  .Take(50)
  .List;
```

The previous code will retrieve the first 50 TCustomer objects, ordered by name. Using

Take(0) will return an empty result. Using Take(-1) is equivalent to not using Take method at all, meaning all records will be returned. Values below -2 (including) are not allowed and might cause errors.

You can skip the first N objects retrieved by using Skip method:

```
Results := Manager.Find<TCustomer>
  .AddOrder(TOrder.Asc('Name'))
  .Skip(10)
  .List;
```

The previous will retrieve customers ordered by name, by will omit the first 10 customers from the list. Using Skip(0) is equivalent to not using Skip method at all, since it means skipping no records. Negative values are not allowed and might cause errors.

Although you can use Skip and Take methods without specifying an order, it often doesn't make sense.

Skip and Take methods are often used for paging results, i.e., returning objects belonging to an specific page. The following code exemplifies how to return objects belonging to the page PageIdx, with PageSize objects in each page:

```
Results := Manager.Find<TCustomer>
  .AddOrder(TOrder.Asc('Name'))
  .Skip(PageIdx * PageSize)
  .Take(PageSize)
  .List;
```

7.9 Removing Duplicated Objects

Sometimes a query might result in duplicated objects. The following query is an example of such queries:

```
Results := Manager.Find<TInvoice>
  .CreateAlias('Items', 'i')
  .Add(TLinq.Eq('i.Price', 20))
  .AddOrder(TOrder.Asc('InvoiceNo'))
  .List;
```

The above criteria will look for all invoices which have any item with price equals to 20. Just like in SQL, this query is doing a "join" between the invoice and invoice items. This means that if an invoice has two or more items with price equals to 20, the same TInvoice object will be returned more than once in the result list.

If that's not what you want, and you just list all invoices matching the specified criteria, without duplicates, just use RemoveDuplicatedEntities to your criteria:

```
Results := Manager.Find<TInvoice>
  .CreateAlias('Items', 'i')
  .Add(TLinq.Eq('i.Price', 20))
```

```
  .AddOrder(TOrder.Asc('InvoiceNo'))
  .RemovingDuplicatedEntities
  .List;
```

And this will bring distinct invoices. This feature is usually useful when you want to filter objects by a criteria applied to many-valued associations, like in the example above, which might return duplicated results.

Please note that the removal of duplicated objects is done at client level by Aurelius framework, not at database level, so performance might be not good with queries that result too many records.

Chapter

Data Binding - TAureliusDataset

8 Data Binding - TAureliusDataset

TMS Aurelius allows you to bind your entity objects to data-aware controls by using a TAureliusDataset component. By using this component you can for example display a list of objects in a TDBGrid, or edit an object property directly through a TDBEdit or a TDBComboBox. TAureliusDataset is declared in unit Aurelius.Bind.Dataset:

```
uses
  {...}, Aurelius.Bind.Dataset;
```

Basic usage is done by these steps:

1. Set the source of data to be associated with the dataset, using SetSourceList method, or a single object, using SetSourceObject
2. Optionally, create a TField for each property/association/sub-property you want to display/edit. If you do not, default fields will be used.
3. Optionally, specifiy a TObjectManager using the Manager property. If you do not, you must manually persist objects to database.

TAureliusDaset is a TDataset descendant, thus it's compatible with all data-aware controls provided by VCL, the Firemonkey live bindings framework and any 3rd-party control/tool that works with TDataset descendants. It also provides most of TDataset functionality, like calculated fields, locate, lookup, filtering, master-detail using nested datasets, among others.

The topics below cover all TAureliusDataset features.

8.1 Providing Objects

To use TAureliusDataset, you must provide to it the objects you want to display/edit. The objects will become the source of data in the dataset.

The following topics describe several different methods you can use to provide objects to the dataset:

8.1.1 Providing an Object List

A very straightforward way to provide objects to the dataset is specifying an external object list where the objects will be retrieved from (and added to).
You do that by using SetSourceList method:

```
var
  People: TList<TPerson>;
begin
  People := Manager.Find<TPerson>.List;
  AureliusDataset1.SetSourceList(People);
```

You can provide any type of generic list to it. You must be responsible for destroying the list object itself, TAureliusDataset will not manage it.
When you insert/delete records in the dataset, objects will be added/removed to the list.

8.1.2 Providing a Single Object

Instead of providing multiple objects, you can alternatively specify a single object.
It's a straightforward way if you intend to use the dataset to just edit a single object.
You must use SetSourceObject method for that:

```
Customer := Manager.Find<TCustomer>(1);
AureliusDataset1.SetSourceObject(Customer);
```

Be aware that TAureliusDataset always works with lists. When you call SetSourceObject, the internal object list is cleared and the specified object is added to it. The internal list then is used as the source list of dataset. This means that even if you use SetSourceObject method, objects might be added to or removed from the internal list, if you call methods like Insert, Append or Delete.

8.1.3 Using Fetch-On-Demand Cursor

You can provide objects to TAureliusDataset by using a query object cursor. This approach is especially useful when returning a large amount of data, since you don't need to load the whole object list first and then provide the whole list to the dataset. Only needed objects are fetched (for example, the objects being displayed in a TDBGrid that is linked to the dataset). Additional objects will only be fetched when needed, i.e, when you scroll down a TDBGrid, or call TDataset.Next method to retrieve the next record.

Note that the advantage of this approach is that it keeps an active connection and an active query to the database until all records are fetched (or dataset is closed).

To use a cursor to provide objects, just call SetSourceCursor method and pass the ICriteriaCursor interface you have obtained when opening a query using a cursor:

```
var
  Cursor: ICriteriaCursor;
begin
  Cursor := Manager.Find<TPerson>.Open;
  AureliusDataset1.SetSourceCursor(Cursor);

  // Or just this single line version:
  AureliusDataset1.SetSourceCursor(Manager.Find<TPerson>.Open);
```

You don't have to destroy the cursor, since it's an interface and is destroyed by reference counting. When the cursor is not needed anymore, dataset will destroy it.

When you call SetSourceCursor, the internal object list is cleared. When new objects are fetched, they are added to the internal list. So, the internal list will increase over time, as you navigate forward in the dataset fetching more records.

8.1.4 Using Criteria for Offline Fetch-On-Demand

Another way to provide objects to TAureliusDataset is providing a TCriteria object to it. Just create a query and pass the TCriteria object using SetSourceCriteria method.

```
var
  Criteria: TCriteria;
begin
  Criteria := Manager.Find<TPerson>;
  AureliusDataset1.SetSourceCriteria(Criteria);

  // Or just this single line version:
  AureliusDataset1.SetSourceCriteria(Manager.Find<TPerson>);
```

In the code above, Aurelius will just execute the query specified by the TCriteria and fill the internal object list with the retrieved objects. This approach is actually not very different than providing an object list to the dataset. The real advantage of it is when you use an overloaded version of SetSourceCriteria that allows paging.

Office fetch-on-demand using paging

SetSourceCriteria method has an overloaded signature that received an integer parameter specifying a page size:

```
  AureliusDataset1.SetSourceCriteria(Manager.Find<TPerson>, 50);
```

It means that the dataset will fetch records on demand, but without needing to keep an active database connection.
When you open a dataset after specifying a page size of 50 as illustrated in the code above, only the first 50 TPerson objects will be fetched from the database, and query will be closed. Internally, TAureliusDataset uses the paging mechanism provided by Take and

Skip methods. If more records are needed (a TDBGrid is scrolled down, or you call TDataset.Next method multiple times, for example), then the dataset will perform another query in the database to retrieve the next 50 TPerson objects in the query.

So, in summary, it's a fetch-on-demand mode where the records are fetched in batches and a new query is executed every time a new batch is needed. The advantage of this approach is that it doesn't retrieve all objects from the database at once, so it's fast to open and navigate, especially with visual controls. Another advantage (when comparing with using cursors, for example) is that it works offline - it doesn't keep an open connection to the database. One disadvantage is that it requires multiple queries to be executed on the server to retrieve all objects.

You don't have to destroy the TCriteria object. The dataset uses it internally to re-execute the query and retrieve a new set of objects. When all records are fetched or the dataset is closed, the TCriteria object is automatically destroyed.

8.2 Internal Object List

TAureliusDataset keeps an internal object list that is sometimes used to hold the objects associated with the dataset records. When you provide an external object list, the internal list is ignored. However, when you use other methods for providing objects, like using cursor (SetSourceCursor), paged TCriteria (SetSourceCriteria), or even a single object (SetSourceObject), then the internal list is used to keep the objects.

When the internal list is used, when new records are inserted or deleted, they are added to and removed from the internal list. When fetch-on-demand modes are used (cursor and criteria), fetched objects are incrementally added to the list. Thus, when you open the dataset you might have 20 objects in the list, when you move the cursor to the end of dataset, you might end up with 100 objects in the list.

So, there might be situations where you need to access such list. TAureliusDataset provides a property InternalList for that. This property is declared as following:

```
property InternalList: IReadOnlyObjectList;
```

The list is accessible through a IReadOnlyObjectList, so you can't modify it (unless, of course, indirectly by using the TDataset itself). The IReadOnlyObjectList has the following methods:

```
IReadOnlyObjectList = interface
  function Count: integer;
  function Item(I: integer): TObject;
  function IndexOf(Obj: TObject): integer;
end;
```

Count method returns the current number of objects in the list.

Item method returns the object in the position I of the list (0-based)

IndexOf method returns the position of the object Obj in the list (also 0-based)

8.3 Using Fields

In TAureliusDataset, each field represents a property in an object. So, for example, if you have a class declared like this:

```
  TCustomer = class
  // <snip>
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    property Birthday: Nullable<TDate> read FBirthday write FBirthday;
  end;
```

when providing an object of class TCustomer to the dataset, you will be able to read or write its properties this way:

```
CustomerName := AureliusDataset1.FieldByName('Name').AsString;
if AureliusDataset1.FieldByName('Birthday').IsNull then
  AureliusDataset1.FieldByName('Birthday').AsDateTime := EncodeDate(1980,
1, 1);
```

As with any TDataset descendant, TAureliusDataset will automatically create default fields, or you can optionally create TField components manually in the dataset, either at runtime or design-time. Creating persistent fields might be useful when you need to access a field that is not automatically present in the default fields, like a sub-property field or when working with inheritance.

The following topics explain fields usage in more details:

Default Fields and Base Class
Self Field
Sub-Property Fields
Entity Fields (Associations)
Dataset Fields (Many-Valued Associations)
Heterogeneous Lists (Inheritance)

8.3.1 Default Fields and Base Class

When you open the dataset, default fields are automatically created if no persistent fields are defined. TAureliusDataset will create a field for each property in the "base class", either regular fields, or fields representing associations or many-valued associations like entity fields and dataset fields. The "base class" mentioned is retrieved automatically by the dataset given the way you provided the objects:

1. If you provide objects by passing a generic list to SetSourceList method, Aurelius will consider the base class as the generic type in the list. For example, if the list type it TList<TCustomer>, then the base class will be TCustomer.

2. If you provide an object by using SetSourceObject, the base class will just be the class of object passed to that method.

3. You can alternatively manually specify the base class, by using the ObjectClass

property. Note that this must be done after calling SetSourceList or SetSourceObject, because these two methods update the ObjectClass property internally. Example:

```
AureliusDataset1.SetSourceList(SongList);
AureliusDataset1.ObjectClass := TMediaFile;
```

8.3.2 Self Field

One special field that is created by default or you can add manually in persistent fields is a field named "Self". It is an entity field representing the object associated with the current record. It's useful for lookup fields. In the following code, both lines are equivalent (if there is a current record):

```
Customer1 := AureliusDataset1.Current<TCustomer>;
Customer2 := AureliusDataset1.EntityFieldByName('Self')
.AsEntity<TCustomer>;
// Customer1 = Customer2
```

8.3.3 Sub-Property Fields

You can access properties of associated objects (sub-properties) through TAureliusDataset. Suppose you have a class like this:

```
  TCustomer = class
  // <snip>
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    property Country: TCountry read FCountry write FCountry;
  end;
```

You can access properties of Country object using dots:

```
AureliusDataset1.FieldByName('Country.Name').AsString := 'Germany';
```

As you might have noticed, sub-property fields can not only be read, but also written to. There is not a limit for level access, which means you can have fields like this:

```
CountryName := AureliusDataset1.FieldByName
('Invoice.Customer.Country.Name').AsString;
```

It's important to note that sub-property fields are **not** created by default when using default fields. In the example of TCustomer class above, only field "Country" will be created by default, but not "Country.Name" or any of its sub-properties. To use a sub-property field, you must manually add the field to the dataset before opening it. Just like any other TDataset, you do that at design-time, or at runtime:

```
with TStringField.Create(Self) do
```

```
begin
  FieldName := 'Country.Name';
  Dataset := AureliusDataset1;
end;
```

8.3.4 Entity Fields (Associations)

Entity Fields are fields that maps to an object property in a container object. In other words, entity fields represent associations in the object. Consider the following class:

```
  TCustomer = class
  // <snip>
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    property Country: TCountry read FCountry write FCountry;
  end;
```

By default, TAureliusDataset will create fields "Id" and "Name" (scalar fields) and "Country" (entity field). An entity field is just a field of type TAureliusEntityField that holds a reference to the object itself. Since Delphi DB library doesn't provide a field representing an object pointer (which makes sense), this new field type is provided by TMS Aurelius framework for you to manipulate the object reference.

The TAureliusEntityField is just a TVariantField descendant with an additional AsObject property, and an addition generic AsEntity<T> function that you can use to better manipulate the field content. To access such properties, you can just cast the field to TAureliusEntityField, or use TAureliusDataset.EntityFieldByName method.

Please note that the entity field just represents an object reference. It's useful for lookup fields and to programatically change the object reference in the property, but it's not useful (and should not be used) for visual binding, like a TDBGrid or to be edited in a TDBEdit, since its content is just a pointer to the object. To visual bind properties of associated objects, use sub-property fields.

The following code snippets are examples of how to use the entity field.

```
// following lines are equivalent and illustrates how to set an
association through the dataset
AureliusDataset1.EntityFieldByName('Country').AsObject :=
TCountry.Create;
(AureliusDataset1.FieldByName('Country') as TAureliusEntityField)
.AsObject := TCountry.Create;
```

Following code shows how to retrieve the value of an association property using the dataset field:

```
Country := AureliusDataset1.EntityFieldByName('Country')
.AsEntity<TCountry>;
```

8.3.5 Dataset Fields (Many-Valued Associations)

Dataset fields represent collections in a container object. In other words, dataset fields represent many-valued associations in the object. Consider the following class:

```
TInvoice = class
// <snip>
public
  property Id: Integer read FId write FId;
  property Items: TList<TInvoiceItem> read GetItems;
end;
```

The field "Items" is expected to be a TDatasetField, and represents all objects (records) in the Items collection. Different from entity fields, you don't access a reference to the list itself, using the dataset field.

In short, you can use the TDatasetField to build master-detail relationships. You can have, for example, a TDBGrid linked to a dataset representing a list of TInvoice objects, and a second TDBGrid linked to a dataset representing a list of TInvoiceItem objects. To link the second dataset (invoice items) to the first (invoices) you just need to set the DatasetField property of the second dataset. This will link the detail dataset to the collection of items in the first dataset. You can do it at runtime or design-time.

The following code snippet illustrates better how to link two datasets using the dataset field. It's worth to note that these dataset fields work as a regular TDatasetField. For a better understanding of how a TDatasetField works, please refer to Delphi documentation.

```
InvoiceDataset.SetSourceList(List);
InvoiceDataset.Manager := Manager1;
InvoiceDataset.Open;
ItemsDataset.DatasetField := InvoiceDataset.FieldByName('Items') as
TDatasetField;
ItemsDataset.Open;
```

Note that there is no need to set the Manager property of nested datasets, it's set automatically. Especially because when using nested datasets, you might want to have the nested dataset to be created automatically by accessing NestedDataset property:

```
InvoiceDataset.SetSourceList(List);
InvoiceDataset.Manager := Manager1;
InvoiceDataset.Open;
ItemsDataset := (InvoiceDataset.FieldByName('Items') as TDatasetField)
.NestedDataset as TAureliusDataset;
```

As with any master-detail relationship, you can add or remove records from the detail/ nested dataset, and it will add/remove items from the collection:

```
ItemsDataset.Append;
ItemsDataset.FieldByName('ProductName').AsString := 'A';
ItemsDataset.FieldByName('Price').AsCurrency := 1;
ItemsDataset.Post;

ItemsDataset.Append;
```

```
ItemsDataset.FieldByName('ProductName').AsString := 'B';
ItemsDataset.FieldByName('Price').AsCurrency := 1;
ItemsDataset.Post;
```

8.3.6 Heterogeneous Lists (Inheritance)

When providing objects to the dataset, the list provided might have objects instances of different classes. This happens for example when you perform a polymorphic query. Suppose you have a class hierarchy which base class is TAnimal, and descendant classes are TDog, TMammal, TBird, etc.. When you perform a query like this:

```
Animals := Manager.Find<TAnimal>.List;
```

You might end up with a list of objects of different classes like TDog or TBird. Suppose for example TDog class has a DogBreed property, but TBird does not. Still, you need to create a field named "DogBreed" so you can display it in a grid or edit that property in a form.

TAureliusDataset allows you to create fields mapped to properties that might not exist in the object. Thus, you can create a persistent field named "DogBreed", or you can change the base class of the dataset to TDog so that the default fields will include a field named "DogBreed".

To allow this feature to work well, when such a field value is requested and the property does not exist in the object, TAureliusDataset will not raise any error. Instead, the field value will be null. Thus, if you are listing the objects in a dbgrid, for example, a column associated with field "DogBreed" will display the property value for objects of class TDog, but will be empty for objects of class TBird, for example. Please note that this behavior only happens when reading the field value. If you try to set the field value and the property does not exist, an error will be raised when the record is posted. If you don't change the field value, it will be ignored.

Also note that the base class is used to create a new object instance when inserting new records (creating objects). The following code illustrates how to use a dataset associated with a TList<TAnimal> and still creating two different object types:

```
Animals := Manager.FindAll<TAnimal>;
DS.SetSourceList(Animals); // base class is TAnimal
DS.ObjectClass := TDog; // not base is class is TDog
DS.Open;
DS.Append;
DS.FieldByName('Name').AsString := 'Snoopy';
DS.FieldByName('DogBreed').AsString := 'Beagle';
DS.Post; // Create a new TDog instance
DS.Append;
DS.ObjectClass := TBird; // change base class to TBird
DS.FieldByName('Name').AsString := 'Tweetie';
DS.Post; // Create a new TBird instance. DogBreed field is ignored
```

8.3.7 Enumeration Fields

Fields that relate to an enumerated type are integer fields that hold the ordinal value of the enumeration. Example:

```
type TSex = (tsMale, tsFemale);

TheSex := TSex(DS.FieldByName('Sex').AsInteger);
DS.FieldByName('Sex').AsInteger := Ord(tsFemale);
```

Alternatively, you can use the sufix ".EnumName" after the property name so you can read and write the values in string format (string fields)

```
SexName := DS.FieldByName('Sex.EnumName').AsString;
DS.FieldByName('Sex.EnumName').AsString := 'tsFemale';
```

8.3.8 Fields for Projection Values

When using projections in queries, the result objects might be objects of type TCriteriaResult. Such object has the content of projections available in the Values property. TAureliusDataset treats such values as fields, so you can define a field for each projection value. Since TAureliusDataset cannot tell in advance what are the available fields, to use such scenario you must previously define the persistent fields for each aliased projection. The following code snippet illustrates how you can use projection values in TAureliusDataset.

```
  with TStringField.Create(Self) do
  begin
    FieldName := 'CountryName';
    Dataset := AureliusDataset1;
    Size := 50;
  end;
  with TIntegerField.Create(Self) do
  begin
    FieldName := 'Total';
    Dataset := AureliusDataset1;
  end;

  // Retrieve number of customers grouped by country
  AureliusDataset1.SetSourceCriteria(
    Manager.Find<TCustomer>
      .Select(TProjections.ProjectionList
        .Add(TProjections.Group('Country').As_('CountryName'))
        .Add(TProjections.Count('Id').As_('Total'))
        )
      .AddOrder(TOrder.Asc('Total'))
  );

  // Retrieve values for the first record: country name and number of
customers
  FirstCountry := AureliusDataset1.FieldByName('CountryName').AsString;
```

```
  FirstTotal := AureliusDataset1.FieldByName('Total').AsInteger;
```

8.4 Modifying Data

Modifying data with TAureliusDataset is just as easy as with any TDataset component. Call Edit, Insert, Append methods, and then call Post to confirm or Cancel to rollback changes.

It's worth note that TAureliusDataset load and save data from and to the **objects in memory**. It means when a record is posted, the underlying associated object has its properties updated according to field values. However the object is **not necessarily** persisted to the database. It depends on if the Manager property is set, or if you have set event handlers for object persistence, as illustrated in code below.

```
// Change Customer1.Name property
DS.Close;
DS.SetSourceObject(Customer1);
DS.Open;
DS.Edit;
DS.FieldByName('Name').AsString := 'John';
DS.Post;
// Customer1.Name property is updated to "John".
// Saving on database depends on setting Manager property
// or setting OnObjectUpdate event handler
```

The following topics explain some more details about modifying data with TAureliusDataset.

8.4.1 New Objects When Inserting Records

When you insert new records, TAureliusDataset will create new object instances and add them to the underlying object list provided to the dataset. The objects are created only when you post the record, and the class of object being created is specified by the base class (either retrieved from the list of objects or manually using ObjectClass property). See Default Fields and Base Class topic for more details.

In the following code, a new TCustomer object will be created when record is posted:

```
Customers := TObjectList<TCustomer>.Create;
DS.SetSourceList(Customer); // base class is TCustomer
DS.Open;
DS.Append;
DS.FieldByName('Name').AsString := 'Jack';
DS.Post; // Create a new TCustomer instance
// Destroy Customers list later!
```

Setting the base class manually is also important if you are using heterogeneous lists and want to create instances of different classes when posting records, depending on an specific situation.

Alternatively, you can set OnCreateObject event handler. This event is called when the dataset needs to create the object, and the event type declaration is below:

```
type
  TDatasetCreateObjectEvent = procedure(Dataset: TDataset; var NewObject:
TObject) of object;
  //<snip>
  property OnCreateObject: TDatasetCreateObjectEvent;
```

If the event handler sets a valid object into NewObject parameter, the dataset will not create the object. If NewObject is unchanged (remaining nil), then a new object of the class specified by the base class is created internally.
Here is an example of how to use it:

```
procedure TForm1.AureliusDataset1CreateObject(Dataset: TDataset; var
NewObject: TObject);
begin
  NewObject := TBird.Create;
end;
//<snip>
AureliusDataset1.OnCreateObject := AureliusDataset1CreateObject;
AureliusDataset1.Append;
AureliusDataset1.FieldByName('Name').AsString := 'Tweetie';
AureliusDatase1.Post; // a TBird object named "Tweetie" will be created
here
```

A final note: objects are not destroyed by TAureliusDataset, even the ones created internally by it. See Objects Lifetime Management for more information.

8.4.2 Manager Property

When posting records, object properties are updated, but are not persisted to the database, unless you manually set events for persistence, or set Manager property. If you set the Manager property to a valid TObjectManager object, then when records are posted or deleted, TAureliusDataset will use the specified manager to persist the objects to the database, either saving, updating or removing the objects.

```
Customers := TAureliusDataset.Create(Self);
CustomerList := TList<TCustomer>.Create;
Manager := TObjectManager.Create(MyConnection);
try
  Customers.SetSourceList(CustomerList);
  Customers.Open;
  Customers.Append;
  Customers.FieldbyName('Name').AsString := 'Jack';
  // On post, a new TCustomer object named "Jack" is created, but not
saved to database
  Customers.Post;

  // Now set the manager
  Customers.Manager := Manager;

  Customers.Append;
```

```
  Customers.FieldbyName('Name').AsString := 'John';
  // From now on, any save/delete operation on dataset will be reflected
on database
  // A new TCustomer object named "John" will be created, and
Manager.Save
  // will be called to persist object in database
  Customers.Post;

  // Record is deleted from dataset and object is removed from database
  Customers.Delete;
finally
  Manager.Free;
  Customers.Free;
  CustomerList.Free;
end;
```

In summary: if you want to manipulate objects only in memory, do not set Manager property. If you want dataset changes to be reflected in database, set Manager property or use events for manual persistence.

8.4.3 Objects Lifetime Management

TAureliusDataset does not manage any object it holds, either the entity objects itself, the list of objects that you pass in SetSourceList when providing objects to it, or even the objects it created automatically when inserting new records. So you must be sure to destroy all of them when needed!

Even when deleting records, the object is not destroyed (if no Manager is attached). The following code causes a memory leak:

```
Customers := TAureliusDataset.Create(Self);
CustomerList := TList<TCustomer>.Create;
try
  Customers.SetSourceList(CustomerList);
  Customers.Open;
  Customers.Append;
  Customers.FieldbyName('Name').AsString := 'Jack';

  // On post, a new TCustomer object named "Jack" is created, but not
saved to database
  Customers.Post;

  // Record is deleted from dataset, but object is NOT DESTROYED
  Customers.Delete;
finally
  Manager.Free;
  Customers.Free;
  CustomerList.Free;
end;
```

In code above, a new object is created in the Post, but when record is deleted, object is not destroyed, although it's removed from the list.

But, be aware that the TObjectManager object itself manages the objects. If you set the Manager property of the dataset, then records being saved will cause objects to be saved or updated by the manager, meaning they will be managed by it. It works just as any object manager. So usually you would not need to destroy objects if you are using a TObjectManager associated with the dataset (but you would still need to destroy the TList object holding the objects). But just know that they are being managed by the TObjectManager object, not by the TAureliusDataset component itself.

8.4.4 Manual Persistence Using Events

To properly persist objects to the database and manage them by properly destroying when needed, you would usually use the Manager property and associate a TObjectManager object to the dataset.
Alternatively, you can also use events for manual persistence and management. Maybe you just want to keep objects in memory but need to destroy them when records are deleted, so you can use OnObjectRemove event. Or maybe you just want to hook a handler for the time when an object is updated and perform additional operations.

The following events for handling objects persistence are available in TAureliusDataset, and all of them are of type TDatasetObjectEvent:

```
type
  TDatasetObjectEvent = procedure(Dataset: TDataset; AObject: TObject) of
 object;

  //<snip>
  property OnObjectInsert: TDatasetObjectEvent;
  property OnObjectUpdate: TDatasetObjectEvent;
  property OnObjectRemove: TDatasetObjectEvent;
```

OnObjectInsert event is called when a record is posted after an Insert or Append operation, right after the object instance is created.
OnObjectUpdate event is called when a record is posted after an Edit operation.
OnObjectRemove event is called when a record is deleted.

In all events, the AObject parameter related to the object associated with the current record.

Note that if one of those event handlers are set, the object manager specified in Manager property will be ignored and not used. So if for example you set an event handler for OnObjectUpdate event, be sure to persist it to the database if you want to, because Manager.Update will not be called even if Manager property is set.

8.5 Locating Records

TAureliusDataset supports usage of Locate method to locate records in the dataset. Use it just as with any regular TDataset descendant:

```
Found := AureliusDataset1.Locate('Name', 'mi', [loCaseInsensitive,
loPartialKey]);
```

You can perform locate on entity fields. Just note that since entity fields hold a reference to the object itself, you just need to pass a reference in the locate method. Since objects cannot be converted to variants, you must typecast the reference to an Integer or IntPtr (Delphi XE2 and up).

```
{$IFDEF DELPHIXE2}
Invoices.Locate('Customer', IntPtr(Customer), []);
{$ELSE}
Invoices.Locate('Customer', Integer(Customer), []);
{$ENDIF}
```

The customer object must be the same. Even if Customer object has the same Id as the object in the dataset, if the object references are not the same, Locate will fail. Alternatively, you can also search on sub-property fields:

```
Found := Invoices.Locate('Customer.Name', Customer.Name, []);
```

In this case, the record will be located if the customer name matches the specified value, regardless if object references are the same or not.
You can also search on calculated and lookup fields.

8.6 Calculated Fields

You can use calculated fields in TAureliusDataset the same way with any other dataset. Note that when calculating fields, you can use regular Dataset.FieldByName approach, or you can use Current<T> property and access the object properties directly.

```
procedure TForm1.AureliusDataset1CalcFields(Dataset: TDataset);
begin
  if AureliusDataset1.FieldByName('Birthday').IsNull then
    AureliusDataset1.FieldByName('BirthdayText').AsString := 'not
specified'
  else
    AureliusDataset1.FieldByName('BirthdayText').AsString := DateToStr
(AureliusDataset1.FieldByName('Birthday').AsDateTime);

  case AureliusDataset1.Current<TCustomer>.Sex of
    tsMale:
      AureliusDataset1.FieldByName('SexDescription').AsString := 'male';
    tsFemale:
      AureliusDataset1.FieldByName('SexDescription').AsString :=
'female';
  end;
end;
```

8.7 Lookup Fields

You can use lookup fields with TAureliusDataset, either at design-time or runtime. Usage is not different from any TDataset.

One thing it's worth note, though, is how to use lookup field for entity fields (associations), which is probably the most common usage. Suppose you have a TInvoice class with a property Customer that is an association to a TCustomer class. You can have two datasets with TInvoice and TCustomer data, and you want to create a lookup field in Invoices dataset to lookup for a value in Customers dataset, based on the value of Customer property.

Since "Customer" is an entity field in Invoices dataset, you need to lookup for its value in the Customers dataset using the "Self" field, which represents a reference to the TCustomer object in Customers dataset. The following code illustrates how to create a lookup field in Invoices dataset to lookup for the customer name based on "Customer" field:

```
// Invoices is a dataset which data is a list of TInvoice objects
// Customers is dataset which data is a list of TCustomer objects

// Create the lookup field in Invoices dataset
LookupField := TStringField.Create(Invoices.Owner);
LookupField.FieldName := 'CustomerName';
LookupField.FieldKind := fkLookup;
LookupField.Dataset := Invoices;
LookupField.LookupDataset := Customers;
LookupField.LookupKeyFields := 'Self';
LookupField.LookupResultField := 'Name';
LookupField.KeyFields := 'Customer';
```

Being a regular lookup field, this approach also works with componentes like TDBLookupComboBox and TDBGrid. It would display a combo with a list of customer names, and will allow you to change the customer of TInvoice object by choosing the item in combo (the field "Customer" in Invoices dataset will be updated with the value of field "Self" in Customers dataset).

8.8 Filtering

TAureliusDataset supports filtering of records by using regular TDataset.Filtered property and TDataset.OnFilterRecord event. It works just as any TDataset descendant. Note that when filtering records, you can use regular Dataset.FieldByName approach, or you can use Current<T> property and access the object properties directly.

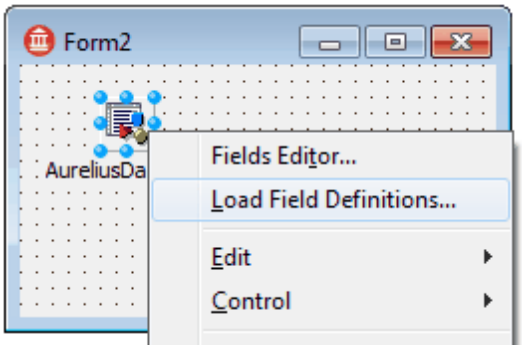
```
procedure TForm1.DatasetFilterRecord(Dataset: TDataset; var Accept:
boolean);
begin
  Accept :=
    (Dataset.FieldByName('Name').AsString = 'Toby')
    or
    (TAureliusDataset(Dataset).Current<TAnimal> is TMammal);
end;
//<snip>
begin
  AureliusDataset1.SetSourceList(Animals);
```

```
  AureliusDataset1.Open;
  AureliusDataset1.OnFilterRecord := DatasetFilterRecord;
  AureliusDataset1.Filtered := True;
end;
```

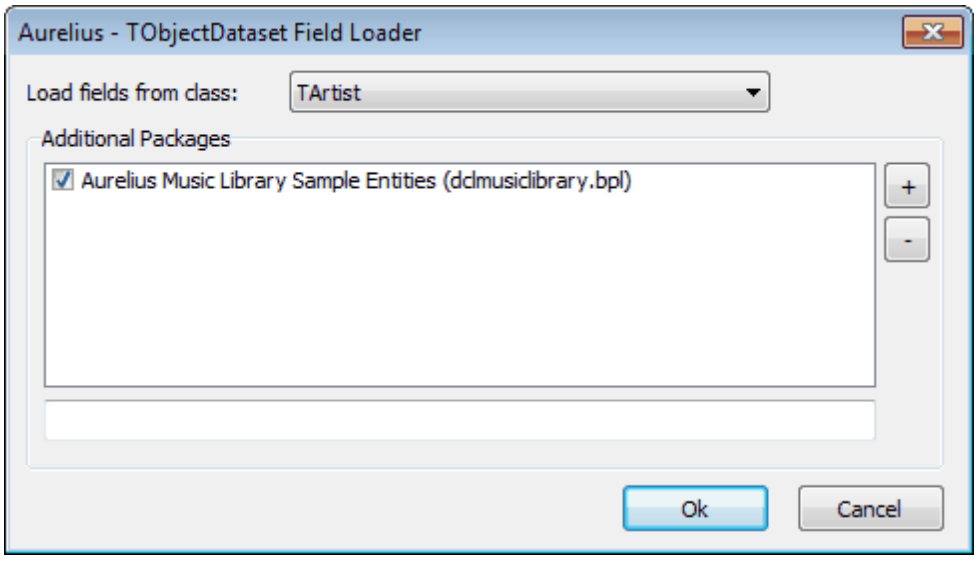
8.9 Design-time Support

TAureliusDataset is installed in Delphi component palette and can be used at design-time and as any TDataset component you can set its fields using fields editor, specify master-detail relationships by setting DatasetField property to a dataset field, create lookup fields, among other common TDataset tasks.

However, creating fields manually might be a boring task, especially if you have a class with many properties and need to create many fields manually. So TAureliusDataset provides a design-time menu option named "Load Field Definitions..." (right-click on the component), which allows you to load a class from a package and create the field definitions from that class.



A dialog appears allowing you to choose a class to import the definitions from. Note that the classes are retrieving from available packages. By default, classes from packages installed in the IDE are retrieved. If you want to use a package that is not installed, you can add it to the packages list. So, for a better design-time experience with TAureliusDataset, **create a package with all your entity classes, compile it**, and load it in this dialog.



The packages in the list are saved in the registry so you can reuse it whenever you need. To remove the classes of a specified package from the combo box, just uncheck the package. The package will not keep loaded: when the dialog closes, the package is unloaded from memory.

Note that the dialog will create fill the **FieldDefs property**, not create field components in the fields editor. The FieldDefs behaves as if the field definitions are being retrieved from a database. You would still need to create the field components, but now you can use the FieldDefs to help you, so you can use "Add All Fields" or "Add Field..." options from the fields editor popup menu. The FieldDefs property is persisted in the form so you don't need to reload the package in case you close the form and open it again. That's its only purpose, and they are not used at runtime.

Chapter

Distributed Applications

9 Distributed Applications

You can build distributed applications using Aurelius. When mapping classes, you can specify any class ancestor, and you can define which fields and properties will be mapped or not. This gives you flexibility to use almost any framework for building distributed applications - even if that framework requires that the classes need to have specific behavior (like inheriting from a specific base class, for example).

Still, Aurelius provides several mechanisms and classes that make building distributed applications even easier. The following topics describe features for building distributed applications using Aurelius.

JSON - JavaScript Object Notation

9.1 JSON - JavaScript Object Notation

When building distributed applications, you need to transfer your objects between peers. Usually to transfer objects you need to convert them (marshal) to a format that you can send through your communication channel. Currently one of the most popular formats for that is the JSON format. It's simple, text representation, that can easily be parsed, lightweight, and portable. You can build your server using Aurelius, retrieve your objects from database, convert them to JSON, send the objects through any communication channel to client, and from the client, you can convert the JSON back to an Aurelius object. Since it's a portable format, your client doesn't even need to be a Delphi application using Aurelius - you can use a JavaScript client, for example, that fully supports the JSON format, or any other language.

To converting Aurelius objects to JSON you can use one of the available JSON serializers:

```
Serializer := TDataSnapJsonSerializer.Create;
try
  JsonValue := Serializer.ToJson(Customer);
finally
  Serializer.Free;
end;
```

To convert a JSON notation back to an Aurelius object, you can use one of the available JSON deserializers:

```
Deserializer := TDataSnapJsonDeserializer.Create;
try
  Customer := Deserializer.FromJson<TCustomer>(JsonValue);
finally
  Deserializer.Free;
end;
```

The following topics describes in more details how to better use the JSON with Aurelius.

Available Serializers
Serialization behavior

Lazy-Loading with JSON
Memory Management with JSON

9.1.1 Available Serializers

Aurelius uses an open architecture in JSON support that allows you to use any framework for parsing and generating the JSON representation. This makes it easy to use your preferred framework for building distributed applications and use legacy code. For example, if you are using DataSnap, Aurelius provides the DataSnap serializer that converts the object to a TJsonValue object which holds the JSON representation structure. You can use the TJsonValue directly in a DataSnap server to send JSON to the client. Other frameworks use different objects for JSON representation (or simply string format) so you can use any you want.

The following table lists the currently available JSON serializer/deserializer classes in Aurelius, what framework they use, and what is the base type that is uses for JSON representation:

Framework	Serializer class	Deserializer class	JSON Class	Declared in unit	Vendor Site
DataSnap	TDataSnapJsonSerializer	TDataSnapJsonDeserializer	TJsonValue	Aurelius.Json.DataSnap	Delphi Native
SuperObject	TSuperObjectJsonSerializer	TSuperObjectJsonDeserializer	ISuperObject	Aurelius.Json.SuperObject	http://code.google.com/p/superobject/

All serializers have a ToJson method that receives an object and returns the type specified by the JSON Class in the table above.
All deserializers have a generic FromJson method that receives the type specified by JSON class in the table above and returns the type specified in the generic parameter.

Both serializer and deserializer need a reference to a TMappingExplorer object to work with. You can pass the object in the Create constructor when creating a serializer/deserializer, or you can use the method with no parameter to use the default mapping setup. The following code snippets illustrate different ways of using the serializers:

Serializing/Deserializing an Aurelius object using DataSnap JSON classes and default mapping setup:

```
uses
  {...}, Aurelius.Json.DataSnap;
var
  Serializer: TDataSnapJsonSerializer;
  Deserializer: TDataSnapJsonDeserializer;
  Customer: TCustomer;
  AnotherCustomer: TCustomer;
  JsonValue: TJsonValue;
begin
  {...}
  Serializer := TDataSnapJsonSerializer.Create;
  Deserializer := TDataSnapJsonDeserializer.Create;
  try
```

```
    JsonValue := Serializer.ToJson(Customer);
    AnotherCustomer := Deserializer.FromJson<TCustomer>(JsonValue);
  finally
    Serializer.Free;
    Deserializer.Free;
  end;
  {...}
end;
```

Serializing/Deserializing an Aurelius object using SuperObject and custom mapping setup:

```
uses
  {...}, Aurelius.Json.SuperObject;
var
  Serializer: TSuperObjectJsonSerializer;
  Deserializer: TSuperObjectJsonDeserializer;
  Customer: TCustomer;
  AnotherCustomer: TCustomer;
  SObj: ISuperObject;
  CustomMappingExplorer: TMappingExplorer;
begin
  {...}
  Serializer := TSuperObjectJsonSerializer.Create(CustomMappingExplorer);
  Deserializer := TSuperObjectJsonDeserializer.Create
(CustomMappingExplorer);
  try
    SObj := Serializer.ToJson(Customer);
    AnotherCustomer := Deserializer.FromJson<TCustomer>(SObj);
  finally
    Serializer.Free;
    Deserializer.Free;
  end;
  {...}
end;
```

9.1.2 Serialization behavior

Aurelius maps each relevant field/attribute to the JSON representation, so that the JSON holds all (and only) relevant information to represent an object state. So for example, a class mapped like this:

```
  [Entity]
  [Table('ARTISTS')]
  [Id('FId', TIdGenerator.IdentityOrSequence)]
  TArtist = class
  private
    [Column('ID', [TColumnProp.Unique, TColumnProp.Required,
TColumnProp.NoUpdate])]
    FId: Integer;
    FArtistName: string;
    FGenre: Nullable<string>;
    function GetArtistName: string;
```

```
    procedure SetArtistName(const Value: string);
  public
    property Id: integer read FId;
    [Column('ARTIST_NAME', [TColumnProp.Required], 100)]
    property ArtistName: string read GetArtistName write SetArtistName;
    [Column('GENRE', [], 100)]
    property Genre: Nullable<string> read FGenre write FGenre;
  end;
```

will generate the following JSON representation:

```
{
    "$type": "Artist.TArtist",
    "$id": 1,
    "FId": 2,
    "ArtistName": "Smashing Pumpkins",
    "Genre": "Alternative"
}
```

Note that fields FId and properties ArtistName and Genre are mapped, and so are the ones that appear in the JSON format. Aurelius includes extra meta fields (starting with $) for its internal use that will make it easy to later deserialize the object. Nullable types and dynamic properties are automatically handled by the serializer/deserializer.

Blob fields

Content of blobs are converted into a base64 string so it can be properly deserialized back to a binary format (Data field is truncated in example below):

```
{
    "$type": "Images.TImage",
    "$id": 1,
    "FId": 5,
    "ImageName": "Landscape",
    "Data":
"TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0
aGlz...
}
```

If blobs are set to be lazy and they are not loaded, then they will not be fully sent in JSON representation, but only a meta information that will allow you to load it later. See more at Lazy-Loading with JSON

Associations

If the object being serialized has associations and/or many-valued associations, those objects are also serialized in the JSON. The following example shows a serialization of a class TSong which has properties Album, Artist and SongFormat that points to other objects:

```
{
    "$type": "Song.TSong",
    "$id": 1,
```

```
    "FAlbum": {
        "$proxy": "single",
        "key": 2,
        "class": "TMediaFile",
        "member": "FAlbum"
    },
    "MediaName": "Taxman2",
    "Duration": 230,
    "FId": 1,
    "FArtist": {
        "$proxy": "single",
        "key": 1,
        "class": "TMediaFile",
        "member": "FArtist"
    },
    "FileLocation": "",
    "SongFormat": {
        "$type": "SongFormat.TSongFormat",
        "$id": 2,
        "FId": 1,
        "FormatName": "MP3"
    }
}
```

If the association is marked as lazy-loading and is not load yet, then they will not be included in JSON representation, but instead a meta information will be included for later loading the value. In the example above, FAlbum and FArtist were defined as proxies and were not loaded, so the object they hold is a proxy meta information. On the other hand, SongFormat property is loaded and the whole TSongFormat object is serialized in it. For more information on lazy-loading, see Lazy-Loading with JSON.

9.1.3 Lazy-Loading with JSON

An object being serialized might have associations and many-valued associations defined to be lazy-loaded. When that is the case and the proxies are not loaded yet, the associated objects are not serialized, but instead, an object with metadata for that proxy is serialized instead. Take a look at the following example (irrelevant parts of the real JSON notation were removed):

```
{
    "$type": "Song.TSong",
    "$id": 1,
    "FId": 1,
    "FAlbum": {
        "$proxy": "single",
        "key": 2,
        "class": "TMediaFile",
        "member": "FAlbum"
    },
    "FileLocation": ""
}
```

In that example, TSong has a FAlbum field of type Proxy<TAlbum>. The song being serialized doesn't have the FAlbum field loaded, so instead of the actual TAlbum object to be serialized, a proxy object is serialized instead. The proxy object is indicated by the presence of the meta property "$proxy", which indicates if it's a proxy for a single object or a list.

How does the deserializer handle this? All JSON deserializers have a property ProxyLoader which points to an interface of type IJsonProxyLoader declared like this:

```
  IJsonProxyLoader = interface
    function LoadProxyValue(ProxyInfo: IProxyInfo): TObject;
  end;
```

while the IProxyInfo object is declared like this (in unit Aurelius.Types.Proxy):

```
  IProxyInfo = interface
    function ProxyType: TProxyType;
    function ClassName: string;
    function MemberName: string;
    function Key: Variant;
  end;
```

When the TSong object in the previous example is deserialized, an internal proxy is set automatically in the FAlbum field. When the Album property of Song object is read, the proxy calls the method LoadProxyValue of the IJsonProxyLoader interface. So for the object to be loaded by the proxy, you must provide a valid IJsonProxyLoader interface in the deserializer so that you can load the proxy and pass it back to the engine. The easiest way to create an IJsonProxyLoader interface is using the TJsonProxyLoader interface object provided by Aurelius.

The following code illustrates how to do it:

```
Deserializer := TDataSnapJsonDeserializer.Create;
try
  Deserializer.ProxyLoader := TJsonProxyLoader.Create(
    function(ProxyInfo: IProxyInfo): TObject
    var
      Serializer: TDataSnapJsonSerializer;
      Deserializer: TDataSnapJsonDeserializer;
      JsonObject: TJsonValue;
    begin
      Serializer:= TDataSnapJsonSerializer.Create;
      Deserializer := TDataSnapJsonDeserializer.Create;
      try
        JsonObject := DatasnapClient.RemoteProxyLoad(Serializer.ToJson
(ProxyInfo));
        Result := Deserializer.FromJson(JsonObject, TObject);
      finally
        Deserializer.Free;
        Serializer.Free;
      end;
    end
    );
```

```
  Song := Deserializer.FromJson<TSong>(JsonValueWithSong);
finally
  Deserializer.Free;
end;
// At this point, Song.Album is not loaded yet
// When the following line of code is executed (Album property is read)
// then the method specified in the ProxyLoader will be executed and
// Album will be loaded
Album := Song.Album;
AlbumName := Album.Name;
```

you can safely destroy the deserializer after the object is loaded, since the reference to the proxy loader will be in the object itself. It's up to you how to implement the ProxyLoader. In the example above, we are assuming we have a client object with a RemoteProxyLoad method that calls a server method passing the ProxyInfo data as json format. In the server, you can easily implement such method just by receiving the proxy info format, converting it back to IProxyInfo interface and then calling TObjectManager.ProxyLoad method:

```
// This methos assumes that Serializer, Deserializer and ObjectManager
objects
// are already created by the server
function TMyServerMethods.RemoteProxyLoad(JsonProxyInfo: TJsonValue):
TJsonValue;
var
  ProxyInfo: IProxyInfo;
begin
  ProxyInfo := Deserializer.ProxyInfoFromJson<IProxyInfo>(JsonProxyInfo);
  Result := Serializer.ToJson(ObjectManager.ProxyLoad(ProxyInfo));
end;
```

Lazy-Loading Blobs

In an analog way, you can lazy-load blobs with Json. It works exactly the same as loading associations. The deserializer has a property named Blob loader which points to an IJsonBlobLoader interface:

```
  IJsonBlobLoader = interface
    function ReadBlob(BlobInfo: IBlobInfo): TArray<byte>;
  end;
```

and the IBlobInfo object is declared like this (in unit Aurelius.Types.Blob):

```
  IBlobInfo = interface
    function ClassName: string;
    function MemberName: string;
    function Key: Variant;
  end;
```

and you can use TObjectManager.BlobLoad method at server side.

9.1.4 Memory Management with JSON

When deserializing a JSON value, objects are created by the deserializer. You must be aware that not only the main object is created, but also the associated objects, if it has associations. For example, if you deserialize an object of class TSong, which has a property TSong.Album, the object TAlbum will be also serialized. Since you are not using an object manager that manages memory for you, in theory you would have to destroy those objects:

```
Song := Deserializer.FromJson<TSong>(JsonValue);
{ do something with Song, then destroy it - including associations }
Song.Album.Free;
Song.Free;
```

You might imagine that if your JSON has a complex object tree, you will end up having to destroy several objects (what about Song.Album.AlbumType.Free, for example). To minimze this problem, deserializers have a property OwnsEntities that when enabled, destroys every object created by it (except lists). So your code can be built this way:

```
Deserializer := TDataSnapJsonDeserializer.Create;
Deserializer.OwnsEntities := true;
Song := Deserializer.FromJson<TSong>(JsonValue);
{ do something with Song, then destroy it - including associations }
Deserializer.Free;
// After the above line, Song and any other associated object
// created by the deserializer are destroyed
```

Alternatively, if you still want to manage objects by yourself, but want to know which objects were created by the deserializer, you can use OnEntityCreated event:

```
Deserializer.OnEntityCreated := EntityCreated;

procedure TMyClass.EntityCreated(Sender: TObject; AObject: TObject);
begin
  // Add created object to a list for later destruction
  FMyObjects.Add(AObject);
end;
```

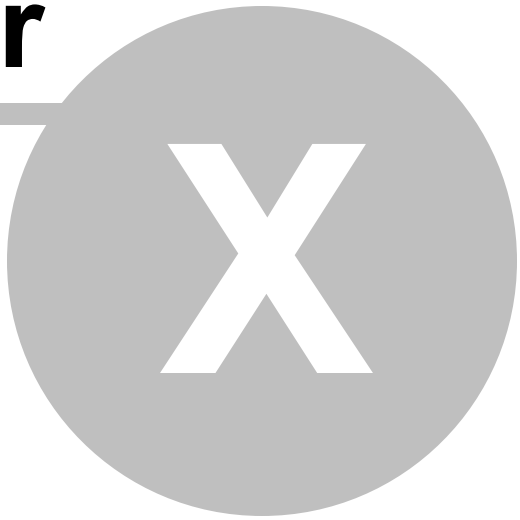
Note about JSON classes created by serializer

You must also be careful when converting objects to JSON. It's up to you to destroy the class created by the serializer, if needed. For example:

```
var
  JsonValue: TJsonValue;
begin
  Value := DataSnapDeserializer.ToJson(Customer);
  // Value must be destroyed later
```

In the previous example, Value is a TJsonValue object and it must be destroyed. Usually you will use DataSnap deserializer in a DataSnap application and in most cases where you use TJsonValue objects in DataSnap, the framework will destroy the object automatically. Nevertheless you must pay attention to situations where you need to destroy it.

Chapter



Advanced Topics

10 Advanced Topics

Here we present some advanced topics about TMS Aurelius.

10.1 Global Configuration

TMS Aurelius has a single global class that has some properties for setting global configuration. This class is declared in unit Aurelius.Globals.Config, and to access the global configuration object, use TGlobalConfigs.GetInstance:

```
uses
  Aurelius.Global.Config;
...
Configs := TGlobalConfigs.GetInstance;
```

The following properties are available in the TGlobalConfigs object:

property SimuleStatements: Boolean;
If true, all statements are not executed on the DBMS, but appear in the listeners.

property MaxEagerFetchDepth: Integer;
Indicates the maximum depth to load objects in eager loading associations. Beyond this depth, the objects still load in lazy mode.

property AutoSearchMappedClasses: Boolean;
If true, all classes declared in your application with [Entity] attribute are automatically added to the framework's MappedClasses.
Removed in version 2.0: Use TMappingSetup.MappedClasses property instead.

property TightStringEnumLength: Boolean;
If true, in enumerations mapped to string columns with no length specified in the Column attribute will generate the column length equal to the largest possible value of the enumeration. Otherwise, the length is DefaultStringColWidth by default (when not specified in Column attribute).

property AutoMappingMode: TAutomappingMode;
Defines the automapping mode. Valid values are:
Off: No automatic mapping. Only elements with attributes are mapped.
ByClass: Automapping is done for classes marked with Automapping attribute.
Full: Full automapping over every registered class and Enumerations.

property AutoMappingDefaultCascade: TCascadeTypes;
property AutoMappingDefaultCascadeManyValued: TCascadeTypes;
If AutoMapping is enabled, defines the default cascade type for all automapped associations (AutoMappingDefaultCascade) and many-valued associations (AutoMappingDefaultCascadeManyValued).
Default values are:

```
AutoMappingDefaultCascade := CascadeTypeAll - [TCascadeType.Remove];
```

```
AutoMappingDefaultCascadeManyValued := CascadeTypeAll;
```

property DefaultStringColWidth: Integer;
Defines the width for string (usually varchar) columns when the width was not particularly specified in Column attribute.