Overview

TMS Aurelius is an Object-Relational Mapping (ORM) framework. Its purpose is to be the definitive ORM framewok for the Delphi 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: https://www.tmssoftware.com/site/aurelius.asp

TMS Software site: https://www.tmssoftware.com

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.

Features

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

- Several database servers supported (MS SQL Server, Firebird, MySQL, PostgreSQL, Oracle, etc.);
- Several database-access components supported (FireDac, UniDac, dbExpress, ADO, AnyDac, SQLDirect, etc.);
- Native database drivers allow direct database access without needing a 3rd party component;
- Import existing database model and generate mapped Aurelius entity classes from it;
- Multi-platform solution - Win32, Win64, Mac OS X, Linux, 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.

In this section:

Getting Started

Basic info about how to get started using TMS Aurelius.

Database Connectivity

How you properly configure Aurelius to access the database where objects will be saved to.

Mapping

Everything about the class-to-database mapping mechanism from TMS Aurelius.

Multi-Model Design

Defining multiple mapping models in TMS Aurelius.

Manipulating Objects

Querying, saving, updating, deleting and other topics about dealing with objects.

Queries

Performing queries at object level with TMS Aurelius.

Dictionary

Dictionary allow building Aurelius queries in an even easier and more productive way.

Data Validation

Add declarative validations to your mapping to make sure your entity is persisted in a valid state.

Filters

Applying global filters to all entities at once, making it easy, for example, to build multitenant applications.

Data Binding - TAureliusDataset

Using TAureliusDataset component to bind entity objects to data-aware controls.

Distributed Applications

Features for building distributed applications using Aurelius.

Events

How to use the event system to receive callback notifications.

Advanced Topics

Some advanced topics about TMS Aurelius.

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.

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.

NOTE
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 (FireDAC, ADO) and obtain an *IDBConnection* interface from it.

NOTE
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.

Or, use a native database driver to connect to the database directly.

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

var
  MyConnection: IDBConnection;
begin
  // FDConnection1 is a FireDac TFDConnection component
  // You can use several different data-access component libraries
  // or use a native database driver to connect directly
  MyConnection := TFireDacConnectionAdapter.Create(FDConnection1, 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.

NOTE
TDatabaseManager is declared in unit `Aurelius.Engine.DatabaseManager`.

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

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

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"?

NOTE

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(
     Linq['Email'].Like('%beatles.org%')
     or Linq['Email'].Like('%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. ou 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 use TMS XData automatic CRUD endpoints to create REST server and distributed applications that automatically send and receive Aurelius objects via JSON.

Database Connectivity

This chapter explains how you properly configure Aurelius to access the database where objects will be saved to.

To connect to a database using Aurelius, you can use:

- Adapter Mode: In this mode you will use an existing 3rd party database-access component, like FireDAC, dbExpress, ADO, etc.
- Native Driver Mode: In this mode TMS Aurelius will connect to the database directly.

The database connection is represented by the IDBConnection Interface.

TAureliusConnection component is the easiest and most straightforward way to configure a connection and retrieve the IDBConnection interface. It supports both adapter and driver mode and has design-time wizards to help you out. With the TAureliusConnection component you can also generate entities from existing database.

Alternatively, you can always create the IDBConnection interface directly from code using the component adapters or native database drivers.

You can also have the option to use the Connection Wizard to automatically create the TAureliusConnection component in a new TDataModule, including the adapted connection component if you're going to use one (FireDac, for example).

See the following topics for detailed information about database connectivity in TMS Aurelius.

Using the Connection Wizard

To connect to a database, you need an IDBConnection interface representing the database connection. The easiest way to get one is using the "TMS Aurelius Connection" wizard which is available in Delphi IDE after you installed Aurelius.

To create a new connection:

1. Choose *File* > *New* > *Other* and then look for "*TMS Business*" category under "Delphi Projects". Then double click "*TMS Aurelius Connection*".
2. Choose between *Adapter Mode* or *Driver Mode*.
   For Adapter Mode, select the *Adapter* (component to access database) and the *SQL Dialect* (type of database server).
   For Driver Mode, select *Driver* to use.
3. A new data module will be created with a TAureliusConnection component already preconfigured. If you used the adapter mode, the adapted component will also be created. Either configure the connection settings in the adapted connection (adapter mode) or directly in TAureliusConnection (for driver mode).
4. To retrieve a new IDBConnection interface from the data module, just use this code:

```
// The name of data module class might vary from TFireDacMSSQLConnection
// depending on selected driver and SQL Dialect
NewConnection := TFireDacMSSQLConnection.CreateConnection;
```

Remarks

The wizard shows the following options:

For Adapter mode

- **Adapter**: Choose the database component you want to use to connect to the database. You can choose any that is supported by Aurelius component adapters, like FireDac, dbExpress, dbGo (ADO), among others.
- **SQL Dialect**: Choose the SQL dialect to be used when executing SQL statements to the database. Some drivers support several dialects (like FireDac for example), and some support just one (for example, SQLite driver only supports SQLite dialect).

For Driver mode

- **Driver**: Choose the native database driver you want to use to connect to database, for example "SQLite" or "MSSQL".

You can freely configure and try the connection at design-time the usual way you do with your component, that's the purpose of it - to be RAD and working at design-time. It's always a good practice to close the connection once you have tested and configured it, though.

The name of the data module is automatically defined by the wizard and it's a combination of the driver and sql dialect you selected. In the example above, it was FireDac driver and MSSQL dialect, but could be different. You can always change this name later.

It's important to note that **no instance** of the data module will be auto-created. Also, the *CreateConnection* method always create a new instance of the data module, so if you intend to use a single global connection for the application (which is usual for client/server applications), call CreateConnection just once and save the created IDBConnection interface for further use.

IDBConnection Interface

The *IDBConnection* interface is the lowest-level representation of 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 a component adapter or a native driver - the two ways available to connect to a database - making it transparent for the framework. Thus, regardless if you connect to the database using FireDac, dbExpress, ADO, IBX, etc., or directly using native drivers, in the end all you need IDBConnection.

To obtain an IDBConnection interface you instantiate a class of an adapter or a driver. The adapters just take an existing data access component (TFDConnection, TSQLConnection, TADOConnection, etc.) and give you back the IDBConnection interface you need to use. The native driver takes connection parameters to know how to connect to the database. To create database connections it's important to know the available adapters and drivers:

- Native Database Drivers
- Component Adapters
- SQL Dialects

In summary:

To obtain an IDBConnection interface using a native driver

Instantiate the connection class for the database you want to connect and pass the parameters in the *Create* method. For example, to connect to SQL Server:

```
uses Aurelius.Drivers.MSSQL;
{...}
var
  MyConnection: IDBConnection;
begin
  MyConnection := TMSSQLConnection.Create(
    'Server=.\SQLEXPRESS;Database=Northwnd;TrustedConnection=True');
  // Use your connection now
  Manager := TObjectManager.Create(MyConnection);
  {...}
end;
```

For more information about the available drivers, the class names and valid parameters, see Native Database Drivers.

To obtain an IDBConnection interface using an adapter

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

If you use FireDAC, for example, just drop a *TFDConnection* 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 *FDConnection1*.

```
FDConnection1: TFDConnection;
```

2. Instantiate an adapter passing the connection component.

```
uses Aurelius.Drivers.FireDac;
{...}
var
  MyConnection: IDBConnection;
begin
  MyConnection := TFireDacConnectionAdapter.Create(FDConnection1, False);
  // Use your connection now
  Manager := TObjectManager.Create(MyConnection);
  {...}
end;
```

For more information about how to create adapters, see Component Adapters.

To obtain an IDBConnection interface from a TAureliusConnection component

Once you have configured your *TAureliusConnection* component (which also provide an adapter mode or native driver mode), just create a new *IDBConnection* interface by using the *CreateConnection* method:

```
var
  MyConnection: IDBConnection;
begin
  MyConnection := AureliusConnection1.CreateConnection;
end;
```

TAureliusConnection Component

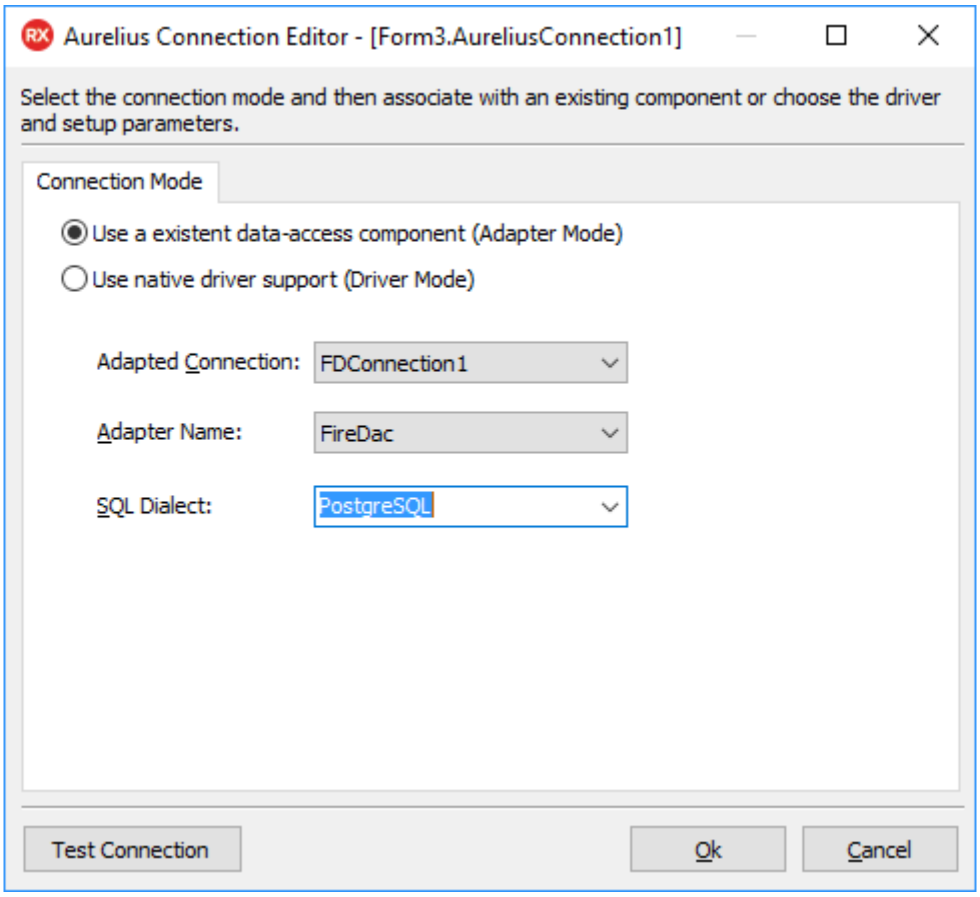
TAureliusConnection component is a RAD and easy way to configure the connection to your database, at both design-time and runtime. In the end, the main purpose of this component is also to provide the *IDBConnection* interface that is used by the whole framework, using the *CreateConnection* method:

```
var MyConnection: IDBConnection;
...
  MyConnection := AureliusConnection1.CreateConnection;
```

Configuring the connection using Connection Editor

Easiest way to configure TAureliusConnection component is double clicking the component at design-time, to open the connection editor. You can then choose if it will connect to the database with an existing database connection - through a component adapter - or directly using native database driver.

To use an adapter, click "Use an existent data-access component (Adapter Mode)":


Aurelius Connection Editor - [Form3.AureliusConnection1]

Select the connection mode and then associate with an existing component or choose the driver and setup parameters.

Connection Mode

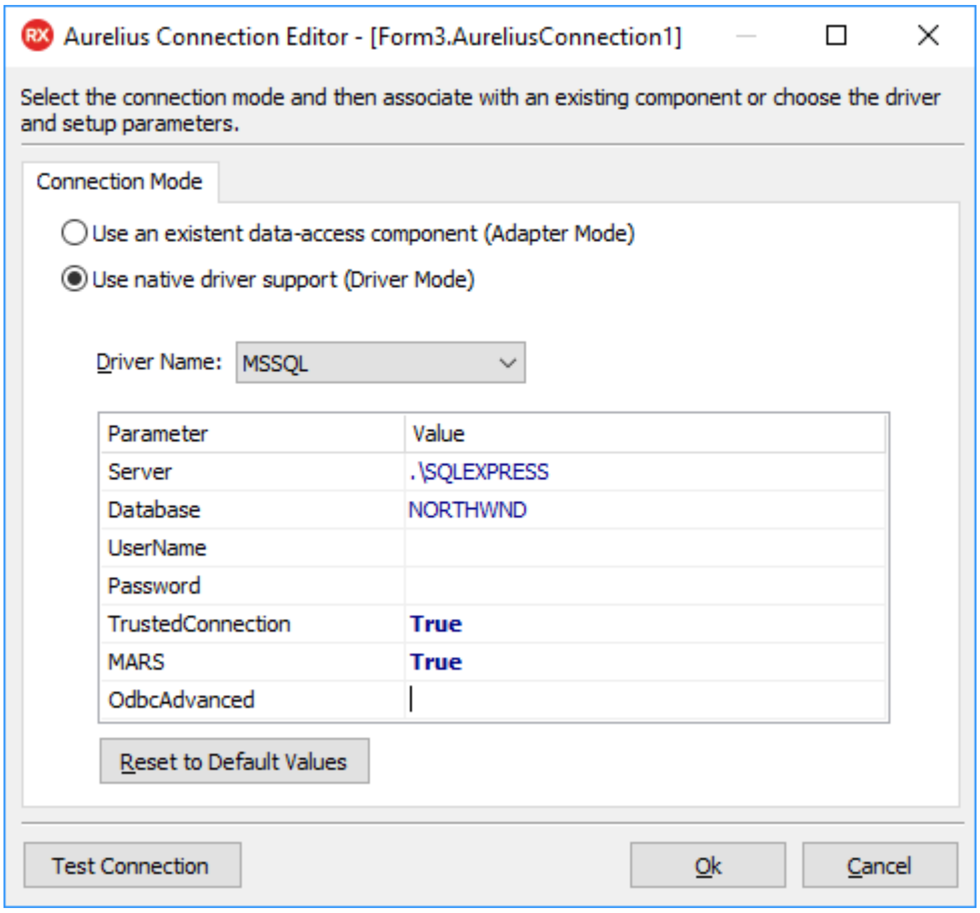
(●) Use a existent data-access component (Adapter Mode)
( ) Use native driver support (Driver Mode)

Adapted Connection: FDConnection1
Adapter Name: FireDac
SQL Dialect: PostgreSQL

For that mode, choose an existing data-access component in the "Adapted Connection" combo. The dialog will list all the supported components. The component to be adapted must be placed in the same form or data module as TAureliusConnection. Components in other forms or data modules won't be displayed.

One adapted connection is chosen, "Adapter Name" and "SQL Dialect" will often be selected automatically. If they don't, just explicitly set the adapter name and the SQL dialect to be used.

To use a native database driver, click "Use native driver support (Driver Mode)":


Aurelius Connection Editor - [Form3.AureliusConnection1]

Select the connection mode and then associate with an existing component or choose the driver and setup parameters.

Connection Mode

Use an existent data-access component (Adapter Mode) [✓ Use native driver support (Driver Mode)]

Driver Name: MSSQL

Parameter	Value
Server	.\SQLEXPRESS
Database	NORTHWND
UserName	
Password	
TrustedConnection	**True**
MARS	**True**
OdbcAdvanced	

Then choose the native "Driver Name". Once it's selected, the valid parameters for the driver will be displayed. Fill in the parameters accordingly. Refer to "Native Database Drivers" topic for the full list of the drivers with their respective driver names and parameters.

You can always use the "Test Connection" button to check if your settings are valid.

Configuring the connection using properties

You can configure the connection directly by setting properties, either at runtime from code, or at design-time using the object inspector.

To connect using a component adapter (adapter mode), set properties *AdaptedConnection*, *AdapterName* and *SQLDialect*. For example:

```
AureliusConnection1.AdaptedConnection := FDConnection1;
AureliusConnection1.AdapterName := 'FireDac';
AureliusConnection1.SQLDialect := 'PostgreSQL';
```

To connection using a native database driver (driver mode), set properties *DriverName* and use *Params* to set the parameters:

```
AureliusConnection1.DriverName := 'MSSQL';
AureliusConnection1.Params.Values['Server'] := '.\SQLEXPRESS';
AureliusConnection1.Params.Values['Database'] := 'NORTHWND';
AureliusConnection1.Params.Values['TrustedConnection'] := 'True';
```

Using the connection

To use TAureliusConnection, use *CreateConnection* method to create a new IDBConnection interface and use it:

```
var
  MyConnection: IDBConnection;
  Manager: TObjectManager;
begin
  MyConnection := AureliusConnection1.CreateConnection;
  Manager := TObjectManager.Create(MyConnection);
end;
```

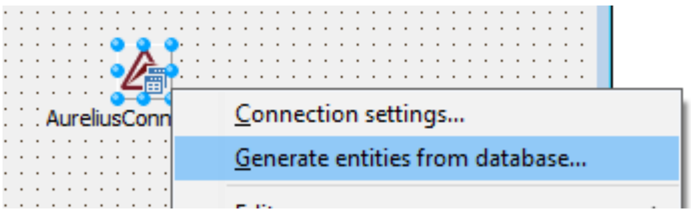
Each call to CreateConnection will create a **new** *IDBConnection* interface. If you are using a component adapter, it will also clone the existing adapted connection. To achieve that, TAureliusConnection will clone the **owner** of the adapted connection. For example, if you are adapting the FireDac TFDConnection component, and that component is placed in a data module named TMyDataModule, each type CreateConnection is called it will create a new instance of TMyDataModule, and then adapt the TFDConnection component in it. When the IDBConnection interface is not referenced anymore and is destroyed, the instance of TMyDataModule will also be destroyed.

Generate Entities From Existing Database

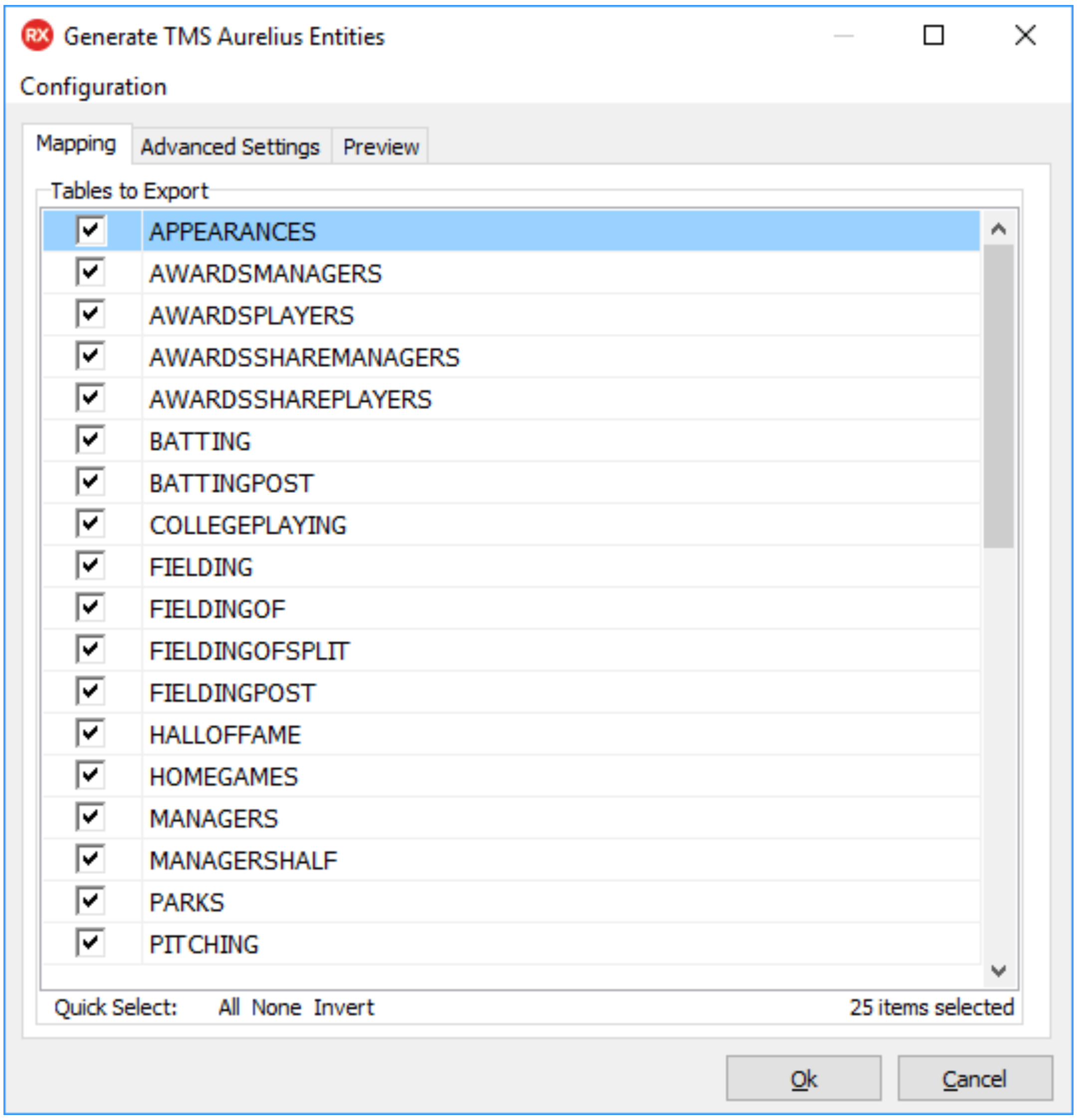
TMS Aurelius is an ORM framework which means you need to declare entity classes and map them to the database. If you have an existing database, you have the option to generate those classes automatically from the existing database.

First way this can be achieved is using the great TMS Data Modeler tool. It's a database modeling tool which can import existing database structure to the model, and then generate Delphi source code with TMS Aurelius classes. It's very powerful, with a scripting system to customize the source code output, ability to separate classes by units, among other things.

But if you don't want to use a separate tool, and not even leave Delphi IDE, you can quickly generate entity classes using TAureliusConnection component. Simply configure the database connection on it, then right-click the component and choose "Generate entities from database...".



This will connect to the database, import the existing database structure, and open the export dialog with several options to customize the output source code. You can then select tables to export, choose naming policy for classes and properties, among other options. You can even preview the final source code in the "Preview" tab, before confirming. When you click "Ok" button, a new unit with the declares entities will be created in the same project of TAureliusConnection component.



- [✓] APPEARANCES
- [✓] AWARDSMANAGERS
- [✓] AWARDSPLAYERS
- [✓] AWARDSSHAREMANAGERS
- [✓] AWARDSSHAREPLAYERS
- [✓] BATTING
- [✓] BATTINGPOST
- [✓] COLLEGEPLAYING
- [✓] FIELDING
- [✓] FIELDINGOF
- [✓] FIELDINGOFSPLIT
- [✓] FIELDINGPOST
- [✓] HALLOFFAME
- [✓] HOMEGAMES
- [✓] MANAGERS
- [✓] MANAGERSHALF
- [✓] PARKS
- [✓] PITCHING

Quick Select: All None Invert 25 items selected

In "Mapping" tab you can choose the tables to export.

In "Advanced Settings" tab you can use the following options:

Naming options

You can define the default rule for naming classes, property/fields, associations and many-valued associations.

Class name comes from table name, property name comes from database field name. Those are the "base names". For associations you have "Use name from" field which specifies what will be used for the "base name". From the base name, the Format Mask will be applied. The "%s" in the format mask will be replaced by the base name. For example, the defualt Format Mask for class naming is "T%s" which means the class name will be the base name (usually Table Caption) prefixed with "T".

Additionally, some naming options allow you to:

- *Camel Case*: The first character of the base name or any character followed by underling will become upper case, all the other will become lower case. For example, if the base name in model is "SOME_NAME", it will become "Some_Name".

- *Remove underline*: All underlines will be removed. "SOME_NAME" becomes "SOMENAME". If combined with camel case, it will become "SomeName".
- *Singularize*: If the base name is in plural, it will become singular. "Customers" become "Customer", "Orders" become "Order". It also applies specified singularization rules for English language (e.g., "People" becomes "Person", etc.).

Dictionary

Data Modeler can also generate a dictionary with metadata for the classes. This dictionary can be used in queries in TMS Aurelius. To generate check "Generate Dictionary". You can also specify:

- *Global Var Name*: Defines the name of Delphi global variable to be used to access the dictionary.

Defaults

Defines some default behaviors when translating tables/fields into classes/properties. You can override this default behaviors individually for each class/property in the "Mappings" tab.

Field	**Description**
Association Fetch Mode	The default fetch mode used for associations. Default value is *Lazy*.
Association Cascade Type	The default cascade definition for associations. Options are "None" (no cascade) and "All but Remove" (all cascade options like save, update, merge, except remove cascade). Default value is *None*.
Many-Valued Association Fetch Mode	The default fetch mode used for many-valued associations. Default is *Lazy*.
Map One-to-One Relationship As	Defines how 1:1 relationships will be converted by default. A 1:1 relationship can be converted as a regular association (property) or can be considered an inheritance between two classes. Default value is *Association*.
Ancestor Class	Specifies the name of the class to be used as base class for all entity classes generated. Default value is empty, which means no ancestor (all classes will descend from *TObject*).
Dynamic Props Container Name	Specifies the default name for the property that will be a container for dynamic properties. If empty, then by default no property will be created in the class.
Check for Missing Sequences	Defines if exporting must abort (raise an error) if a sequence is not defined for a class. Options are: - *If supported by database*: if database supports sequences/generators, then raise an error if a sequence is not defined (default); - *Always*: always raise an error if a sequence is not specified; - *Never*: ignore any sequence check.

Options

Defines some other general options for exporting.

Field	**Description**
Generate Dictionary	Defines if the dictionary will be generated.
Register Entities	When checked, the generated unit will have an `initialization` section with a call to *RegisterEntity* for each class declared in the script (e.g., `RegisterEntity(TSomeClass);`). This will make sure that when using the generated unit, classes will not be removed from the final executable because they were not being used in the application. This option is useful when using the entity classes from a TMS XData server, for example.
Don't use Nullable<T>	By default, non-required columns will be generated as properties of type *Nullable<T>*. Check this option if you don't want to use Nullable, but instead use the primitive type directly (string, integer, etc.).

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 S**
Absolute Database	TAbsoluteDBConnectionAdapter	Aurelius.Drivers.AbsoluteDB	TABSDatabase	https://w
AnyDac	TAnyDacConnectionAdapter	Aurelius.Drivers.AnyDac	TADConnection	https://w
dbExpress	TDBExpressConnectionAdapter	Aurelius.Drivers.dbExpress	TSQLConnection	Delphi N
dbGo (ADO)	TDbGoConnectionAdapter	Aurelius.Drivers.dbGo	TADOConnection	Delphi N
Direct Oracle Access (DOA)	TDoaConnectionAdapter	Aurelius.Drivers.Doa	TOracleSession	https://w
ElevateDB	TElevateDBConnectionAdapter	Aurelius.Drivers.ElevateDB	TEDBDatabase	https://el
FIBPlus	TFIBPlusConnectionAdapter	Aurelius.Drivers.FIBPlus	TFIBDatabase	https://g

Technology	Adapter class	Declared in unit	Adapted Component	Vendor S
FireDac	TFireDacConnectionAdapter	Aurelius.Drivers.FireDac	TFDConnection	Delphi na
IBObjects (IBO)	TIBObjectsConnectionAdapter	Aurelius.Drivers.IBObjects	TIBODatabase	https://w
Interbase Express (IBX)	TIBExpressConnectionAdapter	Aurelius.Drivers.IBExpress	TIBDatabase	Delphi N
NativeDB	TNativeDBConnectionAdapter	Aurelius.Drivers.NativeDB	TASASession	https://w
NexusDB	TNexusDBConnectionAdapter	Aurelius.Drivers.NexusDB	TnxDatabase	https://w
SQL-Direct	TSQLDirectConnectionAdapter	Aurelius.Drivers.SqlDirect	TSDDatabase	https://w
SQLite	TSQLiteNativeConnectionAdapter	Aurelius.Drivers.SQLite	(not applicable)	TMS Aur
UniDac	TUniDacConnectionAdapter	Aurelius.Drivers.UniDac	TUniConnection	https://w
Unified Interbase (UIB)	TUIBConnectionAdapter	Aurelius.Drivers.UIB	TUIBDatabase	https://s
TMS RemoteDB Server	TRemoteDBConnectionAdapter	Aurelius.Drivers.RemoteDB	TRemoteDBDatabase	https://w
ZeosLib	TZeosLibConnectionAdapter	Aurelius.Drivers.ZeosLib	TZConnection	https://s

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 FireDAC adapter constructor receives a *TFDConnection* component.

```
MyConnection := TFireDacConnectionAdapter.Create(FDConnection1, False);
```

The adapter usually detects the SQL Dialect automatically, but you can force the adapter to use a specific dialect, using one of the following overloaded constructors.

Overloaded constructors

There are some overloaded versions of the constructor for all adapters:

```
constructor Create(AConnection: T; AOwnsConnection: boolean); overload; virtual;
constructor Create(AConnection: T; ASQLDialect: string;
  AOwnsConnection: boolean); overload; virtual;
constructor Create(AConnection: T; OwnedComponent: TComponent); overload;
virtual;
constructor Create(AConnection: T; ASQLDialect: string;
  OwnedComponent: TComponent); overload; virtual;
```

- *AConnection:* specify the database-access component to be adapted.
- *AOwnsConnection:* if true, the component specified in *AConnection* parameter will be destroyed when the IDBConnection interface is released. If false, the component will stay in memory.
- *ASQLDialect:* defines the SQL dialect to use when using this connection. If not specified, Aurelius will try to discover the SQL Dialect based on the settings in the component being adapted.
- *OwnedComponent:* specifies the component to be destroyed when the IDBConnection interface is released. This is useful when using data modules (see below).

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 ("Creating the adapter"), the *FDConnection1* 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;
```

Alternatively, you can inform a component to be destroyed when the interface is released. This is useful when you want to create an instance of a TDataModule (or TForm) and use an adapted component that is owned by it. For example:

```
MyDataModule := TConnectionDataModule.Create(nil);
MyConnection := TDBExpressConnectionAdapter.Create(MyDataModule.SQLConnection1, M
yDataModule);
```

The previous code will create a new instance of data module *TConnectionDataModule*, then create a *IDBConnection* by adapting the *SQLConnection1* component that is in the data module. When *MyConnection* is released, the data module (*MyDataModule*) will be destroyed. This is useful if you want to setup the connection settings at design-time, but want to reuse many instances of the data module in different connections (for multi-threading purposes, for example).

Referencing original component

If the component adapter also implements the *IDBConnectionAdapter* interface, you can retrieve the original adapted component. For example, given an IDBConnection that you know was created from a *TFireDacConnectionAdapter*, you can retrieve the TFDConnection object using the following code:

```
var
  MyConnection: IDBConnection;
  FDConnection: TFDConnection;
{...}
  FDConnection := (MyConnection as IDBConnectionAdapter).AdaptedConnection as TFD
Connection;
```

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.

Native Database Drivers

Aurelius provides native database connectivity. That means for some databases, you don't need to use a 3rd-party component adapter to access the database, but instead access it directly through the database client libraries.

The table below shows the existing native drivers and the connection classes.

Database	Driver Name	Connection class	Declared in unit
Microsoft SQL Server	MSSQL	TMSSQLConnection	Aurelius.Drivers.MSSQL
SQLite	SQLite	TSQLiteConnection	Aurelius.Drivers.SQLite

Creating a connection

To use the native driver from code, you usually just create an instance of the specific connection class passing to it a connection string that specifies how to connect to the database. The connection class implements the IDBConnection interface which you can then use. For example:

```
Conn := TMSSQLConnection.Create(
  'Server=.\SQLEXPRESS;Database=Northwnd;TrustedConnection=True');
Manager := TObjectManager.Create(Conn);
```

The connection string is a sequence of *ParamName=ParamValue* separated by semicolons (*Param1=Value1;Param2=Value2*). The param names are specific to each database driver as following.

SQLite Driver

Driver name is "SQLite", and the following parameters are supported:

Parameter	Type	Value	Example values
Database	String	A path to an SQLite database file to be open. Must be a valid SQLite file name, or even ":memory:" for in-memory databases.	C: \sqlite\mydb.sqlite :memory:
EnableForeignKeys	Boolean	Enables enforcement of foreign key constraints (using PRAGMA foreign_keys). Default is false.	True / False

Example:

```
Conn := TSQLiteConnection.Create('Database=C:
\sqlite\mydb.sqlite;EnableForeignKeys=True');
```

MSSQL Driver (Microsoft SQL Server)

Driver name is "MSSQL", and the following parameters are supported:

Parameter	Type	Value	Example values
Server	String	The name of a SQL Server instance. The value must be either the name of a server on the network, an IP address, or the name of a SQL Server Configuration Manager alias.	localhost .\SQLEXPRESS localhost,1522
Database	String	Name of the default SQL Server database for the connection.	northwnd
UserName	String	A valid SQL Server login account.	sa
Password	String	The password for the SQL Server login account specified in the UID parameter.	mypassword
TrustedConnection	Boolean	When "true", driver will use Windows Authentication Mode for login validation. Otherwise instructs the driver to use a SQL Server username and password for login validation, and the UserName and Password parameters must be specified. Default is False.	True/False
MARS	Boolean	Enables or disables multiple active result sets (MARS) on the connection. Default is False.	True/False
OdbcAdvanced	String	Semicolon-separated *param=value* pairs that will be added to the raw connection string to be passed to the SQL Server client.	StatsLog_On=yes;StatsLogFile=C:\temp\mssqlclient.log
LoginTimeout	Integer	Number of seconds to wait for a login request to complete before returning to the application.	10

Parameter	Type	Value	Example values
Driver	String	Specifies the SQL Server driver name (native or ODBC) to be used to connect to the SQL Server. Default is empty, which forces Aurelius to automatically select the most recent driver installed. You should mostly leave this empty, unless you have a reason to use a specific driver.	ODBC Driver 13 for SQL Server

Example:

```
Conn := TMSSQLConnection.Create(
  'Server=.\SQLEXPRESS;Database=Northwnd;TrustedConnection=True');
```

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 FireDac components, the adapter will look to the *DriverID* 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 explicitly 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
Firebird3 (*)	Firebird3	Aurelius.Sql.Firebird3	http://www.firebirdsql.org

SQL dialect	String identifier	Declared in unit	Database Web Site
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
SQL Anywhere	SqlAnywhere	Aurelius.Sql.SqlAnywhere	https://www.sap.com/products/sql-anywhere.html
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);
```

When using a native database driver, the SQL dialect is implicit from the driver you use and there is no need to specify it. The native driver already uses the sql dialects and schema importer units automatically.

(*) The difference between *Firebird* and *Firebird3* is that the latter uses boolean fields and identity fields by default. Please check Configuring SQL Dialects for more details on how to configure specific SQL dialects.

Configuring SQL Dialects

Some SQL Dialects have configuration options that you can use to fine tune how they work. For that you need to retrieve the original SQL Dialect object and then change specific properties. This is the pattern you use to retrieve a generator:

```
uses
  Aurelius.Sql.Register, Aurelius.Sql.Firebird;

var
  Generator: TFirebirdSQLGenerator;
begin
  Generator := (TSQLGeneratorRegister.GetInstance.GetGenerator('Firebird')
    as TFirebirdSQLGenerator);
  // Set Generator properties
end;
```

For all dialects you have the following options:

Properties

Name	Description
EnforceAliasMaxLength: Boolean	Makes sure that the field aliases used by Aurelius in SQL SELECT statements are not longer than the maximum size for field names. When this property is false, the field alias could be longer than maximum allowed for database and would cause errors in some databases, mainly Firebird. This property is there to avoid backward compatibility break, but usually you should always set it to true.
UseBoolean: Boolean	Specifies how boolean values will be represented in database. If False (default), boolean fields will be represented by CHAR(1) type. If True, boolean fields will be represented by BIT/TINYINT type.

For other dialects, you can just replace "Firebird" occurrences by the name of the different dialect. The following sections show the dialects that have specific properties you can configure:

MSSQL (Microsoft SQL Server)

Sample:

```
uses Aurelius.Sql.Register, Aurelius.Sql.MSSQL;
{...}
(TSQLGeneratorRegister.GetInstance.GetGenerator('MSSQL')
  as TMSSQLSQLGenerator).UseBoolean := True;
```

Properties

Name	Description
WorkaroundInsertTriggers: Boolean	Specifies if Aurelius should add statement to retrieve Identity values. Basically it would SET NOCOUNT ON and use a temporary table to retrieve the value. More technical info here: https://stackoverflow.com/a/42393871. This property is true by default to make sure things will work in most situations. But setting it to false might increase performance or work better when identity values are greater than 32 bits. In this case you could set it to false.

Firebird3 (Firebird 3.x)

Sample:

```
uses Aurelius.Sql.Register, Aurelius.Sql.Firebird3;
{...}
(TSQLGeneratorRegister.GetInstance.GetGenerator('Firebird3')
  as TFirebird3SQLGenerator).UseBoolean := False;
(TSQLGeneratorRegister.GetInstance.GetGenerator('Firebird3')
  as TFirebird3SQLGenerator).UseIdentity := False;
```

The code above makes the Firebird3 dialect to behave like the regular Firebird dialect (which is targeted at Firebird 2.x).

Properties

Name	Description
UseBoolean: Boolean	Specifies how boolean values will be represented in database. If False, then booleans will be represented by CHAR(1) type. If True, booleans will be represented by BOOLEAN type. Default is True.
UseIdentity: Boolean	Specifies how ID generators of type *SequenceOrIdentity* will behave. If False, then Sequences will be used. If True, Identity fields will be used. Default is True.

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
SQL Anywhere	SqlAnywhere	Aurelius.Schema.SqlAnywhere
SQLite	SQLite	Aurelius.Schema.SQLite

NOTE
When using a native database driver, the schema importer is implicit from the driver you use. The native driver already uses the sql dialects and schema importer units automatically.

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**	**FireDac**	**FIBPlus**	**IBO**	**IB**
AbsoluteDB		x									
DB2			x		x			x			
ElevateDB							x				
Firebird			x	x				x	x	x	

	Native	Absolute	AnyDac	dbExpress	dbGo	DOA	ElevateDB	FireDac	FIBPlus	IBO	IB
Interbase			x	x				x		x	x
MS SQL Server	x		x	x	x			x			
MySQL			x	x				x			
NexusDB											
Oracle			x	x		x		x			
PostgreSQL			x					x			
SqlAnywhere								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
SqlAnywhere	17
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

Component Library	**Versions**
AnyDac	5.0.3.1917
dbExpress	16.0
dbGo	Delphi 2010 and up
Direct Oracle Access	4.1.3.3
ElevateDB	2.32
FIBPlus	7.2
FireDac	Delphi XE5 and up
IBObjects	4.9.14
IBX	Delphi 2010 up to XE2
NativeDB	1.98
NexusDB	4.5023
SQL-Direct	6.3
UniDac	8.3.1
Unified Interbase (UIB)	2.5 revision 428 (01-Feb-2013)
ZeosLib	7.3

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.

TAureliusDBSchema Component

The *TAureliusDBSchema* component is a non-visual, design-time component that encapsulates the *TDatabaseManager* class, used to build, update and validate the schema structure of the database (tables, fields, foreign and primary keys, etc.).

TAureliusDBSchema and TDatabaseManager have equivalent functionality; the main purpose for TAureliusDBSchema component is to provide an alternative RAD approach: instead of instantiating a TDatabaseManager from code, you just drop a TAureliusDBSchema component, connects it to a TAureliusConnection component, and you are ready to go.

Key properties

Name	Description
Connection: TAureliusConnection	Specifies the TAureliusConnection component to be used as the database connection. TAureliusConnection acts as a connection pool of one single connection: it will create a single instance of IDBConnection and any manager using it will use the same IDBConnection interface for the life of the TAureliusConnection component. The IDBConnection interface will be passed to the TDatabaseManager constructor to create the instance that will be encapsulated.

Name	Description
ModelNames: string	The name(s) of the model(s) to be used by the manager. You can leave it blank, if you do it will use the default model. Two or more model names should be separated by comma. From the model names it will get the property TMappingExplorer component that will be passed to the TDatabaseManager constructor to create the instance that will be encapsulated.
DBManager: TDatabaseManager	The encapsulated TDatabaseManager instance used to perform the database operations.

Usage

As mentioned, TAureliusDBSchema just encapsulates a TDatabaseManager instance. So for all functionality (methods, properties), just refer to TDatabaseManager documentation and related topics that explain how to build, update and validate the database schema.

The encapsulated object is available in property *DBManager*. If you miss any specific method or property in TAureliusDBSchema, you can simply fall back to DBManager instance and use it from there. For example, the following methods are equivalent:

```
AureliusDBSchema1.UpdateDatabase;
AureliusDBSchema1.DBManager.UpdateDatabase;
```

Actually, the first method is just a wrapper for the second one. Here is how *TAureliusDBSchema.UpdateDatabase* method is implemented, for example:

```
procedure TAureliusDBSchema.UpdateDatabase;
begin
  DBManager.UpdateDatabase;
end;
```

Memory management

Here is the lifecycle of the encapsulated TDatabaseManager instance itself:

- The TDatabaseManager instance will be created on demand, i.e., when TAureliusDBSchema is created, the TDatabaseManager is not yet created. It will only be instantiated when needed.
- If the connection or model name is changed, the encapsulated TDatabaseManager instance will be **destroyed**. A new TDatabaseManager instance will be created with the new connection/model, when needed.

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.

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**.

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.

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 (action, 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;
- A new non-unique index (DBIndex);
- 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 warnings:

- 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.

Schema comparison options

You can use some properties to define how Aurelius will detect changes in existing schema.

Properties

Name	Description
IgnoreConstraintName: Boolean	When False, the validator will compare constraints (foreign key and unique key) by their name. If the name is different, they are considered different keys. This is the default for all databases except SQLite. When True, the validator will analyze the content of the foreign key, regardless the name. For example, if the foreign keys relates the same two tables, using the same fields, it's considered to be the same foreign key. You can set this option to True if you have created your database using a different tool than Aurelius, thus the foreign keys might have different names but you don't want Aurelius to recreated them.

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;
```

Other Properties and Methods

List of TDatabaseManager methods and properties not coverered by other topics in this chapter.

Properties

Name	**Description**
UseTransactions: Boolean	When True, all operations performed by TDatabaseManager will be executed in a transaction, i.e., the manager will automatically start a new transaction, and commit it at the end of operations, or rollback if there is an error. Nesting apply (if a transaction was already open, no commit or rollback will be performed). Default is False.

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.

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
  - AbstractEntity
  - Id
  - Table
  - Column

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;
```

Parameters
None.

Usage

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

AbstractEntity

Indicates that the class is an abstract entity: it can hold some mapping information that is inherited by concrete entity classes, but it will not persisted in the database.

Level: Class Attribute

Description
An abstract entity is a class that can have mapping information, but will not be persisted to the database. It allows you to have an entity classs that descend from an abstract entity class and inherit mapping information from it. This way you can have a class hiearchy without having to persist the ancestor classes, like you would have to do using the single-table or joined-tables inheritance strategies.

Not every mapping information can be used in abstract entities. In abstract entities you **can**:

- map primitive type columns (using Column attribute)
- map the entity id (like attributes Id and UnsavedValue)
- map associations
- use attributed-based events
- add attribute-based validation
- use global filters (using attributes like FilterDef and FilterDefParam).

Not supported:

- table-specific mapping attributes like Table, Sequence, UniqueKey, DBIndex, ForeignKey
- attributes related to inheritance strategy like Inheritance
- DiscriminatorColumn
- DiscriminatorValue
- PrimaryJoinColumn.

Constructor

```
constructor Create;
```

Parameters
None.

Example

```
[AbstractEntity]
[Automapping]
[Id('FId', TIdGenerator.IdentityOrSequence)]
TBaseEntity = class
strict private
  FId: Integer;
  [Column('CUSTOM_TAG', [])]
  FCustomTag: Integer;
  FCreatedAt: TDateTime;
  FUpdatedAt: Nullable<TDateTime>;
protected
  [OnInserting]
  procedure OnInserting;
  [OnValidate]
  function ValidateDates: IValidationResult;
public
  property Id: Integer read FTheId write FTheId;
  property CustomTag: Integer read FCustomTag write FCustomTag;
  property CreatedAt: TDateTime read FCreatedAt write FCreatedAt;
  property UpdatedAt: Nullable<TDateTime> read FUpdatedAt write FUpdatedAt;
end;

[AbstractEntity]
[Filter('Multitenant')]
[FilterDef('Multitenant', '{TenantId} = :tenantId')]
[FilterDefParam('Multitenant', 'tenantId', TypeInfo(string))]
TMultitenantEntity = class(TBaseEntity)
private
  [Column('TENANT_ID', [], 50)]
  FTenantId: string;
public
  property TenantId: string read FTenantId write FTenantId;
end;

// TCustomer will inherit all mapping information
// from TMultitenantEntity and TBaseEntity classes
[Entity]
[Automapping]
TCustomer = class(TMultitenantEntity)
strict private
  FName: string;
public
  property Name: string read FName write FName;
end;
```

Id

Specifies 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 or string.
    - *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 hyphens 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 hyphens but no curly brackets). The type of the property that identifies the entity should be string (witha 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 hyphens and no curly brackets). The type of the property that identifies the entity should be string (with a minimum length of 32 characters).
    - *SmartGuid*: Sequential GUID (Globally Unique Identifier) value optimized for the database being used. The generated sequential GUID will minimize clustered index fragmentation, which is an usual problem when using regular GUID's, causing performance loss. Aurelius will choose the best algorithm to generate the GUID sequence depending on the database being used. For most of them, the GUID will be sequential in its string format, which is optimum for most databases and also when you use string properties. For Microsoft SQL Server, for example, it will choose a different algorithm (sequential in the last bytes) which is best given the way SQL Server sorts GUID's internally. In general you should use SmartGuid generator instead of Guid since both achieve the same results but SmartGuid performs better.

> **NOTE**
> For composite id's the *AGenerator* parameter is ignored and *None* is used.

Usage

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

Table

Specifies 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
```

Column

Specifies 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); over
load;
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, Lazy);
  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 *NoInsert* 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;
```

Model

Specifies the model where the entity/class belongs to, in a multi-model design. It's an optional attribute.

Level: Class Attribute

Description
Use the *Model* attribute to tell Aurelius the model where that entity (class) belongs to. This attribute allows you to build multi-model applications, so that you can separate your mapping in multiple models. By using the Model attribute you can easily do it in a declarative way, specifying the model of each class.

You can add multiple Model attributes to the class, meaning that the class belongs to more than one model.

This attribute is optional and if omitted the class will be considered to belonging to the default model.

Constructor

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

Parameters

- *Name*: The name of the model.

Usage

```
[Entity, Automapping]
[Model('Sample')]
TCustomer = class(TObject)

[Entity, Automapping]
[Model('Sample')]
[Model('Security')]
TUserInfo = class(TObject)
```

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); overl
oad;
```

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. This is logical information for the model itself (metadata). This flag will not be used by Aurelius to set the NOT NULL flag of the underlying database field(s). You will still have to set the column as required in the JoinColumn attribute, if needed.

- *Cascade*: Defines how Aurelius will behave on the association when the container object is saved, deleted or updated.
  It's recommended that you use one of the predefined cascades, like *CascadeTypeAll*, *CascadeTypeAllButRemove* or *CascadeTypeAllRemoveOrphan*. For associations, *CascadeTypeAllButRemove* is the most recommended one.

  ```
  TCascadeType = (SaveUpdate, Merge, Remove, RemoveOrphan, Refresh, Evict, Fl
  ush);
  TCascadeTypes = set of TCascadeType;
  CascadeTypeAll = [Low(TCascadeType)..High(TCascadeType)] - [TCascadeType.Re
  moveOrphan];
  CascadeTypeAllRemoveOrphan = CascadeTypeAll + [TCascadeType.RemoveOrphan];
  CascadeTypeAllButRemove = CascadeTypeAll - [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*: When object is refreshed from database, the associated object will also be refreshed.
  - *RemoveOrphan*: Used only in Many-Valued Associations.
  - *Evict*: When object is evicted from manager, the associated object will also be evicted.
  - *Flush*: If an object is flushed explicitly, the associated object will also be flushed. This cascade doesn't have any effect if *Flush* is called for all objects in manager (without parameter).

Usage

```
[Association([], CascadeTypeAllButRemove)]
[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.

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: TColumnProps); overload;
constructor Create(Name: string; Properties: TColumnProps;
  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, Lazy);
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 *Required* 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 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.

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<TAddress>*, 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); overl
oad;
constructor Create(AProperties: TAssociationProps; Cascade: TCascadeTypes;
  MappedBy: string); overload;
```

Parameters

- *AProperties*: Specifies some general properties for the association. Valid values are:

  ```
  TAssociationProp = (Lazy, Required);
  TAssociationProps = set of TAssociationProp;
  ```

  - *Lazy*: The associated list is not loaded together with the current object. Lazy-Loading is used. In a SELECT operation, Aurelius will only retrieve the Id of the parent object. The list will only be loaded when the application effectively needs it (e.g., when user references property *MyObject.AssociatedList*). 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 after the parent is loaded, the associated list will be immediately load, but still with another SELECT statement. For lists, since eager mode will not improve performance, it's always recommended to use Lazy mode, unless you have a very specific reason for not doing so, like for example, you will destroy the object manager after retrieving objects and lazy-loading the lists will not be further possible.
  - *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.
  It's recommended that you use one of the predefined cascades, like *CascadeTypeAll*,

CascadeTypeAllButRemove or *CascadeTypeAllRemoveOrphan*. For many-valued associations, *CascadeTypeAll* or *CascadeTypeAllRemoveOrphan* are the recommended ones.

```
TCascadeType = (SaveUpdate, Merge, Remove, RemoveOrphan, Refresh, Evict, Fl
ush);
TCascadeTypes = set of TCascadeType;
CascadeTypeAll = [Low(TCascadeType)..High(TCascadeType)] - [TCascadeType.Re
moveOrphan];
CascadeTypeAllRemoveOrphan = CascadeTypeAll + [TCascadeType.RemoveOrphan];
CascadeTypeAllButRemove = CascadeTypeAll - [TCascadeType.Remove];
```

  - *SaveUpdate*: 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*: When object is refreshed from database, the associated list will be reloaded. If the list is proxied (lazy-loaded), then the proxy will be reset (unloaded), and objects in list won't be refreshed. If the list is not proxied, objects in list will be refreshed.
  - *RemoveOrphan*: When a detail (child) object is removed from a list, it will also be deleted (removed from database and destroyed). If *RemoveOrphan* is not present, then the child object will not be deleted, just the association with the parent object will be removed (i.e., the foreign key column will be set to null).
  - *Evict*: When object is evicted from manager, the associated object will also be evicted.
  - *Flush*: If an object is flushed explicitly, the associated objects in the list will also be flushed. This cascade doesn't have any effect if *Flush* is called for all objects in manager (without parameter).

- *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([], CascadeTypeAllRemoveOrphan, '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([], CascadeTypeAllRemoveOrphan)]
  [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.

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: TColumnProps); overload;
constructor Create(Name: string; Properties: TColumnProps;
  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, Lazy);
  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>;
```

OrderBy

Specifies the default order of the items in a many-valued association.

Level: Field/Property Attribute

Description
Use *OrderBy* attribute to define in which order the objects in a many-valued association (collection) will be loaded from the database. If no OrderBy attribute is present, no order will be defined (no ORDER BY clause will be added to the SELECT statement that retrieves the records) and items will be loaded according to the default ordering used by the database server. Note that after the items are loaded from the database, no further ordering is performed - this attribute only enforces ordering at database level, not memory level. Thus, if you later manually add new items to the collection in an unsorted order, they will remain that way.

You can specify one or more member names (property or field names) in this attribute (not database column names). Multiple member names must be separated by comma (,). You can use the same member names that you can use when ordering results in a query.

The default order direction is ascending. You can specify a descending order by appending " DESC" (with space) after the member name.

You can also order by members of associated objects. To do that, prefix the member name with the name of the association field/property followed by a "." (dot). Nested associations can be used. For example, if your class has a property "Customer" which in turn has a property "Country", you can order by country's name using "Customer.Country.Name".

Constructor

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

Parameters

- *MemberNames*: Contains the name(s) of the member(s) used to order the collection. Multiple member names must be separated by comma. Associated members must be prefixed with association name followed by dot. You can optionally use "DESC" suffix to order by descending direction.

Usage

```
TTC_Invoice = class
private
  [ManyValuedAssociation([], CascadeTypeAll)]
  [OrderBy('Product.Name, Category DESC')]
  FItems: TList<TTC_InvoiceItem>;
```

Where

Specifies a SQL expression to be added the WHERE clause of the final SQL used to retrieve entities or the items of a many-valued association.

Level: Class or Field/Property Attribute

Description
Use *Where* attribute to define additional filter (SQL expression) to the final SQL used to retrieve a specified entity, a list of entities or items of a many-valued association.

Just like in the query SQL condition, be aware that the SQL clause will be just injected in the SQL statement, you must be sure it will work. You can also use property names 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 the context. For example, "*{Deleted}* = '*F*'".

If you apply the attribute to a field/property, it must be a many-valued association, and the WHERE clause will only apply for that list.

If you apply the attribute to a class, it will apply to **any** situation where entities of that class are retrieved. When you find a single entity, or when you query entities of that class, the filter will be applied. Even if the entity is an association (many-to-one) of a parent entity, the filter will be applied. For example, suppose a *TInvoice* class has a *Customer* property of type *TCustomer*. If TCustomer entity has a [Where] attribute it will be applied when retrieving the Customer of that TInvoice instance. Even if the associated customer exists in the database, if it's filtered out by the WHERE clause, the *TInvoice.Customer* property will come as nil value.

You can add multiple Where attributes in same class or property. They will all be combined with the AND operator.

Constructor

```
constructor Create(const ASqlClause: string);
```

Parameters

- *ASqlClause*: The SQL expression that will be added to the WHERE clause to filter the entity or many-valued association.

Usage

TCustomer entities will not be retrieved if the *Deleted* field is equal to 'T':

```
[Entity, Automapping]
[Where('{Deleted} <> ''T''')]
TCustomer = class
private
  FId: integer;
  FName: string;
```

The *FNewCustomers* list will only bring *TCustomer* objects if the *Status* field is equal to 'New'. Note that the Where clause above of the TCustomer entity will still apply, meaning the Status must be 'New' **and** Deleted must not be 'T':

```
TParent = class
private
  [ManyValuedAssociation([], CascadeTypeAll)]
  [Where('{Status} = ''New''')]
  FNewCustomers: TList<TCustomer>;
```

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
```

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
```

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)
```

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)
```

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); overl
oad;
```

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
```

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. If you want to create a non-exclusive (non-unique) index, use DBIndex attribute instead.

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
```

DBIndex

Defines a non-exclusive index for the table.

Level: Class Attribute

Description
Use *DBIndex* if you want to define a database-level non-exclusive index in the table associated with the class. The index will mostly be used to improve performance when executing queries. If you want to create an unique index, use UniqueKey attribute instead.

Constructor

```
constructor Create(const Name, Columns: string);
```

Parameters

- *Name*: The name of the Index. When updating the database, this is what Aurelius will check to decide if the index needs to be created or not.
- *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

```
[DBIndex('IDX_INVOICE_DATE', 'ISSUEDATE')]
TTC_Invoice = class
```

ForeignKey

Defines the name of a foreign key.

Level: Field/Property Attribute

Description
Use *ForeignKey* to define a custom name for the foreign key generated by an association or many-valued association. This attribute is optional even when Automapping is not specified. When this attribute is not present, Aurelius will automatically choose a name for the foreign key.

Constructor

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

Parameters

- *AName*: Specifies the name of the foreign key.

Usage

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

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);
```

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;
```

Parameters
None.

Usage

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

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;
```

Parameters
None.

Usage

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

Version

Indicates that the class member (field/property) holds the version of the entity, to be used in versioned concurrency control.

Level: Field/Property Attribute

Description
When adding this attribute to any member, Aurelius automatically enabled versioned concurrency control on entities of that class. This means that Aurelius will make sure that updates on that entity will only happen if no other user changed entity data in the meantime.

To accomplish that, the entity must hold the "version" value, so Aurelius knows which is the current version of that entity. You must thus add the *Version* attribute to any member of the class (field or property) so Aurelius knows where to save the version value.

The field/property type **must** be of Integer type.

Constructor

```
constructor Create;
```

Parameters
None.

Usage

```
[Entity]
[Automapping]
TCustomer = class(TObject)
private
  [Version]
  FVersion: integer;
```

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);
```

Parameters

- *AText*: The text to be associated with class, field or property.

Usage

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

Automapping Feature

Automapping is an Aurelius feature that allows you to map 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. All letters become uppercase and upcase characters in the middle of the name are preceded by underline. For example, class "TCustomer" will be mapped to table "CUSTOMER", and class "TMyInvoice" will be mapped to table "MY_INVOICE"

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. All letters become uppercase and upcase characters in the middle of the name are preceded by underline. For example, field "FBirthday" is mapped to table column "BIRTHDAY" and field "FFirstName" is mapped to table column "FIRST_NAME".

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.

Customizing automapping

You can customize the naming in automapping mechanism by passing an engine class to the automapping attribute:

```
  [Entity, Automapping(TMyAutomapping)]
  TMyEntity = class
```

`TMyAutomapping` must inherit from `TAutomappingEngine`, and you can then override some methods to use your custom naming:

```
  TMyAutomapping = class(TAutomappingEngine)
  strict protected
    function FieldNameToSql(const Value: string): string; override;
  public
    function GetTableName(Clazz: TClass): string; override;
    function GetSequenceName(Clazz: TClass): string; override;
  end;

{ TMyAutomapping }

function TMyAutomapping.FieldNameToSql(const Value: string): string;
begin
  Result := Value;
end;

function TMyAutomapping.GetSequenceName(Clazz: TClass): string;
begin
  Result := 'seq' + GetTableName(Clazz);
end;

function TMyAutomapping.GetTableName(Clazz: TClass): string;
begin
  Result := Copy(Clazz.ClassName, Pos('_', Clazz.ClassName) + 1);
end;
```

Nullable 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 `Bcl.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 a global Nullable variable named *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
```

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

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.

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:

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

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

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

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

If the underlying storage column is a memo, text or blob subtype text, Aurelius will make sure that the column will have the proper text value.

If it's a raw binary blob, the string will be saved using Unicode encoding.

You should also use AsUnicodeString for reading data from blobs. If the database blob has a memo value, the DB access component will use its default encoding/charset to read the text, and Aurelius will force the binary data to be kept in memory (in TBlob value) as Unicode encoding. Thus using AsUnicodeString will ensure you will read the correct string value.

For backward compatibility, you can use *AsString* property. That will read/save values using ANSI encoding. Unless you have a specific reason for using AsString, you should always use AsUnicodeString.

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;
```

Loaded and Available properties

TBlob provides two boolean properties: *Loaded* and *Available*, and they refer to the status of data availability when blob content is configured to be lazy-loaded.

Available property allows you to check if blob content is available, without forcing the content to be loaded. If Available is true, it means that the blob content is already available in memory, even if it's empty. If it's false, it means the blob content is not available in memory and a request must be performed to load the content.

Loaded property behaves in a similar way. When Loaded is true, it means that the blob content of a lazy-loaded blob was already loaded from the database. If Loaded is false, it means the content was not loaded.

The difference between Loaded and Available is that when a new TBlob record is created, Available is true (because data is available - it's empty) and Loaded is false (because no content was loaded - because there is no content to load).

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 Invoice table.

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* (Proxy<T> type is declared in unit `Aurelius.Types.Proxy`).
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.Create;
begin
  FItems.SetInitialValue(TList<TInvoiceItem>.Create);
end;

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

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

Proxy<T> Available property

Available property allows you to check if proxy object is available, without forcing it be loaded. If Available is true, it means that the proxy object is already available in memory, even if it's empty. If it's false, it means the object is not available in memory and a request must be performed to load the content. In other words, Available property indicates if accessing the object will fire a new server request to retrieve the object.

Proxy<T> Key property

You can read *Key* property directly from the *Proxy<T>* value to get the database values for the foreign key used to load this proxy. This way you have access to the underlying database value without needing to force the proxy to load. Note that Key might not be always available - it will be filled by the object manager when the data is loaded from the database. If you set the proxy value manually, Key value might differ from the actualy id of the object in Proxy<T>.

Inheritance Strategies

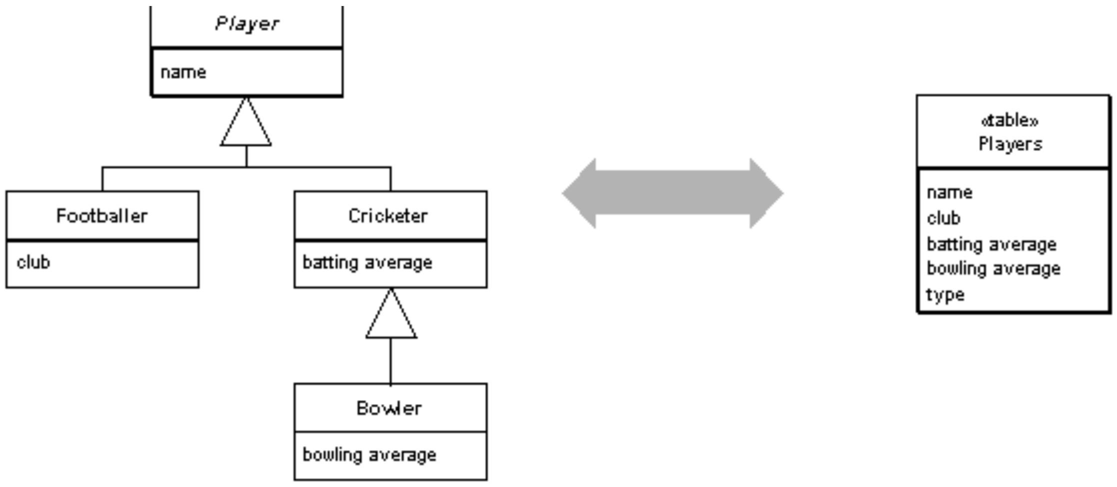
There are currently two strategies for you to map class inheritance into the relational database:

- 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.

Single Table Strategy

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



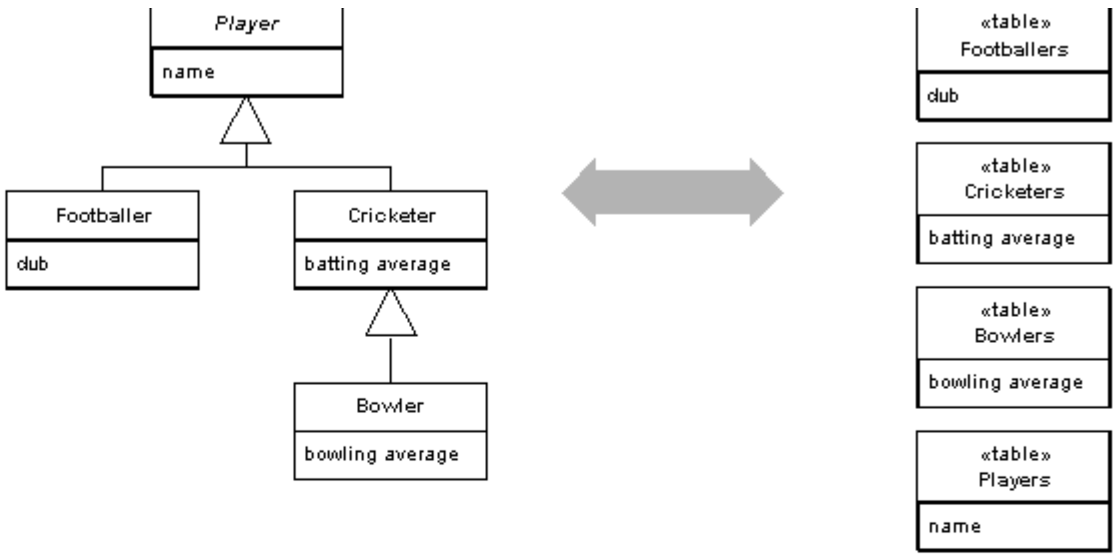
The concrete class of the object is indicated by the values in a special column in the table named *discriminator column*. This column is specified by the programmer and its content is used to identify the real class of the object. The discriminator column must be of string or integer type.

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.

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.

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 FAppointmentDat
e;
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).

Mapping Examples

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

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.
```

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.DontUpda
te])]
    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;

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

  [Entity]
  [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.
```

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.DontUpda
te])]
    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.
```

Registering Entity Classes

Aurelius doesn't require you to register the entity classes. Just by adding Entity attribute to the class it knows that the class is mapped and it will add it automatically to the default model or a model you have explicitly specified.

However, if you don't use the class anywhere in your application, the linker optimizer will remove it from the final application executable, and Aurelius will never know about it (since it retrieves information at runtime). There are situations where this can happen very often:

- You have just started your application and wants Aurelius to create the database structure for you, but you still didn't use any of your classes. Aurelius will not create the tables since the classes just don't exist in executable.
- You are creating a server application, especially using XData, without any specific server-side logic. You will notice that XData will respond to 404 (not found) to the URL resource addresses corresponding to your classes. This is just because XData server doesn't know about those classes.

To solve these kind of problems, all you would have to do is use the class somewhere in your application. It could be a simple "TMyClass.Create.Free". Nevertheless, to help you out in this task, there is a function *RegisterEntity* in unit `Aurelius.Mapping.Attributes` that you can use to make sure your class will be "touched" and thus included in final executable.

So in the same unit you have your classes mapped you can optinally just call *RegisterEntity* in initialization section for all classes to make sure they will be present in application:

```
unit MyEntities;

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

type
  [Entity, Automapping]
  TCustomer = class
  private
    FId: integer;
  {...}

initialization
  RegisterEntity(TCustomer);
  RegisterEntity(TCountry);
  RegisterEntity(TInvoice);
  {...}
end.
```

Multi-Model Design

Most Aurelius applications uses single-model mapping. This means that all classes you map belongs to the same model. So for example when retrieving objects from the database, or creating the database structure, objects of all mapped classes will be available.

But in some situations, you might need to have multiple mapping models. For example, you want your *TCustomer* entity class to belong to your default model, but you want *TUserInfo* entity class to belong to a different model ("Security" model for example). There are several reasons for this, for example:

- You have more than one database you want to access from your application, with totally different structures.
- You have some objects that you don't want to save to a database, but just want to use them in memory (using SQLite memory database).
- You use other tools that uses Aurelius and you want to logically separate your entity classes for that. For example, when using TMS XData, you might want to use different models to create different server setups.
- Any other reason you have to separate your classes into different mappings.

There are two ways to define multiple mapping models: using Model attribute (preferrable), or manually creating a mapping setup. The following topics describe the two options and explain the concepts of multi-model design in Aurelius.

Multi-Model Step-By-Step

This topic explains very shortly how to use multiple mapping models with Aurelius. For more details about each step, please refer to main Multi-Model Design chapter.

1. Add a *Model* attribute to each class indicating the model where the class belongs to:

```
[Entity, Automapping]
[Model('Sample')]
TSampleCustomer = class
{...}

[Entity, Automapping]
[Model('Security')]
TUserInfo = class
{...}

 // no model attribute means default model
[Entity, Automapping]
TCustomer = class
  {...}
```

2. Retrieve the TMappingExplorer object associated with the model:

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

var
  SampleExplorer: TMappingExplorer;
  SecurityExplorer: TMappingExplorer;
  DefaultExplorer: TMappingExplorer;
begin
  SampleExplorer := TMappingExplorer.Get('Sample');
  SecurityExplorer := TMappingExplorer.Get('Security');
  DefaultExplorer := TMappingExplorer.Default;
```

3. Create an object manager using the proper mapping explorer:

```
SampleManager := TObjectManager.Create(SampleConnection, SampleExplorer);
SecurityManager := TObjectManager.Create(SecurityConnection, SecurityExplorer);
DefaultManager := TObjectManager.Create(MyConnection, DefaultExplorer);
```

or simply:

```
SampleManager := TObjectManager.Create(SampleConnection, TMappingExplorer.Get('Sa
mple'));
SecurityManager := TObjectManager.Create(SecurityConnection,
TMappingExplorer.Get('Security'));
DefaultManager := TObjectManager.Create(MyConnection, TMappingExplorer.Default);
```

For default manager you can simply omit the explorer:

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

4. You can also use the explorers in other needed places. For example, to create a database structure:

```
// this example creates tables for "Sample" model in
// a SQL Server database using FireDac,
// and "Security" model in a in-memory SQLite database
SampleConnection := TFireDacConnectionAdapter.Create(FDConnection1, false);
DBManager := TDatabaseManager.Create(SampleConnection, TMappingExplorer.Get('Samp
le'));
DBManager.UpdateDatabase;
DBManager.Free;

SecurityConnection := TSQLiteNativeConnectionAdapter.Create(':memory:');
DBManager := TDatabaseManager.Create(SecurityConnection, TMappingExplorer.Get('Se
curity'));
DBManager.UpdateDatabase;
DBManager.Free;
```

Using Model attribute

Defining multiple mapping models in Aurelius is very straightforward if you use Model attribute. Basically all you need to do is annotate a class with the model attribute telling Aurelius the model where that class belongs to. For example, the following code specifies that class *TUserInfo* belongs to model "Security":

```
// TUserInfo belongs to model "Security"
[Entity, Automapping]
[Model('Security')]
TUserInfo = class
  {...}
```

You can also include the class in multiple models, just by adding the *Model* attribute multiple times. The following example specifies that the class *TSample* belongs to both models "Security" and "Sample":

```
// TSample belongs to model "Security" and "Sample"
[Entity, Automapping]
[Model('Security')]
[Model('Sample')]
TSample = class
  {...}
```

In Aurelius, every mapped class belongs to a model. If you omit the Model attribute (since it's optional), the class will be included in the default model.

```
// This class belongs to default model
[Entity, Automapping]
TCustomer = class
  {...}
```

If you want to add a class to both default model and a different model, you can just add it to default model (named "Default"):

```
// TUser belongs to both "Security" and default model
[Entity, Automapping]
[Model('Security')]
[Model('Default')]
TUser = class
  {...}
```

You can then use the different models by retrieving the TMappingExplorer instance associated with a model.

TMappingExplorer

After Aurelius retrieves information about your mapping, it saves all that info in an object of class *TMappingExplorer* (declared in unit `Aurelius.Mapping.Explorer`). In other words, a TMappingExplorer object holds all mapping information. Although in some cases you might never need to deal with it directly, it is a key class when using Aurelius because that's the class it uses to perform all its operations on the entities.

When you create an object manager, for example, you do it this way:

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

And that is the same for the database manager. You can omit the parameter and create it like this:

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

But this just means that you are telling the manager to use the default mapping explorer. It's the equivalent of doing this:

```
Manager := TObjectManager.Create(DBConnection, TMappingExplorer.Default);
```

Retrieving a TMappingExplorer instance

As explained above, in single-model applications you will rarely need to deal with TMappingExplorer instances. All the mapping is available in the default TMappingExplorer instance, which is used automatically by the object manager and database manager. But when you have multiple mapping models in your application, you will need to tell the manager what mapping model it will be using. To help you in that task, Aurelius provides you with global TMappingExplorer instances. Aurelius creates (in a lazy way) one instance for each mapping model you have.

To retrieve the TMappingExplorer instance associated with a model, just use the *TMappingExplorer.Get* class property passing the model name. In the following example, the object manager will use the "Security" model, instead of the default one.

```
Manager := TObjectManager.Create(DBConnection, TMappingExplorer.Get('Security'));
```

Note that you don't need to destroy the TMappingExplorer instance in this case, those are global instances that are destroyed automatically by Aurelius when application terminates. To retrieve the default instance, use the *Default* property:

```
Manager := TObjectManager.Create(DBConnection, TMappingExplorer.Default);
```

Creating a TMappingExplorer explicitly

Usually you don't need to create a mapping explorer explicitly. As mentioned above, Aurelius automatically creates a default mapping explorer (available in class property *TMappingExplorer.Default*) and always uses it in any place where a TMappingExplorer object is needed but explicitly provided (like when creating the object manager). And you can also retrieve a mapping explorer instance for a specific model. So it's very rare you need to create one your own.

But if you still need to do so, you can explicitly create a TMappingExplorer object using either a mapping setup or a model name. Here are the following available constructors.

```
constructor Create(ASetup: TMappingSetup); overload;
constructor Create(const ModelName: string); overload;
```

To create a mapping explorer based on a mapping setup, just pass the setup to the constructor (check here to learn how to create mapping setups).

```
MyExplorer := TMappingExplorer.Create(MyMappingSetup);
```

Or, alternatively, you can just pass the model name. The explorer will only consider all entities belonging to the specified model:

```
MyExplorer := TMappingExplorer.Create('Sample');
```

NOTE
You are responsible to destroy the TMappingExplorer instance you create explicitly.

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 must be provided to it, in order for the object manager to retrieve meta information about the mapping (or the default TMappingExplorer instance will be used).

To create a TMappingExplorer object explicitly, you can 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.

NOTE
Using Model attribute is a much easier way to create multi-model Aurelius applications when compared to mapping setup. Check the step-by-step topic to learn more about it.

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;
```

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 default 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 manage
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.

Mapped Classes

By default, TMS Aurelius maps all classes in the application marked with Entity attribute, for the default model. 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 class
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, for the **default model** - the classes without a explicit Model attribute.

If you want the mapping setup to automatically load entities from another model, just set the `ModelName` property:

```
  MapSetup.ModelName := 'MyModel';
```

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>;
class function GetDefaultClasses: TEnumerable<TClass>;
class function GetModelClasses(const ModelName: string): TEnumerable<TClass>;
```

Helper functions that return classes in the application marked with Entity attribute.
You can call *GetModelClasses* to retrieve entity classes belonging to the model specified by ModelName.
You can call *GetDefaultClasses* to retrieve entity classes belonging to the default model (either classes with no *Model* attribute or belonging to model "Default").
Or you can use *GetEntityClasses* to retrieve all entity classes regardless of the model they belong to. 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. Note that if *ModelName* is empty string when calling GetModelClasses, model will be ignored and all classes marked with Entity attribute, regardless of the model, will be retrieved.

Calling *GetModelClasses('')* is equivalent to calling *GetEntityClasses*.

Calling *GetModelClasses(TMappedClasses.DefaultModelName)* is equivalent to calilng *GetDefaultClasses*.

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

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.

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); ov
erload;
  constructor Create(Name: string; Properties: TColumnProps; Precision, Scale: In
teger); 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

The overloaded *Create* methods of *TDynamicColumn* are very similar to the ones used in Column attribute. The TDynamicColumn 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.

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; defau
lt;
  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
  // Saving 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.

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(
    (Linq['HairStyle'] = THairStyle.Long) and
    Linq['Extra'].Like('%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);

// use BlobField as usual
BlobField := DS.FieldByName('Photo') as TBlobField;
```

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.

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 mapping model different than the default.

```
Manager := TObjectManager.Create(MyConnection, MyMappingExplorer);
// or
Manager := TObjectManager.Create(MyConnection, TMappingExplorer.Get('MyModel'));
```

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 including a transient into the manager. To effectively persist updates, you need to call *Flush* method.

```
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

Performs all changes made to the managed objects, usually to update objects.

```
Customer1 := Manager.Find<TCustomer>(1);
Customer2 := Manager.Find<TCustomer>(2);
Customer1.Email := 'company@address.com';
Customer2.City := 'Miami';

// Update Customer1 e-mail and Customer2 city in database
Manager.Flush;
```

Flush method for single entity

Commit to the database changes made to a single object - it's an overloaded version of *Flush* method that receives an object:

```
Customer1 := Manager.Find<TCustomer>(1);
Customer2 := Manager.Find<TCustomer>(2);
Customer1.Email := 'company@address.com';
Customer2.City := 'Miami';

// Update Customer1 e-mail - Customer2 changes are not persisted
Manager.Flush(Customer1);
```

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 *ManagedCustomer*. Customer reference will still point to an unmanaged object, so two instances of TCustomer will be in memory.

Replicate method

The *Replicate* method behaves exactly the same as the merge method above. The only difference is that, in the example above, if no Customer with id 12 is found in the database, instead of raising an exception, Replicate will create the new customer with that id.

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);
```

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>
  .Where(Linq['Name'] = 'Mia Rosenbaum')
  .List;
```

CreateCriteria<T> method

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

```
Results := Manager.CreateCriteria<TTC_Customer>
  .Where(Linq['Name'] = 'Mia Rosenbaum')
  .List;
```

Evict method

Use to evict (dettach) an entity from the manager:

```
Manager.Evict(Customer);
```

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);
```

FindCached<T> method

Use *FindCached* method to retrieve an object from the manager's cache, given its Id.

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

This method is similar to Find *method* but the difference is that if the object is not in manager cache, Aurelius will not hit the database to retrieve the objec - instead, it will return nil. Because of that, this method should be used only to check if the object is already in the manager - it's not useful to retrieve data from database.

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

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

IsCached<T> method

Checks if an object of the specified class with the specified id is already loaded in the object manager.

```
if not Manager.IsCached<TCustomer>(CustomerId) then
  ShowMessage('Not loaded');
```

You can use the non-generic version as well:

```
if not Manager.IsCached(TCustomer, CustomerId) then
  ShowMessage('Not loaded');
```

HasChanges method

Checks if a call to *Flush* will result in database operations. In other words, verifies if any object in the manager was modified since it was loaded from the database. *HasChanges* checks not only if properties were modified but also if any lists of the object has been modified (an item was added or removed).

```
Customer := Manager.Find<TCustomer>(1);
Customer.Email := 'company@address.com';
if Manager.HasChanges then
  // code will enter here as Email was changed
```

HasChanges checks for any change in all objects in manager. You can use an overload that receives an object as parameter, to check if that specific entity was modified:

```
if Manager.HasChanges(Customer) then // only checks for changes in 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>;
```

UseTransactions property

When true, all internal operations peformed by the object manager (Save, Update, Merge, Remove, etc.) as enclosed between transactions (it means if no transaction is active, the manager will create one just for the operation, then later commit). This is needed because even a single manager operation can perform several SQL statements in the database (due to cascades for example).

If false, the manager won't create new transactions, and it's up to you to make sure that a transaction is active, otherwise if the internal process fails, some records might become updated in the database, while others don't.

The default value of this property is controlled globally by the TGlobalConfiguration object.

DeferDestruction property

When true, all manager operations that destroy objects, such as Remove, will not immediately destroy them, but instead hold them in an internal list to be destroyed when the object manager is destroyed.

This can be useful if you still have references to the removed object in places like lists or datasets, and such references might still be used until manager is destroyed. This sometimes avoids invalid pointer operations and access violations caused by referencing those destroyed instances.

For backward compatibility, default value is false.

TAureliusManager Component

The *TAureliusManager* component is a non-visual, design-time component that encapsulates the TObjectManager class, used to persist and retrieve objects in database.

TAureliusManager and *TObjectManager* have equivalent functionality, the main purpose for TAureliusManager component is to provide an alternative RAD approach: instead of instantiating a TObjectManager from code, you just drop a TAureliusManager component, connects it to a TAureliusConnection component, and you are ready to go.

Key properties

Name	Description
Connection: TAureliusConnection	Specifies the TAureliusConnection component to be used as the database connection. Any persistence operation performed by the manager will use the connection provided by the TAureliusConnection component. TAureliusConnection acts as a connection pool of one single connection: it will create a single instance of IDBConnection and any manager using it will use the same IDBConnection interface for the life of the TAureliusConnection component. The IDBConnection interface will be passed to the TObjectManager constructor to create the instance that will be encapsulated.
ModelName: string	The name of the model to be used by the manager. You can leave it blank, if you do it will use the default model. From the model name it will get the property TMappingExplorer component that will be passed to the TObjectManager constructor to create the instance that will be encapsulated.
ObjManager: TObjectManager	The encapsulated TObjectManager instance used to perform the database operations.

Usage

As mentioned, TAureliusManager just encapsulates a TObjectManager instance. So for all functionality (methods, properties), just refer to TObjectManager documentation and related topics that explain how to save objects, update them, retrieve, querying, etc.

The encapsulated object is available in property *ObjManager*. If you miss any specific method or property in TAureliusManager, you can simply fall back to ObjManager instance and use it from there. For example, the following methods are equivalent:

```
AureliusManager1.Save(Customer);
AureliusManager1.ObjManager.Save(Customer);
```

Actually effectively, the first method is just a wrapper for the second one. Here is how TAureliusManager.Save method is implemented, for example:

```
procedure TAureliusManager.Save(Entity: TObject);
begin
  ObjManager.Save(Entity);
end;
```

TObjectManager memory management

All entities are managed inside the TObjectManager instance, using the regular memory management mechanism of such class. The only thing you should be aware is the lifecycle of the TObjectManager instance itself:

- The TObjectManager instance will be created on demand, i.e., when TAureliusManager is created, the TObjectManager is not yet created. It will only be instantiated when needed;
- If the connection or model name is changed, the encapsulated TObjectManager instance will be **destroyed** (and as a consequence all entities managed by it). A new TObjectManager instance will be created with the new connection/model, when needed.

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

Manually adding ownership

The manager owns all entity objects it manages, in other words, which are persisted. But sometimes you want to tell the manager to own an object even before a persistence operation, using the `AddOwnership` method. This is very common when using the `Save` method. The object passed to `Save` will only be owned by the manager if the operation is successful. If the operation fails, you would have to manually destroy it.

To improve this workflow, you can explicitly tell the manager to own the entity to be saved in advance, so regardless if the `Save` method fails, or not, the manager will always destroy the object:

```
Customer := TTC_Customer.Create;
Customer.Name := 'Customer Name';

// Tell the manager to destroy Customer
ObjectManager1.AddOwnership(Customer);

// Customer will always be destroyed even
// if Save fails
ObjectManager1.Save(Customer);
```

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.Find<TCustomer>
  .Add(Linq['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.Find<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.Find<TCustomer>
  .Where(Linq['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.
```

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 saved, 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.

Updating Objects - Flush

You modify objects using the TObjectManager method *Flush*. The state of all objects persisted in object manager is tracked by it. Thus, if you change any property of any object after it's loaded by the database, those changes will be updated to the database when Flush method is called. Consider the example below:

```
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.

It's possible that, by any reason, you want to update a detached object, in other words, an object that is not being tracked (persisted) by the manager. This might happen, for example, if you loaded an object with the manager, then destroyed the manager but kept the object reference (using *TObjectManager.OwnsObjects = false*). Or, for example, if you created the object instance yourself, and set its id property to a valid value. In this case the object is not in the manager, but you want to update the database using the object you have.

In this case, you can use *Update* method. This method will just take the passed transient instance and attach it to the TObjectManager. Then when you later call Flush, the changes will be persisted to the database. Note that when you call Update, no data is retrieved from the database. This means that the object manager doesn't know the original state of the object (data saved in database). The consequence is that **all** properties of the object passed to Update method will later be saved to the database when Flush is called. So you must be sure that all the persistent properties of the object have the correct value to be saved to the database.

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

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.

Merging

If you call *Update* method passing, say, *Object1*, but there was already another object attached to the TObjectManager with the same id (*Object2*), an exception will be raised. In this case, you can use Merge method to merge a transient object ("outside" the manager) into a persistent object ("inside" the manager).

Flushing a single object

Calling *Flush* might be slow if you have many entities in the manager. Flush will iterate through all entities and check if any of them is modified - and persist changes to the database. Alternatively, you can flush a single entity by using an overloaded version of Flush that receives a single object:

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

In the example above, only changes made to *Customer1* will be persisted. *Customer2* changes will still be in memory only, and you would have to call *Flush* or *Flush(Customer2)* to persist the changes. This gives you finer control over what should be persisted and helps you increase performance of your code.

You must be careful, though, about associated objects. When you call Flush without specifying an object you are safe that all changes in the manager are persisted. You flushing a single object, associated objects might be flushed or not, depending on how the cascade options are set for that Association (or Many-Valued Association). If the association includes the *TCascadeType.Flush*, then it will also be flushed.

Merging/Replicating 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* or *Replicate* methods. These methods behave almost exactly the same, and 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 no 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 the object has an id and no object is found in the database with that id, the behavior depends on the method called (and that is the only difference between Merge and Replicate methods):

- if *Merge* method was called, an exception will be raised;
- if *Replicate* method was called, a new object with the specified id will be saved (inserted).

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

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/Replicate method is a reference to the persistent object in the object manager.

If the transient object passed to Merge/Replicate has no id, then a *Save* operation takes place. Merge/Replicate will create a new internal instance of object, copy all the contents from the passed object to the internal one, and Save (insert) the newly created object. Again, the object returned by Merge/Replicate is different from the one passed. Take a look at the following example:

```
NewCustomer := TCustomer.Create;
NewCustomer.Name := 'John';
MergedCustomer := Manager2.Replicate<TCustomer>(NewCustomer);
// MergedCustomer <> NewCustomer! NewCustomer must be destroyed
```

In the example above, *NewCustomer* doesn't have an id. In this case, Merge/Replicate will create a new customer in database, and return the newly created object. *MergedCustomer* points to a different instance than NewCustomer. MergedCustomer is the persistent one that is tracked by the object manager (and will be destroyed by it when manager is destroyed). NewCustomer continues to be a transient instance and must be manually destroyed.

Note that Merge/Replicate does nothing in the database in update operations - it just updates the persistent object in memory. To effectively update the object in the database you should then call *Flush* method. The only exception is the one described above when the object has no id, or when Replicate saves a new object with existing id. In those cases, a Save (insert) operation is performed immediately in the database.

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 merged/replicated into the manager and the reference will be changed. For example, if *Customer* has a *Country* property pointing to a transient *TCountry* object, The TCountry object will me merged, a new instance will be returned from the merging process, and *Customer.Country* property will be changed to reference the new instanced returned by the merging process.

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 remove. 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.

The object passed to Remove method will eventually be destroyed. If TObjectManager.DeferDestruction property is false (default), the object will be destroyed immediately. If it's true, object will be destroyed when the manager is destroyed.

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 tries to load the object from database. If there is no object (record) in the database with that Id, it returns nil.

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

The associations 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-loaded associations or not.

If you want to retrieve several objects of a class from the database using some criteria (filtering, ordering), just use Find without parameter, it will return a *Criteria* object which you can use to add filters, ordering and later retrieve the results:

```
var
  Customers: TList<TCustomer>;
begin
  Customers := Manager1.Find<TCustomer>.List;
  // Take just the first 10 customers ordered by name
  Customers := Manager1.Find<TCustomer>.Take(10).OrderBy('Name').List;
```

Aurelius is very powerful on querying capabilities. There is a full chapter explaining how to perform Aurelius queries.

Refreshing Objects

You can refresh an object using *Refresh* method from a TObjectManager object. Just pass the object that you want to refresh. The object must be attached to the object manager.

```
Manager1.Refresh(Customer1);
```

Refresh method performs operates almost the same way as Find method. The main difference is that Find method only create new instances that don't exist in the manager and if the instance already exists, it's left untouched. Refresh method, instead, will perform the SELECT statement in the database no matter what, and if the instances already exist in manager, it will update its properties and associations with data retrieved from the database, discarding the existing values in memory, if different.

Note existing transient associations will NOT be destroyed. For example, consider the following code:

```
Customer1 := Manager.Find<TCustomer>(1);
NewCountry := TCountry.Create;
Customer1.Country := NewCountry;
Manager.Refresh(Customer1);
```

In the code above a *TCustomer* instance is loaded from the database, and its *Country* property is updated to point to a transient *TCountry* reference. When Refresh method is called, *Customer1* properties will be reloaded from the database, and thus Country property will point again to the original TCountry instance in the manager (or nil if there is no country associated with he customer). However, the instance referenced by *NewCountry* will not be destroyed. It's up to you to destroy the transient instances unreferenced by Refresh method.

The cascades defined in Association and ManyValuedAssociation attributes in your class are applied here. Any associated object or collection item with Refresh cascade will also have its properties refreshed.

Evicting Objects

In some situations you want to remove (dettach) an object from the TObjectManager, but without deleting it from database (so you can't use *Remove* method) and without destroying the instance.

To do that, you can use *Evict* method. Just pass the object that you want to evict. If the object is not attached to the manager, no operation is performed.

```
Manager1.Evict(Customer1);
```

The cascades defined in Association and ManyValuedAssociation attributes in your class are applied here. Any associated object or collection item with cascade option including *TCascadeType.Evict* will also be evicted (dettached) from the manager.

Note that since the object is not in the manager anymore, you must be sure to destroy it (unless of course you attach it to another manager using for example Update method). Also pay attention to associated objects. If *TCascadeType.Evict* is defined for associated objects, they will also be evicted and must be destroyed as well.

Transaction Usage

You can use transactions when manipulating objects, so that you make sure all operations under the transaction are performed successfully (commit) or anything is reverted (rollback). Usage is simple and is done pretty much the same way you would do when accessing a database in a traditional way.

The transactions are started under the IDBConnection interface context. You can start a transaction using *IDBConnection.BeginTransaction* method, which will return a *IDBTransaction* interface. The IDBTransaction in turn has only two methods: *Commit* and *Rollback*.

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

var
  Transaction: IDBTransaction;
begin
  Transaction := Manager.Connection.BeginTransaction;
  try
    { Perform manager operations }
    Transaction.Commit;
  except
    Transaction.Rollback;
    raise;
  end;
end;
```

Transactions in Aurelius can be nested. This means that if a transaction was already started in IDBConnection but not commited or rolled back yet, creating a new transaction and commiting or rolling it back has no effect. For example:

```
OuterTransaction := Manager.Connection.BeginTransaction;
InnerTransaction := Manager.Connection.BeginTransaction;
InnerTransaction.Commit; // This has NO effect, the same for rollback.
OuterTransaction.Commit; // Commit (or Rollback) is effectively performed here
```

Concurrency Control

When working with multiple users/clients, it might be possible that two or more users try to change the same entity (records in database). TMS Aurelius provides some mechanisms to avoid problems in those situations.

Changed fields

When updating objects, Aurelius detects which property have changed since the entity was loaded from the database in the manager, and it only updates those columns in the database. For example, suppose two users load the same *TCustomer* (with same id) from the database at the same time:

```
// User1
User1Customer := Manager1.Find<TCustomer>(1);
// User2
User2Customer := Manager2.Find<TCustomer>(1);
```

Now first user changes customer's city and update, and second user changes customer's document and update:

```
// User1
User1Customer.City := 'New City';
Manager1.Flush;
// User2
User2Customer.Document := '012345';
Manager2.Flush;
```

Here are the SQL executed by Aurelius for each user (SQL were simplified for better understanding, the actual SQL uses parameters):

```
-- User1:
Update Customer
Set City = 'New City'
Where Id = 1

-- User2:
Update Customer
Set Document = '012345'
Where Id = 1
```

Even if TCustomer class has lots of customer, and some properties might be outdated in memory, it doesn't cause any trouble or data loss here, because only changed data will be commited to the database. In the end, the TCustomer object in database will have both the new city and new document correct.

This is a basic mechanism that solves concurrency problems in many cases. If it's not enough, you can use entity versioning.

Entity Versioning

It might be possible that two users change the exactly same property, in this case, one of the users will "lose" their changes, because it will be overwritten by the other user. Or some other types of operations are performed where all fields are updated (when entity is put in manager without being loaded from database for example, so the manager can't tell which properties were changed).

Or maybe you just need to be sure that the object being updated needs to hold the very latest data. A typical case is where you are updating account balance or inventory records, so you increment/decrement values and need to ensure that no other user changed that data since you loaded.

In this case, you can use entity versioning. To accomplish this, you just need to create an extra integer property in the class, map it (so it's persisted in database) and add the *[Version]* attribute to it:

```
[Entity, Automapping]
TCustomer = class
private
  FId: Integer;
  FName: String;
  {...}
  [Version]
  FVersion: Integer;
  {...}
end;
```

And that's it. Once you do this, Aurelius will make sure that if you update (or delete) an entity, data it holds is the very latest one. If it's not, because for example another user changed the database record in the meanwhile, an exception will be raised and then you can decide what to do (refresh the object for example).

Let's take a look at how it works. First, two users load the same object at the same time:

```
// User1
User1Customer := Manager1.Find<TCustomer>(1);
// User1Customer.Version is 1

// User2
User2Customer := Manager2.Find<TCustomer>(1);
// User2Customer.Version is 1
```

Then User1 updates customer:

```
User1Customer.City := 'New City';
User1Customer.Flush;
// User1Customer.Version becomes 2 (also in database)
```

This is the SQL executed by Aurelius:

```
Update Customer
Set City = 'New City', Version = 2
Where Id = 1 and Version = 1
```

Record is changed successfully because the current version in database is 1, so the actual record is updated.

Now, if User2 tries to update the old customer:

```
// User2Customer.Version is still 1!
User2Customer.City := 'Another city';
User2Customer.Flush;
```

Aurelius tries to execute the same SQL:

```
Update Customer
Set City = 'Another City', Version = 2
Where Id = 1 and Version = 1
```

However this will fail, because the version in the database is not 1 anymore, but 2. Aurelius will detect that no records were affected, and will raise an *EVersionedConcurrencyControl* exception.

Cached Updates

When persisting objects by calling object manager methods like Save, Flush or Remove, the manager will immediately perform SQL statements to reflect such changes in the database. For example, calling this:

```
Manager.Remove(Customer);
```

Will immediately execute the following SQL statement in the database:

```
DELETE FROM CUSTOMERS WHERE ID = :id
```

With cached updates feature, you have the option to defer the execution of all SQL statements to a later time, when you call *ApplyUpdates* method. This is enabled by setting *CachedUpdates* to true. Take the following code as an example:

```
Manager.CachedUpdates := True;
Manager.Save(Customer);
Invoice.Status := isPaid;
Manager.Flush(Invoice);
WrongCity := Manager.Find<TCity>(5);
Manager.Remove(City);
Manager.ApplyUpdates;
```

This code should perform the following SQL statements:

1. INSERT (to save the customer - see "exceptions" below);
2. UPDATE (to modify the status field of the invoice);
3. DELETE (to delete the city).

However, all the statements will be executed one after another, when *ApplyUpdates* is called.

Exceptions

There are exceptions for this rule. When the id value of an entity is generated by the database during an INSERT statement, then the INSERT SQL statement will be executed immediately. So for example, the statement below:

```
Manager.Save(Customer);
```

Might be executed immediately, even when CachedUpdates property is true, if *Customer* is set to be of type IdentityOrSequence, and the underlying database uses the identity mode (the id is generated automatically during the INSERT statement).

Note that this doesn't apply if the database uses SEQUENCE. In other words, even if the id is of type *IdentityOrSequence*, if the id is generated by reading the SEQUENCE value, then the SQL statement used to retrieve the SEQUENCE value will be executed, but the INSERT statement will be deferred until ApplyUpdates is called.

Cached actions

You can read the *CachedCount* property to know how many actions are pending and will be applied in the next call to CachedUpdates.

```
TotalPendingActions := Manager.CachedCount;
```

Batch (Bulk) Updates

Aurelius allows you to modify data in batches. For example, suppose you want to update one hundred customer records, modifying their phone number. With batch updates, a single SQL statement will be executed to modify all records at once. This improves performance significantly in such operations.

Batch updates works together with cached updates. You need to use cached updates mechanism, then just use *TObjectManager.BatchSize* property to tell Aurelius the maximum number of records that can be updated using a single SQL statement.

Consider the following code:

```
// Retrieve some customers from database
CustomerA := Manager.Find<TCustomer>(1);
CustomerB := Manager.Find<TCustomer>(2);
CustomerC := Manager.Find<TCustomer>(3);

// Set batch size and enable cache updates
Manager.BatchSize := 100;
Manager.CachedUpdates := True;

// Modify customer A, B and C
CustomerA.City := 'New York';
Manager.Flush(CustomerA);

CustomerB.City := 'Berlin';
Manager.Flush(CustomerB);

CustomerC.City := 'London';
Manager.Flush(CustomerC);

// Now apply updates
Manager.ApplyUpdates;
```

Aurelius will execute a single SQL statement to modify the three customers. The SQL statement will be something like this:

```
UPDATE Customers SET Name = :p1 WHERE Id = :p2
```

And the parameter values for all records ("New York" and 1, "Berlin" and 2, "London" and 3) will be sent at once together with the SQL statement.

Batch algorithm

Aurelius will perform the batch automatically. You can perform any actions you want. All actions will be cached, and if the actions build SQL statement that are the same, they will be grouped together in a batch, up until the size specified by *BatchSize*.

Thus, the order of actions is important. For example, if you perform:

- *Insert* CustomerA
- *Modify* CustomerA City
- *Insert* CustomerB
- *Modify* CustomerB City

No batches will be created and a SQL statement will be executed for each operation. However, if you perform the actions this way:

- *Insert* CustomerA
- *Insert* CustomerB
- *Modify* CustomerA City
- *Modify* CustomerB City

Then two batches of size 2 will be created. A single SQL statement will be executed to insert the two customers, and another SQL statement will be executed to modify the two customers.

Also note that the batch will be performed if *all SQL statements are the same*. For example, if you modify Customer A city, and then modify Customer B name, no batch will be created. Even though both are update actions, the first will result in a SQL statement that modifies city field, and the other will result in a SQL statement that modifies name field.

Driver-dependent behavior

The way the statement is executed depends on the underlying driver and RDBMS being used. Some drivers support the batch mechanism where all parameters are sent at once. Other drivers do not support it, in this case Aurelius will simulate such batch by executing several SQL statements using a pre-prepared one. Still it will be faster than executing a single statement for every record.

The current drivers that support real batch updates (array DML) are:

- Native Aurelius connectivity;
- FireDAC;
- UniDAC (except when connecting to Firedac, Interbase and NexusDB).

For all other drivers, the batch will be simulated. Note that you don't need to modify anything in your code, regardless of the driver and the database you are using. Aurelius will work transparently in all cases, and use the best mechanism available to perform the batch updates.

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`.

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 TCriteria 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
```

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 := Linq['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(Linq['Name'] = 'Mia Rosenbaum')
    .List;
```

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

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.

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 *Find* method, you will need to call *List<T>* method.

```
var
  Results: TList<TCustomer>;
  MyCriteria: TCriteria;
begin
  MyCriteria := ObjectManager1.Find(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.Find<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.

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 *TObject*:

```
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 *EResultsNotUnique* 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.

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

The Open method returns an *ICriteriaCursor* (or *ICriteriaCursor<T>*) interface which is destroyed automatically by reference counting. The underlying *TCriteria* object (*MyCriteria* variable in the example above) is automatically destroyed when cursor is destroyed. Since ICriteriaCursor<T> implements *GetEnumerator* you can also iterate through the returned entities directly:

```
var
  FetchedCustomer: TCustomer;
begin
  for FetchedCustomer in ObjectManager1.Find<TCustomer>.Open do
  begin
    // Do something with FetchedCustomer
  end;
```

The ICriteriaCursor and ICriteriaCursor<T> interfaces are declared as following.

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

ICriteriaCursor<T: class> = interface(ICriteriaCursor)
  function Get: T;
  function GetEnumerator: TEnumerator<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*.

ResultClass method returns the class of the returned objects. Usually it's the same as *BaseClass*, unless in specific cases like when you are using projections, for example. In this case *ResultClass* will be *TCriteriaResult*.

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.Find<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.

```
uses {...}, Aurelius.Criteria.Projections,
  Aurelius.CriteriaBase, Aurelius.Criteria.Linq;

var
  Results: TObjectList<TCriteriaResult>;
begin
  Results := Manager.Find<TTC_Estimate>
    .CreateAlias('Customer', 'c')
    .Select(TProjections.ProjectionList
      .Add(TProjections.Sum('EstimateNo').As_('EstimateSum'))
      .Add(TProjections.Group('c.Name'))
      )
    .Add(Linq['c.Name'].Like('M%'))
    .OrderBy('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;
  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.

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:

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

Results := Manager1.Find<TCustomer>
  .Where(Linq['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 (*Add* and *Where* methods are equivalents):

```
Results := Manager1.Find<TCustomer>
  .Add(Linq['Country'] = 'US')
  .Add(Linq['Age'] = 30)
  .List;
```

Or you can simply use logical operators directly:

```
Results := Manager1.Find<TCustomer>
  .Where((Linq['Country'] = 'US') and (Linq['Age'] = 30))
  .List;
```

In the topics below you will find all the advanced features for building queries in Aurelius.

Creating Expressions Using Linq

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 *Linq* factory. It's declared in `Aurelius.Criteria.Linq` unit:

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

Linq variable is just a helper object with several methods (*Equal*, *GreaterThan*, etc.) that you can use to easily create TCustomCriterion instances. For example, the following lines produce the same object and will result in the same query:

```
Criterion := TSimpleExpression.Create(TPropertyProjection.Create('Age'), 30, eoGr
eater));
Criterion := Linq.GreaterThan('Age', 30);
Criterion := Linq['Age'] > 30;
```

You can always use the default indexed property passing the property name to start using queries. That will represent a property projection:

```
Linq[<propertyname>]
```

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 Linq to create the following conditions:

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, or the **=** operator, they all do the same.

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

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'] = 'Mia Rosenbaum')
  .List;
```

Another way to write it:

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

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, or the **>** operator, they all 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>
  .Where(
    (Linq['Birthday'] > EncodeDate(1981, 10, 10))
    and (Linq['Birthday'] < EncodeDate(1986, 2, 2))
  )
  .List;
```

Another way to write it:

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

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, or **>=** operator, they all 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>
  .Where(
    (Linq['Birthday'] >= EncodeDate(1981, 10, 10))
    and (Linq['Birthday'] <= EncodeDate(1986, 2, 2))
  )
  .List;
```

Another way to write it:

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

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, or **<** operator, they all 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>
  .Where(
    (Linq['Birthday'] > EncodeDate(1981, 10, 10))
    and (Linq['Birthday'] < EncodeDate(1986, 2, 2))
  )
  .List;
```

Another way to write it:

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

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, or **<=** operator, they all 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>
  .Where(
    (Linq['Birthday'] >= EncodeDate(1981, 10, 10))
    and (Linq['Birthday'] <= EncodeDate(1986, 2, 2))
  )
  .List;
```

Another way to write it:

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

Like

Retrieves a condition where the specified property (or projection) value contains the text specified. It's equivalent 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(
    Linq['Sex'].IsNotNull and Linq['Name'].Like('M%')
  )
  .List;
```

Another write to write it:

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

ILike

Retrieves a condition where the specified property (or projection) value contains the text specified, case insensitive. It's equivalent to the ILIKE 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" (or "m", it's case insensitive).

```
Results := Manager.Find<TCustomer>
  .Where(
    Linq['Sex'].IsNotNull and Linq['Name'].ILike('M%')
  )
  .List;
```

Another write to write it:

```
Results := Manager.Find<TCustomer>
  .Where(
    Linq.IsNotNull('Sex') and Linq.ILike('Name', 'M%')
  )
  .List;
```

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>
  .Where(
    (Linq['Sex'] = tsFemale) or Linq['Sex'].IsNull
  )
  .List;
```

Another way to write it:

```
Results := Manager.Find<TCustomer>
  .Where(
    Linq.Eq('Sex', tsFemale) or Linq.IsNull('Sex')
  )
  .List;
```

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(
    Linq['Name'].Like('M%') and Linq['Sex'].IsNotNull
  )
  .List;
```

Another way to write it:

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

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>
  .Where(Linq.IdEq(1))
  .UniqueResult;
```

Example - Using composite id: return person 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.Find<TPerson>
    .Where(Linq.IdEq(Id))
    .UniqueResult;
```

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 CUSTOMER_NAME is equal to "Mia Rosenbaum".

```
Results := Manager.Find<TCustomer>
  .Where(Linq.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 the 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>
  .Where(Linq.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')
  .Where(Linq.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')
    .Where(Linq.Sql('{Name} = ''United States'''))
  .List<TCustomer>;
```

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>
  .Where(Linq.Sql<TSex>('{Sex} IN (?)', TSex.tsFemale))
  .List;
```

Example - using two parameters of type *TDate*:

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

Starts With

Retrieves a condition where the specified property (or projection) string value must start with the specified value.

Example - Return customers where *Name* property starts with "Mia".

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].StartsWith('Mia'))
  .List;
```

Alternative way to write it:

```
Results := Manager.Find<TCustomer>
  .Where(Linq.StartsWith('Name', 'Mia'))
  .List;
```

Ends With

Retrieves a condition where the specified property (or projection) string value must end with the specified value.

Example - Return customers where *Name* property ends with "Junior".

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].EndsWith('Junior'))
  .List;
```

Alternative way to write it:

```
Results := Manager.Find<TCustomer>
  .Where(Linq.EndsWith('Name', 'Junior'))
  .List;
```

Contains

Retrieves a condition where the specified property (or projection) string value must contain the specified value.

Example - Return customers where *Name* property contains "Walker".

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].Contains('Walker'))
  .List;
```

Alternative way to write it:

```
Results := Manager.Find<TCustomer>
  .Where(Linq.Contains('Name', 'Walker'))
  .List;
```

In

The actual method name is "*_In*". Checks if the value of a specified property (or projection) belongs to a set of predefined values. The predefined set of values can be of type string, integer or enumerated.

Example - Return invoices where *Status* property is either *Approved* or *Rejected*, and year of issue date is *2016* or *2014*.

```
Results := Manager.Find<TInvoice>
  .Add(Linq['Status']._In([TInvoiceStatus.Approved, TInvoiceStatus.Rejected]))
  .Add(Linq['IssueDate'].Year._In([2016, 2014])
  .List;
```

Alternative way to write it:

```
Results := Manager.Find<TInvoice>
  .Add(Linq._In('Status', [TInvoiceStatus.Approved, TInvoiceStatus.Rejected]))
  .Add(Linq._In('IssueDate', [2016, 2014]);
  .List;
```

Comparing Projections

In most of the examples of filtering in queries, we used just the name of the property and compare it to a value. For example:

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'] = 'Mia')
.List;
```

But Aurelius query is much powerful than that. *Linq['Name']* actually represents a projection, and you can use any projection in any expression you want.

This gives you great flexibility since you can create many different types of projections and compare them. For example, you can compare two projections, as follows.

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

```
Results := Manager.Find<TOrder>
  .Where(Linq['CancelationDate'] > Linq['ShippingDate'])
  .List;
```

Or you can even use complex expressions. We can for example change the above query to bring all orders where the year of cancelation date is the same as the year of shipping date:

```
Results := Manager.Find<TOrder>
  .Where(Linq['CancelationDate'].Year = Linq['ShippingDate'].Year)
  .List;
```

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 three options: use subcriteria, aliases or rely on auto alias mechanism.

Auto alias

You can search associated objects by simply referencing their subproperties from the association property. Support you have a `TEstimate` class which has a `Customer` property of type `TCustomer`. The customer entity, in turn, has a `Name` property. You can query for all invoices which customer name starts with `M` this way:

```
Results := Manager.Find<TEstimate>
  .Where(Linq['Customer.Name'].StartsWith('M'))
  .List;
```

Aurelius will automatically create a subcriteria with alias `Customer`. Nested associations are possible by adding more of them separated by dots. To query for invoices where country of customer is United States:

```
Results := Manager.Find<TEstimate>
  .Where(Linq['Customer.Country.Name'] = 'United States')
  .List;
```

This feature was introduced in version 5.6 and is turned on by default.

Using aliases

Instead of using auto alias, you can explicitly create an alias for an association to filter by sub properties of such association.

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Where(Linq['c.Name'].Like('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 *List<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.Find<TEstimate>
  .CreateAlias('Customer', 'ct')
  .CreateAlias('ct.Country', 'cn')
  .Where(Linq['cn.Name'] = 'United States')
  .List;
```

Using SubCriteria

You can alternatively 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')
    .Where(Linq['Name'].Like('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 object (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.Find<TEstimate>
  .SubCriteria('Customer')
    .SubCriteria('Country')
      .Where(Linq['Name'] = 'United States')
      .List<TEstimate>;
```

Mixing SubCriteria and aliases

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

```
Results := Manager.Find<TEstimate>
  .SubCriteria('Customer')
    .CreateAlias('Country', 'cn')
    .Where(Linq['cn.Name'] = 'United States')
  .List<TEstimate>;
```

Specifying Eager fetching for associations loaded as lazy by default

Your class mapping might have defined associations to be marked as lazy-loaded (using proxies). This means if you retrieve one hundred records and you want to access the associated object, one hundred SQL statements will be executed to retrieve such value. You can optionally override the default loading mechanism and set the association to be eager-loaded. This way Aurelius will build an extra JOIN in the SQL statement to retrieve the associated objects in a single SQL.

You can that by using `FetchEager` method passing the name of the association to be loaded eagerly:

```
  Results := Manager.Find<TEstimate>
    .FetchEager('Customer')
    .List;
```

You can also use dots (`.`) to provide subproperties to be loaded eagerly:

```
  Results := Manager.Find<TEstimate>
    .FetchEager('Customer.Country')
    .List;
```

Alternatively, you can also pass `TFetchMode.Eager` as the third parameter of `CreateAlias` or second parameter of `SubCriteria` method:

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'ct', TFetchMode.Eager)
  .List;
```

With either of the queries above, even if *TEstimate.Customer* association is set as lazy-loading, Aurelius will create a single SQL with a JOIN between estimates and customers and retrieve all customers at once. This gives you an extra degree of flexibility when it comes to optimize your application.

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 either *AddOrder* or *OrderBy* methods 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.

Example - Retrieve customers ordered by *Name*.

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].Like('M%'))
  .OrderBy('Name')
  .List;
```

Same query using *AddOrder* (instead of *OrderBy*):

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].Like('M%'))
  .AddOrder(TOrder.Asc('Name'))
  .List;
```

You can also use association aliases in orderings.

Example - Retrieve all estimates which *IssueDate* is not null, ordered by customer name in descending order (second parameter in *OrderBy* specify ascending/descending - false means descending, it's true by default).

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Where(Linq['IssueDate'].IsNotNull)
  .OrderBy('c.Name', false)
  .List;
```

Same query using *AddOrder*:

```
Results := Manager.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Add(Linq['IssueDate'].IsNotNull)
  .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.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo').As_('EstimateSum'))
    .Add(TProjections.Group('c.Name'))
    )
  .Where(Linq['c.Name'].Like('M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

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, a function that retrieves the year of a date, 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.

Usually you will use projections to return specific/calculated values instead of objects, or to perform complex condition expressions (to retrieve all customers where the year of birthday column is equal to 1999).

For example, the following query retrieves the number of invoices for the year 2013 and illustrates how to use projections in both *select* and *where* parts of the query.

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

TotalInvoicesFor2013 := Manager.Find<TInvoice>
  .Select(TProjections.Count('Id'))
  .Where(Linq['IssueDate'].Year = 2013)
  .UniqueValue;
```

The following topics explain in details what projections are and how you can use them.

Projections Overview

Any projection object descends from *TProjection* class. To make a query return projections (calculated values) instead of entities, use the *SetProjections* or *Select* method.

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

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

Value := Manager.Find<TEstimate>
  .Select(TProjections.Sum('EstimateNo'))
  .CreateAlias('Customer', 'c')
  .Where(Linq['c.Name'].Like('M%'))
  .UniqueValue;
```

You can only have a single projection specified for the select part of 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.Find<TEstimate>
  .CreateAlias('Customer', 'c')
  .Select(TProjections.ProjectionList
    .Add(TProjections.Sum('EstimateNo'))
    .Add(TProjections.Group('c.Name'))
    )
  .ListValues;
```

Note that when using projections, the query can instances of the queried class or list of *TCriteriaResult* objects, which you can use to retrieve the projection values. The result depends if you use `ListValues` / `UniqueValue` or `List` / `UniqueResult` methods to retreive the results.

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.

You can also use projections in the *where* clause to add complex queries. Many of the condition expressions you can use in a query can compare projections, for example:

```
YoungCustomers := Manager.Find<TCustomer>
  .Where(Linq['Birthday'].Year > 2000)
  .List;
```

will list all customers which year of birth is greather than 2000.

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('Tota
l'));
Projection := TProjections.Sum('Total');
```

You can use TProjections to create the following projections:

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

Example - Calculates the sum of all estimates where the customer name begins with "M".

```
Value := Manager.Find<TEstimate>
  .Select(Linq['EstimateNo'].Sum)
  .CreateAlias('Customer', 'c')
  .Where(Linq['c.Name'].Like('M%'))
  .UniqueValue;
```

Alternative way to write the same query:

```
Value := Manager.Find<TEstimate>
  .Select(TProjections.Sum('EstimateNo'))
  .CreateAlias('Customer', 'c')
  .Where(Linq['c.Name'].Like('M%'))
  .UniqueValue;
```

Prop

Creates a projection that represents the value of a property. In most cases, you will use that projection transparently, because the following constructions will return such projection for you:

```
Linq['Name']
Linq['IssueDate']
```

Alternatively there are overloads for almost all methods in *Linq* and *TProjection* classes that accept 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.

The following two queries are equivalent, both retrieve the name of the customers ordered by the *Name*:

```
Results := Manager.Find<TCustomer>
  .Select(Linq['Name'])
  .AddOrder(TOrder.Asc(Linq['Name']))
  .ListValues;

{...}

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

The following three queries are also equivalent:

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

{...}

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

{...}

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

Limiting the selected properties

The property project is the only projection you can use and still retrieve entity objects instead of `TCriteriaResult`. If your query has only property projections, then you can still use `List` (or `UniqueResult`) to retrieve entities:

```
Customers := Manager.Find<TCustomer>
  .Select(TProjections.ProjectionList
      .Add(Linq['Name'])
      .Add(Linq['Birthday'])
    )
  .List;
```

In the above example, the query will return a list of `TCustomer` objects. But they underlying SQL statement will be optimized to only include columns `Name` and `Birthday`, and the returned `TCustomer` object will only have such properties set.

WARNING
Be careful when using entities which properties are only partial set. If you use such objects to do a full update, the unreturned empty properties will be set back to the database and respective database columns will be erased.

You can also use aliased projections to specify subproperties to return:

```
Estimates := Manager.Find<TEstimate>
  .Select(TProjections.ProjectionList
    .Add(Linq['EstimateNo'])
    .Add(Linq['Customer'])
    .Add(Linq['Customer.Name'])
    .Add(Linq['Customer.Sex'])
    .Add(Linq['Customer.Country'])
    .Add(Linq['Customer.Country.Id'])
  )
  .FetchEager('Customer')
  .FetchEager('Customer.Country')
  .List
```

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;
```

Add

Adds two numeric values.

Example:

```
Results := Manager.Find<TInvoice>
  .Select(Linq['Total'] + Linq['Additional'])
  .List;
```

Another way to write it:

```
Results := Manager.Find<TInvoice>
  .Select(Linq.Add(Linq['Total'], Linq['Additional']))
  .List;
```

Subtract

Subtracts two numeric values.

Example:

```
Results := Manager.Find<TInvoice>
  .Select(Linq['Total'] - Linq['Discount'])
  .List;
```

Another way to write it:

```
Results := Manager.Find<TInvoice>
  .Select(Linq.Subtract(Linq['Total'], Linq['Discount']))
  .List;
```

Multiply

Multiplies two numeric values.

Example:

```
Results := Manager.Find<TInvoiceItem>
  .Select((Linq['Quantity'] * Linq['UnitaryValue']).As_('TotalValue'))
  .List;
```

Another way to write it:

```
Results := Manager.Find<TInvoiceItem>
  .Select(Linq.Multiply(Linq['Quantity'], Linq['UnitaryValue']).As_('TotalValue')
)
  .List;
```

Divide

Divides two numeric values.

Example:

```
Results := Manager.Find<TInvoiceItem>
  .Select((Linq['Total'] / Linq['Quantity']).As_('ItemValue'))
  .List;
```

Another way to write it:

```
Results := Manager.Find<TInvoiceItem>
  .Select(Linq.Multiply(Linq['Total'], Linq['Quantity']).As_('ItemValue'))
  .List;
```

Aurelius ensures consistency among different databases. When performing division between two integer values, many databases truncate the result and return an integer, rounded value. For example, 7 / 5 results 1. Some databases do not behave that way.

In Aurelius, the division operator performs with Pascal behavior: the result is a floating point operation, even when dividing two integer values. Thus, 7 / 5 will return 1.4, as expected.

Condition

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

Example - 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>
  .Select(TProjections.ProjectionList
    .Add(Linq['Name'])
    .Add(TProjections.Condition(
      Linq['Sex'].IsNull,
      Linq.Literal<string>('Null'),
      TProjections.Condition(
        Linq['Sex'] = tsMale,
        Linq.Literal<string>('Male'),
        Linq.Literal<string>('Female')
        )
      )
    )
  )
  .ListValues;
```

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.

Example - Retrieves some literal values.

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

Another example using Condition projection:

```
Results := Manager.Find<TCustomer>
  .Select(TProjections.ProjectionList
    .Add(Linq['Name'])
    .Add(TProjections.Condition(
      Linq['Sex'].IsNull,
      Linq.Literal<string>('Null'),
      TProjections.Condition(
        Linq['Sex'] = tsMale,
        Linq.Literal<string>('Male'),
        Linq.Literal<string>('Female')
        )
      )
    )
  )
  .ListValues;
```

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.

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.

Example - Creates a projection list with two projections: 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;
```

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.

Example - 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(Linq['EstimateNo'].Sum.As_('EstimateSum'))
    .Add(Linq['c.Name'].Group)
    )
  .Add(Linq['c.Name'].Like('M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

Alternatively you can create aliased projections using the *TProjections.Alias* 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.Alias(TProjections.Sum('EstimateNo'), 'EstimateSum'))
    .Add(TProjections.Group('c.Name'))
    )
  .Add(Linq.Like('c.Name', 'M%'))
  .AddOrder(TOrder.Asc('EstimateSum'))
  .ListValues;
```

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.Find<TCustomer>
  .CreateAlias('Country', 'c')
  .Select(TProjections.ProjectionList
    .Add(Linq['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 placeholders 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.

Year

Retrieves the year of a specified date/time value.

Example:

```
.Where(Linq['IssueDate'].Year = 2013)
```

Year method creates a projection that extracts the year of a date value. Equivalent code:

```
.Where(Linq.Eq(TProjections.Year('IssueDate'), 2013))
```

Month

Retrieves the month of a specified date/time value.

Example:

```
.Where(Linq['IssueDate'].Month = 11)
```

Month method creates a projection that extracts the month of a projection with a date value. Equivalent code:

```
.Where(Linq.Eq(TProjections.Month('IssueDate'), 11))
```

Day

Retrieves the day of a specified date.

Example:

```
.Where(Linq['IssueDate'].Day = 31)
```

Day method creates a projection that extracts the day of a projection with a date value. Equivalent code:

```
.Where(Linq.Eq(TProjections.Day('IssueDate'), 31))
```

Hour

Retrieves the hour of a specified date/time value.

Example:

```
.Where(Linq['AppointmentTime'].Hour > 12)
```

Hour method creates a projection that extracts the hour of a projection with a date/time value. Equivalent code:

```
.Where(Linq.Gt(TProjections.Hour('AppointmentTime'), 12))
```

Minute

Retrieves the minute of a specified date/time value.

Example:

```
.Where(Linq['AppointmentTime'].Minute > 45)
```

Minute method creates a projection that extracts the number of minutes of a projection with a date/time value. Equivalent code:

```
.Where(Linq.Gt(TProjections.Minute('AppointmentTime'), 45))
```

Second

Retrieves the second of a specified date/time value.

Example:

```
.Where(Linq['AppointmentTime'].Second > 45)
```

Second method creates a projection that extracts the number of seconds of a projection with a date/time value. Equivalent code:

```
.Where(Linq.Gt(TProjections.Second('AppointmentTime'), 45))
```

Upper

Converts a string value to upper case.

Example:

```
.Where(Linq['Name'].Upper = 'JACK')
```

Equivalent code:

```
.Where(Linq.Eq(TProjections.Upper('Name'), 'JACK'))
```

Lower

Converts a string value to lower case.

Example:

```
.Where(Linq['Name'].Lower = 'jack')
```

Equivalent code:

```
.Where(Linq.Eq(TProjections.Lower('Name'), 'jack'))
```

Concat

Concatenates two strings.

Example:

```
.Select(Linq['FirstName'].Concat(' ').Concat(Linq['LastName']))
```

Equivalent code:

```
.Select(Linq.Concat(Linq.Concat(Linq['FirstName'], ' - '), Linq['LastName']))
```

Aurelius does not ensure cross-database consistent when it comes to null handling. Oracle treats null as empty strings, so if your expression is concatenating a null value, result will be null in all databases except Oracle, where it will concatenate the two strings normally (considering null as empty string).

Length

Returns the number of characters in a string.

Example:

```
// Return entities which name has less than 10 characters
.Where(Linq['Name'].Length < 10)
```

Equivalent code:

```
// Return entities which name has less than 10 characters
.Where(Linq.LessThan(TProjections.Length('Name'), 10))
```

ByteLength

Returns the number of bytes in a binary property.

Example:

```
// Return entities which Photo has less than 65536 bytes
.Where(Linq['Photo'].ByteLength < 65536)
```

Equivalent code:

```
// Return entities which Photo has less than 65536 bytes
.Where(Linq.LessThan(TProjections.ByteLength('Photo'), 65536))
```

Substring

Returns a substring of the specified string.

Example:

```
// Return the first 5 characters of the name
.Select(Linq['Name'].Substring(1, 5))
```

First parameter is the start index of substring, 1-based. Thus, 1 represents the first character of the string, 2 the second, etc. Second parameter is the length of substring to be returned.

Equivalent code which passes the projection/property name as the first parameter:

```
// Return the first 5 characters of the name
.Select(TProjections.Substring('Name', 1, 5))
```

Position

Returns the index value of the first character in a specified substring that occurs in a given string.

Example:

```
// Return entities only if the position of "@" character
// in the EMailAddress property is higher than 5
.Where(Linq['EmailAddress'].Position('@') > 5)
```

The parameter is the substring to be searched for. The result is the index of the first occurrence of the string, 1-based. In other words, if the substring occurs in the first character, the result is 1. If the substring is not found, result is 0.

Equivalent code which passes the projection/property name as the first parameter:

```
// Return entities only if the position of "@" character
// in the EMailAddress property is higher than 5
.Where(Linq.GreaterThan(TProjections.Position('@', 'EmailAddress'), 5)))
```

SqlFunction

Calls a custom SQL function. Aurelius provides many cross-database projection functions like Year, Upper, Concat, etc. But in case you want to call an specific database function, or create your own, you can use *SqlFunction* to call it.

For example, if you want to use PostgreSQL's *Unaccent* function:

```
.Where(Linq.ILike(
  Linq.SqlFunction('unaccent', nil, Linq['Name']),
  Linq.SqlFunction('unaccent', nil, Linq.Value<string>(SomeValue))
 ))
```

First parameter is the name of the function.

Second parameter is the value type (*PTypeInfo*) returned by the function. If the type of function result is the same of the type of the parameter, you can simply pass *nil*. In this example, *Name* is a string field, and unaccent also returns a string value, so you can just use nil.

If the function is not registered by default in Aurelius system (which is the case for *Unaccent* function), Aurelius will raise an error when trying to execute the query, informing that function could not be found. You need to register the function in the specific Dialect using *RegisterFunction*:

```
uses
  {...}, Aurelius.Sql.Interfaces, Aurelius.Sql.Register, Aurelius.Sql.Functions;

TSQLGeneratorRegister.GetInstance.GetGenerator('POSTGRESQL')
  .RegisterFunction('unaccent', TSimpleSQLFunction.Create('unaccent'));
```

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(Linq['Name'].Like('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.

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>
  .OrderBy('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>
  .OrderBy('Name')
  .Skip(10)
  .List;
```

The previous code will retrieve customers ordered by name, but 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>
  .OrderBy('Name')
  .Skip(PageIdx * PageSize)
  .Take(PageSize)
  .List;
```

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(Linq['i.Price'] = 20)
  .OrderBy('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(Linq['i.Price'] = 20)
  .OrderBy('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.

Cloning a Criteria

Aurelius *TCriteria* object also has a *Clone* method you can use to clone the criteria. This might useful when you want to reuse the criteria multiple times and maybe slightly change from the base criteria:

```
MyCriteria := Manager.Find<TCustomer>
  .Where(Linq['Name'] = 'Mia');

ClonedCriteria := MyCriteria.Clone;
ClonedCriteria.OrderBy('Id');
MyResults := MyCriteria.List<TCustomer>;
ClonedResults := ClonedCriteria.List<TCustomer>;
```

Refreshing Results

When performing a query, Aurelius will keep exisiting entities in the cache. For example, if your query returns two *TCustomer* objects with ID's 10 and 15, if there are already instances of those objects in the manager, they will be kept in the cache with existing properties and will not be updated.

Alternatively, you can use *Refreshing* method when building the criteria to tell Aurelius that you want existing objects to be objects with current database values.

The query below will bring all *TCustomer* objects which year of birthday is 1999. If any of those customers are already in the manager, their properties will still be updated with values retrieved from the database:

```
MyCriteria := Manager.Find<TCustomer>
  .Where(Linq['Birthday'].Year = 1999)
  .Refreshing
  .List;
```

Note that when refreshing an object that has lazy-loaded associations, the proxy is updated and not immediately loaded. When the associated object (or list) is then read, Aurelius will try to load the objects and if they are in the cache, they will not be updated. This means if you have lazy-loaded association, specially lists, and you want the list objects to be refreshed themselves, you should iterate through the list and call *Refresh* for each item manually.

Dictionary

Aurelius dictionary is intended to be used in Aurelius queries. The idea is that instead of using strings to reference entity properties, you use the dictionary directly. For example, a usual Aurelius query is like this:

```
Results := Manager.Find<TCustomer>
  .Where(Linq['Name'].Like('M%'))
  .List;
```

But the above approach is error prone, the `'Name'` string can be wrong, it can be wrongly typed, or the name might be modified. You will only find the errors at runtime. With the dictionary, you can write the query like this:

```
Results := Manager.Find<TCustomer>
  .Where(Dic.Customer.Name.Like('M%'))
  .List;
```

With the code above you have full code completion when you are coding, so you don't have to remember the property names, and you get errors at compile-time.

WARNING
Aurelius dictionary should only be used from Delphi XE6 and later versions. Older Delphi versions have RTTI issues that cause the dictionary to not work correctly in some situations, especially the dictionary validator.

Dictionary generation

To be used, the dictionary has to be generated from the existing mapped classes or the database. "Generate" means creating a Delphi unit `.pas` file with source code containing the dictionary classes. You can then add such unit to the `uses` clause and use the dictionary in queries. There are three ways you can generate the dictionary.

Generate from application

The dictionary can be generated directly from your entity classes. It has to be generated from an existing Delphi application so that Aurelius retrieve information via RTTI and generate the source code unit for you. This is very convenient if you use a code-first approach, when you create your entity classes and ask Aurelius to create the database for you.

To generate the dictionary, you should use the `TDictionaryGenerator` class declared in unit `Aurelius.Dictionary.Generator`. It can be as simple as this. Use the unit:

```
uses Aurelius.Dictionary.Generator;
```

And then use this line of code anywhere in your application:

```
TDictionaryGenerator.GenerateFile('C:\SomeFolder\MyDictionary.pas');
```

The above line is enough to generate the unit in the specified file location. `TDictionaryGenerator` provides several options you can configure, in case you want to have more control over the generated dictionary. Here is a more advanced use of it:

```
var
  Generator: TDictionaryGenerator;
  SourceCode: string;
begin
  // Create the dictionary for the entity classes defined in a specific
  // mapping explorer. In this case, use model "Accounting"
  Generator := TDictionaryGenerator.Create(TMappingExplorer.Get('Accounting'));
  try
    // The DictionaryId is used to build the name of dictionary interface and
class.
    // By default the model name is used, so the default interface and class
names would be
    // IAccountingDictionary and TAccountingDictionary.
    // The change below will make them IMyAccountingDictionary and
TMyAccountingDictionary
    Generator.DictionaryId := 'MyAccounting';

    // The name of the global variable holding the dictionary. By default, the
name will be Dic.
    // With the change below, an eventual dictionary entity
    // will be accessed as D.Customer instead of Dic.Customer
    Generator.GlobalVarName := 'D';

    // The name of the unit to be generated. When you call
TDictionaryGenerator.GenerateFile
    // this is automatically set based on the file name. In this case we have to
set it directly.
    // If you do not set this, the default unit name will be Unit1.
    Generator.OutputUnitName := 'AccountingDictionary';

    // Retrieve the full source code as a string
    SourceCode := Generator.GenerateSource;

    // Save the source to a file
    TFile.WriteAllText('C:\SomeFolder\AccountingDictionary.pas', SourceCode);
  finally
    Generator.Free;
  end;
end;
```

Generate from database

In a database-first approach, you create your database tables first, and then generate Aurelius classes from such database. You can do that using the TMS Data Modeler tool or using the TAureliusConnection "generate entities" wizard from the IDE.

In both cases, you have an option to also generate the dictionary, in addition to the Aurelius classes. This is straightforward and the dictionary will be automatically created for you.

Generate from command-line tool

You can also generate the dictionary from a command-line tool called AureliusDictionaryGenerator. This can be useful if you want to automate the dictionary generation and for some reason you don't want to do it from your own application.

With the generator tool you need to have a package file (.bpl) compiled with your entities, and then pass the package file to the it. It will extract the classes from the package and generate the source code file. The source code of the AureliusDictionaryGenerator tool is available in Aurelius distribution under the `demos` folder.

When you launch the tool without passing parameters, you get the help page:

```
PS> .\AureliusDictionaryGenerator.exe
TMS Aurelius Dictionary Generator version 0.1
Copyright (c) TMS Software. All rights reserved.

OutputFileName - Required parameter was not provided.

AureliusDictionaryGenerator.exe <OutputFileName> [options]

<OutputFileName>               - Output file name to be generated
-pvalue, /package:value        - The bpl package file to extract the model from
-ivalue, /id:value             - Dictionary id
[-gvalue], [/globalvar:value] - Global variable name, default: Dic
[-mvalue], [/model:value]      - Name of the model to generate the dictionary
                                 for, default: Default
```

Parameters `-g` and `-m` are optional and if omitted the default values are used. The other parameters are required and their equivalent are described in section Generate from application.

Here is one example that reads Aurelius entity classes from package `C:\MyProject\Bpl\Entities.bpl` and generates the dictionary `Default` in file `C:\MyProject\MyDictionary.pas`:

```
AureliusDictionaryGenerator.exe -i:Default -p:"C:\MyProject\Bpl\Entities" "C:
\MyProject\MyDictionary.pas"
```

Using the dictionary

Using Aurelius dictionary is very simple. It's intended to be used in Aurelius queries and all you have to do is to use the dictionary properties and query from them. The dictionary properties holds a tree structure representing all the entity classes you have in your model, and the properties and associations for each entity class.

To use the dictionary just add the dictionary unit name (the file you generated) to your unit uses clause, and it's available via the `Dic` global variable (unless you explicitly changed the variable name when generating the dictionary).

Simple properties as projections

Just reference the entity and the property and compare it to a value. The property in dictionary is a query projection, which means you can also use all methods available in projections like Sum, Contains and many others.

You can do simple comparisons like these:

```
Manager.Find<TCustomer>
  .Where(Dic.Customer.CountryName = 'Germany')
  .Where(Dic.Customer.Name.Contains('Herwig'))
```

And:

```
Manager.Find<TInvoice>
  .Where(
    (Dic.Invoice.IssueDate >= EncodeDate(2020, 10, 10))
    and (Dic.Invoice.IssueDate < EncodeDate(2021, 2, 2)))
  .OrderBy(Dic.Invoice.Code)
```

Associations

The dictionary also makes it very easy to query by associated objects. All you need do is just use the associated properties into a deeper level and do the comparison the same way. For example, the following query calculates the sum of totals for invoices which country is Germany. It uses the customer associated with the invoice, and then the country associated with the customer:

```
Manager.Find<TInvoice>
  .Select(Dic.Invoice.Total.Sum)
  .Where(Dic.Invoice.Customer.Country.Name = 'Brazil')
```

To make it easier to read and write the queries, you can also put some associations in specific variables like this:

```
var
  Est: IEstimateDictionary;
begin
  Est := Dic.Estimate;
  Manager.Find<TEstimate>
    .Select(TProjections.ProjectionList
      .Add(Est.EstimateNo.Sum)
      .Add(Est.Customer.Name.Group))
    .Where(Est.IssueDate.IsNotNull)
    .Where(
      (Est.EstimateNo.Sum >= 6)
      and (Est.EstimateNo.Sum <= 14))
    .OrderBy(Est.Customer.Name)
```

Dictionary validation

Since dictionary must be explicitly generated from existing classes, it's possible that it might get outdated. You might add new entity classes, or even add, rename or remove properties in existing entity classes. This might lead, again, to runtime errors. For example, you might have written the expresion `` `Dic.Customer.Name = 'John' `` but later you renamed the `Name` property to `CompanyName` property. You will get an error at runtime indicating that `Name` property does not exist.

To avoid such problems Aurelius provides the dictionary validator. It's just a simple check you add at the beginning of your application (or at any point you want, but obviously before you execute any code that uses the dictionary). The validator will perform a full check of the dictionary and make sure its contents matches the real entity classes.

Using the dictionary is very simple and can be accomplished with a single line of code. You need to first add the `Aurelius.Dictionary.Validator` unit to your uses clause:

```
uses Aurelius.Dictionary.Validator;
```

And then use `Check` method of `TDictionaryValidator` class. The method can optionally receive a mapping explorer, this way you can explicitly tell Aurelius the model which the dictionary should be validated for.

```
  // Check if the dictionary is valid for the default model
  TDictionaryValidator.Check(Dic);

  // Check if another dictionary in a different unit is valid
  // for model Accounting
  TDictionaryValidator.Check(AccountingDictionary.Dic, TMappingExplorer.Get('Acco
unting'));
```

The above code will raise an `EDictionaryValidationException` exception with detailed information about the problems (the exception class has an `Errors` property you can inspect).

Alternatively, if you prefer a silent validation, you can create a `TDictionaryValidator` instance yourself, call the `Validate` method (which returns a boolean value indicating if the validation was succesful) and then inspect the `Errors` property yourself:

```
Validator := TDictionaryValidator.Create(TMappingExplorer.Default);
if not Validator.Validate(Dic) then
begin
  for ErrorMessage in Validator.Errors do
    LogSomewhere(ErrorMessage);
```

Data Validation

Aurelius provides easy and straightforward ways to validate your entities before they are persisted. By adding specific attributes to your entity, you can easily ensure that the entity is always saved to the database in a valid state.

In the following example, our mapping specifies that the `FName` field of the `TCustomer` class is required and its length must not exceed 20 characters:

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

type
  [Entity, Automapping]
  TCustomer = class
  strict private
    FId: Integer;

    [Required, MaxLength(20)]
    FName: string;
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
  end;
```

If we then try to save an customer with a name longer than 20 characters:

```
  var Customer := TCustomer.Create;
  Manager.AddOwnership(Customer);
  Customer.Name := 'Name too long for customer';
  Manager.Save(Customer);
```

Aurelius will refuse to save it, raising an `EEntityValidationException` exception:

```
EEntityValidationException: Validation failed for entity of type
"Entities.Customer.TCustomer": Field FName must have no more than 20 character(s)
```

NOTE

The validations occur at application level. They are **not** related to database-level checks. For example, to set the column length at database level, and more properties like nullability, you still should use mapping attributes, like the Column attribute.

Built-in Validators

Aurelius provides several built-in validators that you can use by applying attributes to your mapped class members:

Required

Ensures that the class member has a valid value.

```
[Required]
FRate: Integer;

[Required]
FBirthday: Nullable<TDateTime>;
```

Validation of `FRate` will never fail because even a `0` (zero) value is considered a valid value. `FBirthday` will fail if it's null only.

> **NOTE**
> The exception is the string value. An empty string is not considered a value and will fail the `Required` validation

MaxLength

Specifies the maximum length of a string value.

```
[MaxLength(30)]
FName: string;
```

MinLength

Specifies the minimum length of a string value.

```
[MinLength(5)]
FName: string;
```

Range

Specifies the range of valid values for numeric values.

```
[Range(1, 10)]
FRate: Integer;
```

In the previous example, `FRate` value must be between 1 and 10.

EmailAddress

Ensures the string values contains a valid e-mail address.

```
[EmailAddress]
FEmail: string;
```

RegularExpression

Ensures the string value matches the provided regular expression.

```
[RegularExpression('^[0-9]{5}$')]
FZipCode: string;
```

Entity validators

In addition to class members, you can also apply validators at the entity-level. You should use it the same way: add a validation attribute to the entity class. Of course, the validation attribute must make sense for the whole entity. None of the current built-in validators can be applied to the entity, since they check class member values, but you can create custom validators and apply them to the class.

But the more straightforward way to add entity validators is using the `OnValidate` attribute. Just add the attribute to a method you want to be executed when the entity should be validated:

```
  {$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}
  TCustomer = class
    [OnValidate]
    function CheckBirthday: IValidationResult;
    [OnValidate]
    function CheckName(Context: IValidationContext): IValidationResult;
```

The method must always return an interface of type `IValidationResult`. The method can receive a single argument of type `IValidationContext`, or no parameter at all.

WARNING
By default, Delphi doesn't generate RTTI for non-published methods. That's why you must add the directive `{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}` to your class. If you don't do that, Aurelius won't know about the mentioned methods and they will not be invoked when the validation is performed!

This is a example of implementation:

```
function TCustomer.CheckBirthday: IValidationResult;
begin
  Result := TValidationResult.Create;
  if Birthday.IsNull then Exit;

  if YearOf(Birthday) < 1899 then
    Result.Errors.Add(TValidationError.Create('A person born in XIX century is 
not accepted'));
  if (MonthOf(Birthday) = 8) and (DayOf(Birthday) = 13) then
    Result.Errors.Add(TValidationError.Create('A person born on August, 13th is 
not accepted'));
end;

function TCustomer.CheckName(Context: IValidationContext): IValidationResult;
begin
  Result := TValidationResult.Create;
  if Name = 'invalid name' then
    Result.Errors.Add(TValidationError.Create('Invalid name'));
end;
```

You can have multiple methods marked with the `OnValidate` attribute. All methods will be executed, in the order they are declared in the class.

NOTE
The class member validators are executed first. If and only if there are no errors in class member validations, the entity validators are executed.

Validation Messages

Each built-in validator has its own error message predefined. If the validation fails, the predefined error message will be displayed. For example, the following validation:

```
[EmailAddress]
FEmail: string;
```

If failed, it will generate the error message:

```
Field FEmail is not a valid e-mail address
```

You can customize such messages in two ways.

DisplayName

This is not a validator, but it's an attribute you can add to any class member to modify its name when it's used in validation error messages.

```
[EmailAddress, DisplayName('e-mail')]
FEmail: string;
```

When used with the `DisplayName` attribute as above, the `EmailAddress` validator will now generate the following error message:

```
Field e-mail is not a valid e-mail address
```

One advantage of using `DisplayName` attribute is that the modified name will be used in all existing validations set for the class member.

Custom error message

If changing `DisplayName` is not enough, you can simply provide a full custom error message. Every built-in validator attribute can receive an additional parameter with the custom error message to be used:

```
[DisplayName('e-mail')]
[EmailAddress('You must provide a valid e-mail address for field "%0:s"')]
FEmail: string;
```

In case of a failed validation, the following error message will be generated:

```
You must provide a valid e-mail address for field "e-mail"
```

Note how the custom error message can be combined with the `DisplayName` attribute. The `%0:s` parameter is valid for all built-in validators and contain the member name. Other validators can have additional parameters, for example the `Range` validator provides the minimum and maximum allowed values in parameters `%1:s` and `%2:s` respectively.

Handling Failed Validation

If one or more validation fails, an exception of type `EEntityValidationException` is raised and the persistence operation will not complete.

In most cases, you won't have to do anything special to handle it - the exception will propagate as any other Delphi exception: if you are running a desktop application, the default exception handler will show a message to the user. If you are running an application server, you should be already catching and logging the exceptions, so it won't be different in this case.

The `EEntityValidationException` has some specific properties you can inspect in a `try..except..on` block to gather more details about the validation. The `Entity` property contains the entity instance which failed to validate. The `Results` property contains a list of `IManagerValidationResult` interfaces with specific information for each validation.

Consider the following mapping and validations:

```
type
  [Entity, Automapping]
  TCustomer = class
  strict private
    FId: Integer;

    [Required, MaxLength(20)]
    FName: string;

    [EmailAddress]
    FEmail: string;

    [DisplayName('class rate')]
    [Range(1, 10, 'Values must be %1:d up to %2:d for field %0:s')]
    FRate: Integer;
  public
    property Id: Integer read FId write FId;
    property Name: string read FName write FName;
    property Email: string read FEmail write FEmail;
    property Rate: Integer read FRate write FRate;
  end;
```

If we try to save such entity with many wrong properties, many validations will fail:

```
procedure SaveWrongCustomer;
var
  Customer: TCustomer;
begin
  Customer := TCustomer.Create;
  Manager.AddOwnership(Customer);
  Customer.Name := 'Too long name for customer';
  Customer.Email := 'foo';
  Manager.Save(Customer);
end;
```

You can use the following code to catch the validation exception and log detailed information:

```
var
  ValidationResult: IManagerValidationResult;
  Error: IValidationError;
begin
  try
    SaveWrongCustomer;
  except
    on E: EEntityValidationException do
    begin
      WriteLn(Format('Validation failed for entity %s:', [E.Entity.ClassName]));
      for ValidationResult in E.Results do
        for Error in ValidationResult.Errors do
          WriteLn('  ' + Error.ErrorMessage);
    end;
  end;
end;
```

Which will generate the following output:

```
Validation failed for entity TCustomer:
  Field FName must have no more than 20 character(s)
  Field FEmail is not a valid e-mail address
  Values must be 1 up to 10 for field class rate
```

Disabling Validations

Data validation is enabled by default. If you have added validators to your mapping, then they will already be enforced when you try to save an entity.

In case you want to disable validations (for example, to improve performance), you can set `TObjectManager.ValidationsEnabled` property to False:

```
  Manager.ValidationsEnabled := False;
  Manager.Save(Customer); // no validation performed
```

Custom Validators

In addition to the built-in validators, you can create your own custom validators, including attributes, that you can apply to your entities.

First, create a new class implementing the `IValidator` interface:

```
uses {...}, Aurelius.Validation.Interfaces;

type
  TMyDataValidator = class(TInterfacedObject, IValidator)
  public
    function Validate(const Value: TValue; Context: IValidationContext): IValidat
ionResult;
  end;
```

The interface has a single method `Validate` that you need to implement, which receives the `Value` to be validated and must return an `IValidationResult` interface:

```
function TMyDataValidator.Validate(const Value: TValue;
  Context: IValidationContext): IValidationResult;
begin
  // Add your own logic to check if Value is valid
  if IsValid(Value) then
    Result := TValidationResult.Success
  else
    Result := TValidationResult.Failed(Format(ErrorMessage,
      [Context.DisplayName]))
end;
```

Then just create your new attribute inheriting from `ValidationAttribute` and override the `GetValidator` method:

```
  MyDataAttribute = class(ValidationAttribute)
  strict private
    FValidator: IValidator;
  public
    constructor Create;
    function GetValidator: IValidator; override;
  end;

{...}

constructor MyDataAttribute.Create;
begin
  inherited Create;
  FValidator := TMyDataValidator.Create;
end;

function MyDataAttribute.GetValidator: IValidator;
begin
  Result := FValidator;
end;
```

After this, all you need is to add the attribute to all cla members you want `MyData` validation to be applied:

```
[MyData]
FMyProp: string;
```

Manual Validation

Validations are performed automatically before an entity is about to be persisted (if `ValidationsEnabled` is false).

But if for any reason you want to validate an entity without persisting it, just call `Validate` method of the object manager:

```
Manager.Validate(MyEntity);
```

If any validation fails, an exception will be raised and you can handle it as usual.

Global Filters

With Aurelius you can define filters that are applied to several entities at once.

You can specify the filters using the `FilterDef` attribute once in any entity of your model, then add a `Filter` attribute for each entity you want to be filtered. For example:

```
[Entity, Automapping]
[FilterDef('Multitenant', '{TenantId} = :tenantId')]
[FilterDefParam('Multitenant', 'tenantId', TypeInfo(string))]
[Filter('Multitenant')]
TProduct = class
private
  FId: Integer;
  FName: string;
  FTenantId: string;
public
  property Id: Integer read FId write FId;
  property Name: string read FName write FName;
  property TenantId: string read FTenantId write FTenantId;
end;
```

And finally, enable filters at the TObjectManager level, using `EnableFilter` method:

```
Manager.EnableFilter('Multitenant')
  .SetParam('tenantId', 'microsoft');
Products := Manager.Find<TProduct>.OrderBy('Name').List;
```

Once you do that, the `SELECT` statement built by Aurelius to retrieve the products will include the filter condition specified in the attribute. The above code would generate an SQL statement like this:

```
SELECT A.NAME, A.ID, A.TENANT_ID
  FROM PRODUCT A
  WHERE A.TENANT_ID = :p0
  ORDER BY A.NAME;

:p0 = 'microsoft'
```

This will apply for **all** entities with the `Multitenant` filter defined, even if they are associations.

This makes it very easy to build multitenant applications, for example. You don't have to worry about adding filters to every Aurelius query you build. Just code it with the regular business logic, and Aurelius global filter will take of adding the filters that apply globally to all entities.

Creating filter definitions

To create a filter definition, use the `FilterDef` and `FilterDefParam` attributes:

```
[FilterDef('Multitenant', '{TenantId} = :tenantId')]
[FilterDefParam('Multitenant', 'tenantId', TypeInfo(string))]
TProduct = class
```

The first parameter of `FilterDef` attribute is the filter name. The second parameter is *optional*, and contains the filter condition that will be applied to all entities that don't explicitly add a filter condition in their `Filter` attribute.

NOTE
Filter definitions are global to a model. Do not add two or more `FilterDef` attributes with the same filter name, even in different entities

Filter conditions and parameters

Filter conditions are SQL condition expressions that are added to the WHERE clause of any SELECT statement used to retrieve entities. They will also be applied when the entity is being queried as an associated object.

```
[FilterDef('Deleted', '{Deleted} = 0')]
```

You should use aliases to refer to database columns, with the class field (or property) name wrapped by brackets (`{ }`), in the same you will use aliases in SQL conditions .

Filter conditions can also use parameters, that are defined by prefixing the parameter name with `:` (colon). In the following example, `tenantId` is a parameter.

```
[FilterDef('Multitenant', '{Multitenant} = :tenantId')]
```

For each parameter in the filter condition, you must explicitly add a `FilterDefParam` attribute in addition to the `FilterDef` attribute, specifying the Delphi type of the parameter. In this example, `tenantId` parameter is of type `string`:

```
[FilterDefParam('Multitenant', 'tenantId', TypeInfo(string))]
```

First argument of `FilterDefParam` is the filter name, second is the parameter name, third is the Delphi type of the parameter.

Applying filters to entities

Once a filter is defined, you can specify which entities might have such filter applied. This is as simple as adding a `Filter` attribute to any entity you want the filter to be applied, specifying the filter name as the first argument:

```
  [Filter('Multitenant')]
  [Filter('Deleted')]
  TProduct = class

{...}

  [Filter('Multitenant')]
  TCustomer = class
```

In the above example, filters `Multitenant` and `Delete` will be applied to entity `Product` when they are enabled. On the other hand, `TCustomer` entity will only have filter `Multitenant` applied to it. Filter `Deleted` will have no effect on entity `Customer`.

WARNING
If you are using an inheritance strategy, you should only apply filters to then entity class which is the root of the class hierarchy. Do not apply the filter in a descendant class.

If a filter is applied to a descendant class in a class hierarchy, Aurelius might bring wrong results if such classes as retrieved as associated objects.

You can also override the default filter condition specified in the filter definition, by passing a new filter condition as the second parameter:

```
  [Filter('Multitenant', {AnotherTenantId} = :tenantId)]
  TOtherClass = class
```

The filter condition must use the same parameters specified in the filter definition.

Enabling filters

Finally, filters are enabled at the TObjectManager level, using `EnableFilter` method.

```
  Manager.EnableFilter('Delete');
  Products := Manager.Find<TProduct>.OrderBy('Name').List;
```

Once you enable a filter for a specific object manager, all SQL query statements executed by that manager will have the filter condition added to the WHERE clause when searching for the involved entity(ies).

WARNING
Filters are always disabled by default. If you don't call `EnableFilter` to enable a specific filter, no conditions are applied to any entity using that filter!

If the filter definition includes a condition that uses parameters, you **must** set the value of each parameter of the filter using `SetParam` method:

```
  Manager.EnableFilter('Multitenant')
    .SetParam('tenantId', 'microsoft');
```

You can also disable a filter using `DisableFilter` method and check if a filter is enabled using `FilterEnabled`:

```
if Manager.FilterEnabled('Multitenant') then
  Manager.DisableFilter('Multitenant');
```

Filter enforcer

When you enable a filter, it "only" applies the SQL WHERE condition to SELECT statements, making all your data automatically filtered. But it doesn't do anything when you insert, update or delete a database record.

Fortunately, Aurelius provides mechanism named "filter enforcer" which does that for you: it makes sure that whenever you modify entity data (create, update, delete), it will be consistent with the condition specified in the filter.

In other words, considering the `Multitenant` filter example: if you have enabled the `Multitenant` filter to retrieve data when `tenantId` is equals to `microsoft`, the filter enforcer will make sure that you also don't create, update or delete an entity if `TenantId` property is not `microsoft`.

Use the following code to activate the filter enforcer:

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

// variable/class field declaration
Enforcer: TFilterEnforcer;

// creating and activating filter enforcer
Enforcer := TFilterEnforcer.Create('Multitenant', 'TenantId', 'FTenantId');
Enforcer.Activate(TMappingExplorer.Default);

// deactivating and destroying filter enforcer
Enforcer.Deactivate(TMappingExplorer.Default);
Enforcer.Free;
```

You can see that the `TFilterEnforcer` class constructor receives three parameters: The first is the filter name (`Multitenant`), the second is the name of a filter condition parameter, the third is the name of the class member (field or property) that will be checked against the parameter value.

Also, when you activate a filter enforcer, you must specify to which mapping explorer (model) it will be applied. The above example is using the default model.

The enforcer will subscribe to key model events. Whenever an entity is about to be inserted, updated or deleted, the enforcer will check if:

1. If the specified filter is active (`Multitenant`);
2. If yes, check if the value of specified class member (`FTenantId`) matches the value of filter parameter (`tenantId`).

If the condition 2 above is tested and fails, an exception will be raised. Optionally you can ask the enforcer to auto comply to the value in either insert and/or update operations:

```
Enforcer.AutoComplyOnInsert := True;
Enforcer.AutoComplyOnUpdate := True;
```

When the two properties above are enabled, the enforcer will check if the class member (`FTenantId`) is empty. If it is, then instead of raising an error when the filter is enabled, it will automatically set the member value using the parameter value. In other words: if a `Multitenant` filter is active with `tenantId` parameter set to `microsoft`, if you try to insert or update a record without specifying the `TenantId` property, the enforce will automatically fill it with the `microsoft` value for you.

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.

TAureliusDataset 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.

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.

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. When you insert/delete records in the dataset, objects will be added/removed to the list.

By default, TAureliusDataset doesn't own the passed list object, meaning you are responsible for destroying the list object itself, TAureliusDataset will not destroy it. You can change this behavior passing a second boolean parameter to `SetSourceList` indicating you want the dataset to destroy it:

```
// Passing True will not require you destroy People list
AureliusDataset1.SetSourceList(People, True);
```

With the code above, you don't have to worry about destroying `People` list.

NOTE
When the source list is owned, Aurelius dataset will destroy it **when it's closed**. If you want to close and reopen the dataset in this case, you must provide a new list object, since the previous one was destroyed.

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*.

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.

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.

Offline 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.

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).

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

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;
```

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
```

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').AsSt
ring;
```

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;
```

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 := TCo
untry.Create;
```

Following code shows how to retrieve the value of an association property using the dataset field:

```
Country := AureliusDataset1.EntityFieldByName('Country').AsEntity<TCountry>;
```

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 by default there is no need to set the Manager property of nested datasets. There is a *TAureliusDataset.ParentManager* property which defaults to true, that indicates that the Manager of the dataset will be same as the Manager of the parent dataset (which is the dataset of the linked DatasetField). In this case, whenever you Post or Delete a record in the detail dataset, the detail object will be immediately persisted in the database.

In case you don't want this behavior (for example, you want the details dataset to save objects in memory and only when the master object is saved you have details being saved at once), you can explicitly set the *Manager* property of the details dataset to nil. This will automatically set the ParentManager property to false:

```
InvoiceDataset.SetSourceList(List);
InvoiceDataset.Manager := Manager1;
// Set Manager to nil so only save items when InvoiceDataset is posted.
// ItemsDataset.ParentManager will become false
ItemsDataset.Manager := nil;
InvoiceDataset.Open;
```

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;
```

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; // now 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
```

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';
```

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;
```

NOTE
The *TCriteriaResult* objects provided to the dataset might be automatically destroyed when the dataset closes, depending on how you provide objects to the dataset. If you use *SetSourceCursor* or *SetSourceCriteria*, they are automatically destroyed. This is because since the objects are fetched automatically by the dataset, it manages it's life-cycle. When you use *SetSourceList* or *SetSourceObject*, they are **not destroyed** and you need to do it yourself.

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.

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 object might be created when the record enters insert state (default) or only when you post the record (if you set *TAureliusDataset.CreateObjectOnPost* property to true). 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 *Append* is called (if you call *Cancel* the object will be automatically destroyed):

```
Customers := TObjectList<TCustomer>.Create;
DS.SetSourceList(Customer); // base class is TCustomer
DS.Open;
DS.Append; // Create a new TCustomer instance
DS.FieldByName('Name').AsString := 'Jack';
DS.Post;
// Destroy Customers list later!
```

If you set *CreateObjectOnPost* to true, the object will only be created on Post.

```
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: T
Object);
begin
  NewObject := TBird.Create;
end;

//<snip>

AureliusDataset1.OnCreateObject := AureliusDataset1CreateObject;
AureliusDataset1.Append; // a TBird object named "Tweetie" will be created here
AureliusDataset1.FieldByName('Name').AsString := 'Tweetie';
AureliusDatase1.Post;
```

NOTE
After *Post*, objects created by TAureliusDataset are not destroyed anymore. See Objects Lifetime Management for more information.

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.

Please refer to the topic using Dataset Fields to learn how the *Manager* property is propagated to datasets which are linked to dataset fields.

Objects Lifetime Management

TAureliusDataset usually 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 the objects it created automatically when inserting new records. So you must be sure to destroy all of them when needed! The only two exceptions are described at the end of this topic.

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.

Exceptions

There are only two exceptions when objects are destroyed by the dataset:

1. A record in Insert state is not Posted.
   When you Append a record in the dataset, an object is created (unless CreateObjectsOnPost property is set to true). If you then Cancel the inserting of this record, the dataset will silently destroy that object.

2. When objects of type TCriteriaResult are passed using SetSourceCursor or SetSourceCriteria.
   In this case the objects are destroyed by the dataset.

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
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.

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.

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;
```

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).

Filtering

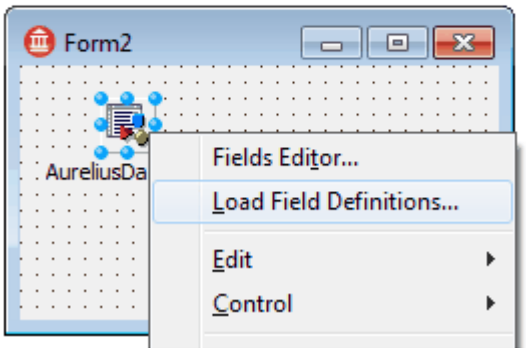
TAureliusDataset supports filtering of records by using regular *TDataset.Filtered* property and *TDataset.OnFilterRecord* event. It works just as any TDataset descendant. Note that when filtering records, you can use regular *Dataset.FieldByName* approach, or you can use *Current<T>* property and access 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;
```

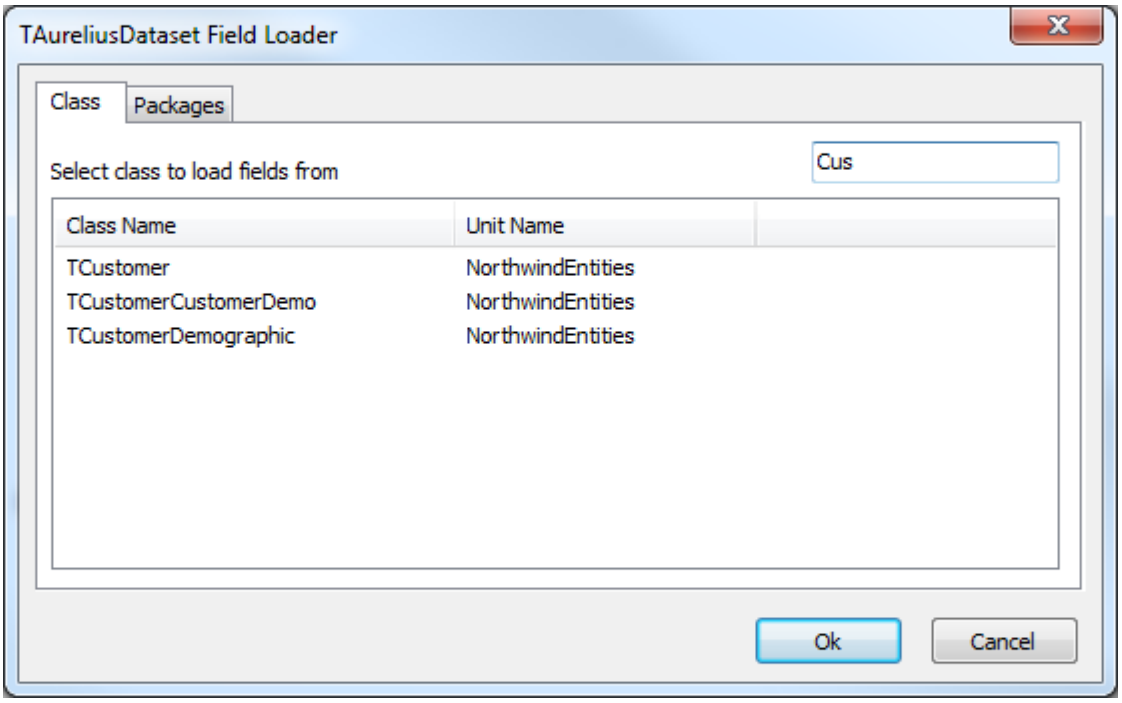
Design-time Support

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.



Select class to load fields from

Cus

Class Name	Unit Name
TCustomer	NorthwindEntities
TCustomerCustomerDemo	NorthwindEntities
TCustomerDemographic	NorthwindEntities

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 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.

Other Properties And Methods

List of TAureliusDataset methods and properties not coverered by other topics in this chapter.

Methods

Name	Description
procedure FillRecord(Obj: TObject);	Updates all dataset field values with the respective property values of `Obj` object. This is useful to "copy" all values from `Obj` to the dataset fields.
procedure RefreshRecord;	Updates all dataset field values from the existing underlying object. Use `RefreshRecord` if you have modified the object properties directly and want the dataset to reflect such changes.

Properties

Name	Description
CreateSelfField: Boolean	When True (default), the dataset will include the *Self* field in the list of default fieldsdefs. If False, the field will not be created.
DefaultsFromObject: Boolean	When True, brings field default values with object state. When inserting a new record in TAureliusDataset, all fields come with null values by default (DefaultsFromObject is False). By setting this property to True, default (initial) value of the fields will come from the property values of the underlying object.
FieldInclusions: TFieldInclusions	Determines which special "categories" of fields will be created automatically by the dataset when it's open and no persistent fields are defined. This is a set of TFieldInclusion enumeration type which have the following options. The value of this property by default is *[TFieldInclusion.Entity, TFieldInclusion.Dataset]*.

Name	Description
IncludeUnmappedObjects: Boolean	When True, the dataset will also create field definitions for object (and lists) properties that are not mapped. In other words, you can view/edit transient object properties. The default is False which means only Aurelius associations will be visible.
ReadOnly: Boolean	If true, puts the dataset in read-only mode, so data cannot be edited by visual data-aware controls. Default is false.
SubpropsDepth: Integer	Allows automatic loading of subproperty fields. When loading field definitions for TAureliusDataset at design-time, or when opening the TAureliusDataset without persistent fields, one *TField* for each property in object will be created. By increasing SubpropsDepth to 1 or more, TAureliusDataset will also automatically include subproperty fields for each property in each association, up to the level indicated by SubpropsDepth. For example, if SubpropsDepth is 1, and there is an association field named "Customer", the dataset will also create fields like "Customer.Name", "Customer.Birthday", etc.. Default is 0 (zero).
SyncSubProps: Boolean	Allows automatic updating of associated fields. When an entity field (e.g., "Customer") of the TAureliusDataset component is modified, all the subproperty fields (e.g., "Customer.Name", "Customer.Birthday") will be automatically updated with new values if this property is set to True. Default is False.
RecordCountMode: TRecordCountMode	When using dataset in paged mode using SetSourceCriteria, by default the total number of records is not known in advance until all pages are retrieved. RecordCount property returns -1 until all records are fetched. You can use this property change the dataset algorithm used to return the RecordCount property value. See below the valid values.

TFieldInclusions

```
type
  TFieldInclusion = (Entity, Dataset);
  TFieldInclusions = set of TFieldInclusion;
```

- *TFieldInclusion.Entity*:
  If present, Aurelius dataset will create entity fields for properties that hold object instances (usually associations). For example, for a class *TCustomer* with a property *Country* of type *TCountry*, an entity field "Country" will be created.
- *TFieldInclusion.Dataset*:
  If present, Aurelius dataset will create dataset fields for properties that hold object lists. For example, for a class *TInvoice* with a property *Items* of type *TList<TInvoiceItem>*, a dataset field "Items" will be created.

TRecordCountMode

```
type
  TRecordCountMode = (Default, Retrieve, FetchAll);
```

- *TRecordCountMode.Default*:
  RecordCount always return -1 if not all records are fetched from the database. No extra statements are performed.
- *TRecordCountMode.Retrieve*:
  An extra statement will performed in database to retrieve the total number of records to be retrieved. RecordCount property will return the correct value even if not all records are fetched from the database. This has a small penalty performance since it requires another statement to be executed. The extra statement will only be executed if RecordCount property is read.
- *TRecordCountMode.FetchAll*:
  All records will be retrieved to properly return the RecordCount value. This is maximum penalty performance in exchange for always returning the correct value of RecordCount property.

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

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

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**
DataSnap	TDataSnapJsonSerializer	TDataSnapJsonDeserializer	TJsonValue	Aurelius.Json.DataSna
SuperObject	TSuperObjectJsonSerializer	TSuperObjectJsonDeserializer	ISuperObject	Aurelius.Json.SuperOb

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;
```

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": "TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBi
eSB0aGlz..."
}
```

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.

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 method 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 *BlobLoader* 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.

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 deserialized. 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 minimize 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;
```

In addition to OnEntityCreated event, the deserializer also provides *Entities* property which contains all objects created by it:

```
property Entities: TEnumerable<TObject>;
```

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
  JsonValue := DataSnapDeserializer.ToJson(Customer);
  // JsonValue must be destroyed later
```

In the previous example, *JsonValue* 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.

Events

Aurelius provides an event system which you can use to receive callback notifications when some events occur, for example, an entity update or a item is included in a collection. This chapter explains how to use this event system and what events are available.

Using Events

Subscribing from code

Events in Aurelius are available in the *Events* property of the TMappingExplorer object. Such property refers to a *TManagerEvents* (declared in unit `Aurelius.Events.Manager`) object with several subproperties, each to them related to an event. For example, to access the *OnInserted* event of the default *TMappingExplorer*:

```
uses {...}, Aurelius.Mapping.Explorer, Aurelius.Events.Manager;

TMappingExplorer.Default.Events.OnInserted.Subscribe(
  procedure(Args: TInsertedArgs)
  begin
    // Use Args.Entity to retrieve the inserted entity
  end
);

TMappingExplorer.Default.Events.OnUpdated.Subscribe(
  procedure(Args: TUpdatedArgs)
  begin
    // Use Args.Entity to retrieve the updated entity
  end
);
```

In a less direct way, using method reference instead of anonymous method:

```
uses {...}, Aurelius.Mapping.Explorer, Aurelius.Events.Manager;

procedure TSomeClass.MyInsertedProc(Args: TInsertedArgs);
begin
  // Use Args.Entity to retrieve the inserted entity
end;

procedure TSomeClass.MyUpdatedProc(Args: TUpdatedArgs);
begin
  // Use Args.Entity to retrieve the updated entity
end;

procedure TSomeClass.RegisterMyEventListeners;
var
  Events: TManagerEvents;
begin
  Events := TMappingExplorer.Default.Events;
  Events.OnInserted.Subscribe(MyInsertedProc);
  Events.OnUpdated.Subscribe(MyUpdatedProc);
end;
```

The events are available in the TMappingExplorer object so the listeners will receive notifications about any event fired by any *TObjectManager* created that references the specified TMappingExplorer object. In other words, the events are "global" for that mapping explorer.

Listeners are method references that receive a single object as a parameter. Such object has several properties containing relevant information about the event, and differ for each event type. Names of event properties, method reference type and arguments follow a standard. The event property is named "On<event>", method reference type is "T<event>Proc" and parameter object is "T<event>Args". For example, for the "Deleted" event, the respective names will be "*OnDeleted*", "*TDeletedProc*" and "*TDeletedArgs*".

All events in Aurelius are multicast events, which means you can add several events handlers (listeners) to the same event. When an event occurs, all listeners will be notified. This allows you to add a listener in a safe way, without worrying if it will replace an existing listener that might have been set by other part of the application. You should use *Subscribe* and *Unsubscribe* methods to add and remove listeners, respectively. Note that since listeners are method references, you must sure to unsubscribe the same reference you subscribed to:

```
var
  LocalProc: TInsertedProc;
begin
  LocalProc := MyInsertedProc;
  Events.OnInserted.Subscribe(LocalProc);
  {...}
  Events.OnInserted.Unsubscribe(LocalProc);
end;
```

Passing just the method name doesn't work:

```
Events.OnInserted.Subscribe(MyInsertedProc);
{...}
// this will NOT unsubscribe the previous subscription:
Events.OnInserted.Unsubcribe(MyInsertedProc);
```

TAureliusModelEvents Component

An alternative, more RAD way to use events is the *TAureliusModelEvents* component. Just drop the component in the form and double click the desired event in the object inspector to create an event handler.

The events available are exactly the same ones that you can set from code, like OnInserting, OnSqlExecuting, etc. See all the available events in this chapter.

Key properties

Name	Description
ModelName: string	The name(s) of the model(s) to be used by the manager. You can leave it blank, if you do it will use the default model. Two or more model names should be separated by comma. From the model names it will get the property TMappingExplorer component that will be passed to the *TDatabaseManager* constructor to create the instance that will be encapsulated.

Note the events are "cumulative", the same way you do it from code. It means that if you add two or more TAureliusModelEvents in your application and set an event handler for the same event in all of them, all the handlers will be fired. That's very convenient to event handler code that is specific to each context in your app.

Using Attributes

A third and even more straightforward way to respond to events is to use attributes. You can simply add an attribute to a method of an entity class, and that method will be invoked when the event is triggered. Consider the following example:

```
type
  {$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}
  TCustomer = class
  strict private
    [OnInserting] procedure OnInserting(Args: TInsertingArgs);
    [OnInserted] procedure OnInserted;

    [OnUpdated, OnInserted] procedure AfterModification;
  {...}
  end;
```

When `TCustomer` entity is about to be inserted in the database, the `OnInserting` method will be invoked. After the record is inserted, the method `OnInserted` is invoked.

WARNING
By default, Delphi doesn't generate RTTI for non-published methods. That's why you must add the directive `{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}` to your class. If you don't do that, Aurelius won't know about the mentioned methods and they will not be invoked when the events are triggered!

Note that you can use the same method to handle more than one event. The method `AfterModification` will be invoked when event `OnUpdated` is fired, and also when `OnInserted` is fired.

Finally, you can use two different signatures for the methods: the method can receive a single `Args` parameters of the type of the event (see the available events below).

Alternatively, you can declare the method without specifying any parameter. This is useful if you don't need any information from event arguments and you want to keep your class as clean as possible (no dependency on a specific event type).

Available events

OnInserting Event

Occurs right before an entity is inserted (create) in the database. Note that the event is fired for **every** entity that is about to be inserted. For example, a single Manager.Save call might cause several entities to be inserted, due to cascades defined in the associations. In this case the event will be fired multiple times, one for each saved entity, even when the developer only called *Save* once.

Example:

```
TMappingExplorer.Default.Events.OnInserting.Subscribe(
  procedure(Args: TInsertingArgs)
  begin
    // code here
  end
);
```

TInsertingArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The entity about to be inserted.

Name	Description
Master: TMasterObjectValue	The parent object of the object being inserted. This property comes with a value in the case of list items (ManyValuedAssociation) that don't have a reference back to parent (unidirectional). *TMasterObjectValue* has two relevant properties: "*MasterObject*" which is the instance of parent object, and "*MasterAssocMember*" which is the name of the list property the item being inserted belongs to (for example, "InvoiceItems").

OnInserted Event

Occurs right after an entity is inserted (create) in the database. Note that the event is fired for **every** entity inserted. For example, a single Manager.Save call might cause several entities to be inserted, due to cascades defined in the associations. In this case the event will be fired multiple times, one for each saved entity, even when the developer only called *Save* once.

Example:

```
TMappingExplorer.Default.Events.OnInserted.Subscribe(
  procedure(Args: TInsertedArgs)
  begin
    // code here
  end
);
```

TInsertedArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The entity that was inserted.
Master: TMasterObjectValue	The parent object of the object being inserted. This property comes with a value in the case of list items (ManyValuedAssociation) that don't have a reference back to parent (unidirectional). *TMasterObjectValue* has two relevant properties: "*MasterObject*" which is the instance of parent object, and "*MasterAssocMember*" which is the name of the list property the item being inserted belongs to (for example, "InvoiceItems").

OnUpdating Event

Occurs right before an entity is about to be updated in the database.

Example:

```
TMappingExplorer.Default.Events.OnUpdating.Subscribe(
  procedure(Args: TUpdatingArgs)
  begin
    // code here
  end
);
```

TUpdatingArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The entity that is going to be updated.
OldColumnValues: TDictionary<string, Variant>	Represents the old object state using column name/value pairs. Don't confuse it with property names/values. For example, if the object has a property named "Name" that is mapped to a column database "CUSTOMER_NAME", the dictionary will contain "CUSTOMER_NAME" in the string key, and the respective value. Thus, associations are also represented by the foreign key column names/values.
NewColumnValues: TDictionary<string, Variant>	Same as *OldColumnValues*, but contains the new state values. Comparing what has changed between *NewColumnValues* and *OldColumnValues* will give you the names of the columns that will be updated in the database.
ChangedColumnNames: TList<string>	Contains a list of names of all columns that will be updated in the UPDATE statement.
RecalculateState: Boolean	If you have changed any property value of the entity that is about to be updated, you need to set *RecalculateState* to True to force Aurelius to recalculate the columns that were modified and update the object state in the manager cache. For better performance, leave it false if you haven't modified any property.

OnUpdated Event

Occurs right after an entity is updated in the database.

Example:

```
TMappingExplorer.Default.Events.OnUpdated.Subscribe(
  procedure(Args: TUpdatedArgs)
  begin
    // code here
  end
);
```

TUpdatedArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The entity that was updated.
OldColumnValues: TDictionary<string, Variant>	Represents the old object state using column name/value pairs. Don't confuse it with property names/values. For example, if the object has a property named "Name" that is mapped to a column database "CUSTOMER_NAME", the dictionary will contain "CUSTOMER_NAME" in the string key, and the respective value. Thus, associations are also represented by the foreign key column names/values.
NewColumnValues: TDictionary<string, Variant>	Same as *OldColumnValues*, but contains the new state values. Comparing what has changed between *NewColumnValues* and *OldColumnValues* will give you the names of the columns that will be updated in the database.
ChangedColumnNames: TList<string>	Contains a list of names of all columns that were updated in the UPDATE statement.

OnDeleting Event

Occurs right before an entity is about to be deleted from the database. Note that the event is fired for **every** entity deleted. For example, a single Manager.Remove call might cause several entities to be deleted, due to cascades defined in the associations. In this case the event will be fired multiple times, one for each deleted entity, even when the developer only called *Remove* once.

Example:

```
TMappingExplorer.Default.Events.OnDeleting.Subscribe(
  procedure(Args: TDeletingArgs)
  begin
    // code here
  end
);
```

TDeletingArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The entity about to be deleted.

OnDeleted Event

Occurs right after an entity is deleted from the database. Note that the event is fired for **every** entity deleted. For example, a single Manager.Remove call might cause several entities to be deleted, due to cascades defined in the associations. In this case the event will be fired multiple times, one for each deleted entity, even when the developer only called *Remove* once.

When the event is fired, the entity object is still a valid reference, but will be destroyed right after the event listener returns.

Example:

```
TMappingExplorer.Default.Events.OnDeleted.Subscribe(
  procedure(Args: TDeletedArgs)
  begin
    // code here
  end
);
```

TDeletedArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Entity: TObject	The deleted entity.

OnCollectionItemAdded Event

Occurs when an item is added to a collection, at database level. In other words, when the foreign key of an item entity is set to point to the parent entity.

Example:

```
TMappingExplorer.Default.Events.OnCollectionItemAdded.Subscribe(
  procedure(Args: TCollectionItemAddedArgs)
  begin
    // code here
  end
);
```

TCollectionItemAddedArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Parent: TObject	The parent entity which contains the collection where the item was added to.
Item: TObject	The item entity added to the collection.

Name	Description
MemberName: string	The member name (field or property) of the parent entity that holds the collection.

OnCollectionItemRemoved Event

Occurs when an item is removed from a collection, at database level. In other words, when the foreign key of an item entity is set to null (or to a different parent entity).

Example:

```
TMappingExplorer.Default.Events.OnCollectionItemRemoved.Subscribe(
  procedure(Args: TCollectionItemRemovedArgs)
  begin
    // code here
  end
);
```

TCollectionItemRemovedArgs Properties

Name	Description
Manager: TBaseObjectManager	The TObjectManager object which fired the event.
Parent: TObject	The parent entity which contains the collection where the item was removed from.
Item: TObject	The item entity removed from the collection.
MemberName: string	The member name (field or property) of the parent entity that holds the collection.

OnSqlExecuting Event

Occurs right before an SQL statement is executed.

Example:

```
TMappingExplorer.Default.Events.OnSqlExecuting.Subscribe(
  procedure(Args: TSQLExecutingArgs)
  begin
    // code here
  end
);
```

TSQLExecutingArgs Properties

Name	Description
SQL: string	The SQL statement that will be executed.

Name	Description
Params: TEnumerable<TDBParam>	A list of *TDBParam* objects used for the SQL statement execution. The TDBParam object has the properties *ParamName*, *ParamType* and *ParamValue*.

Advanced Topics

Here we present some advanced topics about TMS Aurelius.

Global Configuration

TMS Aurelius has a single global class that has some properties for setting global configuration. This class is declared in unit `Aurelius.Global.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:

SimuleStatements: Boolean
If true, all statements are not executed on the DBMS, but appear in the listeners.

MaxEagerFetchDepth: Integer
Indicates the maximum depth to load objects in eager loading associations. Beyond this depth, the objects still load in lazy mode.

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.

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).

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.

AutoMappingDefaultCascade: TCascadeTypes
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;
```

DefaultStringColWidth: Integer
Defines the width for string (usually varchar) columns when the width was not particularly specified in Column attribute.

UseTransactionsInManager: Boolean
Defines the default value for the TObjectManager.UseTransactions. Default is true, meaning all internal manager operations will be performed with transactions. If you want to disable this (mostly for backward compatibility) for the whole application instead of setting the property for each manager, you can set this property to false.

UseTransactionsInDBManager: boolean
Defines the default value for the TDatabaseManager.UseTransactions. Default is false, meaning no transactions will be used to execute SQL statements for creating/updating tables, columns, foreign keys, etc. If you want to enable this for the whole application instead of setting the property for each database manager, you can set this property to true.

Object Factory

In several conditions, Aurelius needs to create entity instances. For example, when retrieving entities from the database, Aurelius needs to create instances of those entities. To do that, Aurelius uses an internal object factory. By default, this factory just creates entities by calling a parameter-less constructor named "Create".

Such mechanism works in most cases. But in the case you want to create your entities yourself (for example all your entities have a *Create* constructor that need to receive a parameter), you can change the object factory and implement it yourself.

To do that, all you need is to implement an *IObjectFactory* interface (declared in unit `Bcl.Rtti.ObjectFactory`):

```
IObjectFactory = interface
  function CreateInstance(AClass: TClass): TObject;
end;
```

It has a single method *CreateInstance* which receives the *TClass* and must return a *TObject* which is a new instance of that class.

Once you have created such instance, you can replace the default one used by Aurelius. You can do it at the TMappingExplorer level, thus changing the factory for everything in Aurelius that is related to that explorer:

```
TMappingExplorer.Default.ObjectFactory := MyObjectFactory;
```

Or you can change it for a TObjectManager object specifically. This gives you more fine-grained control, for example in case your entities need to be created under a specific context:

```
Manager.ObjectFactory := MyObjectFactory;
```

About

This documentation is for TMS Aurelius.

In this section:

What's New in TMS Aurelius

Version 5.11 (September-2022)

- **New: Associations now can be used in abstract entities.**
- **Improved**: Design-time components were greyed out in component palette if current platform was different than Win32.

Version 5.10 (August-2022)

- **Fixed:** Access Violation when using SubCriteria in Aurelius queries (regression). See https://support.tmssoftware.com/t/exception-with-latest-version/18897

Version 5.9 (July-2022)

- **New: You can now use property projections when retrieving regular entities. This will allow you to optimize the queries by limiting the properties to be retrieved for the entities.**

```
Estimates := Manager.Find<TEstimate>
  .Select(TProjections.ProjectionList
    .Add(Linq['EstimateNo'])
    .Add(Linq['Customer'])
    .Add(Linq['Customer.Name'])
    .Add(Linq['Customer.Sex'])
    .Add(Linq['Customer.Country'])
    .Add(Linq['Customer.Country.Id'])
  )
  .FetchEager('Customer')
  .FetchEager('Customer.Country')
  .List
```

- **New: Automapping attribute now can receive an engine class that allows customizing automatic naming.**
- **Fixed:** TFetchMode.Eager now forces eager loading of many-valued associations as well.
- **Fixed**: Typo in design-time `TAureliusConnection` dialog.

Version 5.8 (March-2022)

- **Improved:** BREAKING CHANGE: Range and MinLength validators don't fail anymore if value is empty.

Version 5.7 (February-2022)

- **Improved:** Now a custom id generator can be defined for composite ids.
- **Improved:** When an entity is deleted using Manager.Remove, its cascaded many-valued association lists are also cleared.
- **Improved:** Better error message when trying to use an entity with no Id attribute configured.
- **Improved:** Most database drivers now implement `IDBResultSet3` interface, which provides the `FieldCount` method.
- **Improved:** Better error message when trying to use associated entities not belonging to the model.
- **Improved:** Better error messages for mapping errors, it's now informing the offending entity class.
- **Fixed:** `MaxLength` validator failing for null string values.
- **Fixed:** Unique key names were being generated in Delphi 11 with different values then previous Delphi versions.
- **Fixed:** Still range check errors (Integer overflow) was being raised in `TAureliusDataset` in some situations when the dataset had a high number of fields.
- **Breaking change**: `TEvent<T>` type moved to unit `Bcl.Types`.

Version 5.6 (October-2021)

- **New:** Aurelius dictionary allows you to write queries in a much easier and safe way, even with associated objects:

```
Manager.Find<TInvoice>
  .Select(Dic.Invoice.Total.Sum)
  .Where(Dic.Invoice.Customer.Country.Name = 'Brazil')
```

- **New:** Aurelius dictionary now can be generated in many ways: from existing databases, from existing entity classes in your application, or even from external command-line tool.
- **New:** Aurelius dictionary validator makes sure your dictionary is never out of sync with the real entity classes, ensuring that you safely write queries that won't crash at runtime.
- **New**: Auto alias mechanism allows querying by associated objects without the need to use `CreateAlias`:

```
Manager.Find<TEstimate>
  .Where(Linq['Customer.Country.Name'] = 'United States')
```

- **Improved:** Using global filter in a descendant entity class (using an inheritance strategy) is not supported and now an exception is raised if you try to use it.
- **Fixed:** Integer overflow when opening TAureliusDataset with too many fields.

- **Fixed**: Very specific error happening when a global filter was applied to an associated object, and such object is a descendant class in a joined-table hierarchy. For example, suppose you are querying for `TInvoice` entities. Such `TInvoice` entity has an association `Customer` of type `TSpecificCustomer`. Such `TSpecificCustomer` is part of a joined-table hierarchy and is not the root class, but a descendant one. The bug would happen if there is a global filter declared specifically in `TSpecificCustomer` class, and such filter is enabled. Aurelius would generate an invalid SQL statement.

Version 5.5 (September-2021)

- **New:** Delphi 11 support.

Version 5.4 (July-2021)

- **Fixed:** Wrong data retrieved in the following corner case: a criteria loading an associated entity that is part of a joined-table inheritance, and such class has a global filter. Also, ElevateDB, NexusDB and AbsoluteDB won't support such construction. This required a significant refactor in the SQL building mechanism and changed the final generated SQL. Please pay attention to any possible issues, specially when dealing with entities in a joined-table hierarchy.

- **Fixed:** Automapping in abstract entities was requiring Id attribute to be present. Now it's optional (Id can be defined in concrete entity classes).

- **Fixed**: `TCriteria<E>.FetchEager` was incorrectly returning a `TCriteria` class instead of `TCriteria<E>`.

- **Fixed:** Issue when `OnUpdating` event had two subscribers. If the first subscribers modifies data, the second subscriber won't get the correct value in `NewColumnValues` parameter.

Version 5.3 (Jun-2021)

- **New: TCriteria.FetchEager method allows defining a specific lazy association to be loaded eagerly in a specific query.**

- **Improved:** Nullable type `Nullable<T>` is now in unit `Bcl.Types.Nullable`. Old unit `Aurelius.Types.Nullable` still exist but we recommend gradually moving the reference to the new unit.

- **Fixed:** `TAureliusDataset.BeforeScroll/AfterScroll` events not firing in detail dataset when the parent record was changed.

- **Fixed:** Database update failing to create non-unique indexes when the index name had spaces and other non-id characters.

Version 5.2 (Apr-2021)

- **New**: **Abstract entities. You can now add mapping attributes to a class tagged with the [AbstractEntity] attribute,** create entities inheriting from such class and having the mapping be inherited from abstract to concrete entity.

```
[AbstractEntity, Automapping]
TBaseEntity = class
strict private
  FCreatedAt: TDateTime;
  FUpdatedAt: Nullable<TDateTime>;
public
  property CreatedAt: TDateTime read FCreatedAt write FCreatedAt;
  property UpdatedAt: Nullable<TDateTime> read FUpdatedAt write FUpdatedAt;
end;

[Entity, Automapping]
TCustomer = class(TBaseEntity)
{...}
end;
```

- **Improved**: Additional checks when unloading SQLite native library to avoid AV in very specific situations (units finalizing in a different order).
- **Fixed**: Calling `FieldDefs.Update` from `TAureliusDataset` would cause the dataset to be open with no data, if objects were previously provided using method `SetSourceCriteria`, `SetSourceCursor` or `SetSourceList` (with an owned list).

Version 5.1 (Mar-2021)

- **New: TObjectManager.Validate method to force an entity validation without saving.**
- **Improved: MusicLibrary demo is now multitenant** and includes examples for global filters, data validation and attributed-based events.
- **Improved:** Native MSSQL Driver now supports query parameters of type `TBcd` (`FmtBCD`).
- **Improved:** `TAureliusDataset` now supports `TByteField` (`ftByte`) and `TLongWordField` (`ftLongWord`).
- **Fixed:** Global filters not working with entity classes using joined tables hierarchy.
- **Fixed:** Execution of batch statements causing Access Violation when `Assertions` compiler option was set to off.
- **Fixed:** Access Violation when enabling filters in managers using the default mapping explorer.
- **Fixed:** Error message `"Filter definition not found"` when enabling a global filter before the mapping explorer was used for the first time.

- **Fixed:** `AutoComplyOnInsert` option in global filters not working, sometimes, when used in an entity class with single table inheritance.
- **Fixed:** Several methods in `TMappingExplorer` were incorrectly flagged as deprecated.

Version 5.0 (Mar-2021)

- **New: Data validation system provides a way to add validation rules to entities that make sure your entities will be always persisted in a valid state.** Now you can add rules via attributes, create custom validation, and more:

```
[Entity, Automapping]
TCustomer = class
strict private
  FId: Integer;

  [Required, MaxLength(20)]
  FName: string;

  [EmailAddress]
  FEmail: string;

  [DisplayName('class rate')]
  [Range(1, 10, 'Values must be %1:d up to %2:d for field %0:s')]
  FRate: Integer;
```

- **New: Global filters mechanism allows applying query filters to all entities at once.** Among other things, it's a very handy feature for single-database multitenant applications.

```
  [FilterDef('Multitenant', '{TenantId} = :tenantId')]
  [FilterDefParam('Multitenant', 'tenantId', TypeInfo(string))]
  [Filter('Multitenant')]
  TProduct = class
{...}
  Manager.EnableFilter('Multitenant')
    .SetParam('tenantId', 'microsoft');
  Products := Manager.Find<TProduct>.OrderBy('Name').List;
```

- **New: It's now possible to handle events directly from the entity class, using event-handling attributes**. This way you can organize better your business logic letting the entity class itself handle many of logic when persistence events happen.

```
type
  {$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}
  TCustomer = class
  strict private
    [OnInserting] procedure OnInserting(Args: TInsertingArgs);
    [OnInserted] procedure OnInserted;

    [OnUpdated, OnInserted] procedure AfterModification;
  {...}
  end;
```

- **New: Filter enforcer mechanism uses global filter definitions to make sure any data modification complies with the active filter**. In multitenant applications, for example, it ensures that no data is saved, updated or deleted if the record belongs to a tenant different than the one specified by the current `tenantId` parameter.
- **New: Entity-level validation via `OnValidate` attribute**. Add the `OnValidate` to any method of your entity class that you want to be called before an entity validation:

```
  TCustomer = class
    [OnValidate]
    function CheckBirthday: IValidationResult;
    [OnValidate]
    function CheckName(Context: IValidationContext): IValidationResult;
```

- **New: TObjectManager.AddOwnership method** makes sure Aurelius object manager will always destroy the passed object, no matter if the persistence operations

```
  Customer := TCustomer.Create;
  Manager.AddOwnership(Customer);
  Manager.Save(Customer);
  // no need to destroy Customer
  // even if Save fails
```

- **New: TAureliusDataset methods RefreshRecord and FillRecord**. It's useful when you have modified the object properties directly and want the dataset to reflect such changes.
- **New: TAureliusDataset now can optionally own the source list, destroying it when it closes**. Now the `SetSourceList` method of Aurelius dataset can receive an optional boolean parameter that, if true, will cause the list to be automatically destroyed when the dataset is closed. You won't have to worry about destroying the list in this case.
- **New: TMappingSetup.ModelName property** allows automatically loading entities from a different model when using mapping setups.
- **Improved:** More performance improvements.
- **Improved:** `ILIKE` operator now working on all databases (using `UPPER` when the database doesn't natively support `ILIKE` syntax).
- **Fixed:** Access Violation when `MappedBy` parameter of a ManyValuedAssociation attribute referenced an invalid class member.

- **Fixed:** Internal method `TMappingExplorer.GetAssociations` causing Access Violation when the passed class was not an Aurelius entity (regression).
- **Fixed:** `INSERT` statement was being generated with a wrong field name when such name needed special treatment due to not being a valid identifier.
- **Fixed:** Memory leak when an `EPropertyNotFound` exception was raised from using a wrong `CreateAlias` call in Aurelius criteria. For example, in code `Find<TMyObject>.CreateAlias('WrongPropertyName', 'p')`, an expected exception will be raise because `WrongPropertyName` property doesn't exist. However, this was causing a memory leak.

Version 4.18 (Sep-2020)

- **Improved:** Overall performance increase, with Aurelius now being up twice faster, specially in insert and update operations.
- **Improved:** Better handling of transactions when using UIB components. Now UIB driver automatically opens a transaction to perform SQL statements, if no transaction is active.
- **Improved:** AnyDAC support is deprecated. AnyDAC was the predecessor of FireDAC, and we considered it doesn't make sense anymore to keep supporting it.
- **Improved:** Support for Boolean fields in ElevateDB databases.
- **Breaking change:** The way to use internal classes TInsertCommand and TUpdateCommand has been modified.
- **Fixed:** TAureliusDataset now returns correct value when reading OldValue property of dataset fields.
- **Fixed:** AbsoluteDB boolean literal regression. Value projections of type boolean, like "Linq.Value<Boolean>(False)" was broken.
- **Fixed:** Access Violation (instead of a better error message) when the Id mapping attribute makes a reference to a class member that doesn't exist.
- **Fixed:** Direct Oracle Access driver now works fine with Unicode memo fields (NCLOB).

Version 4.17 (Aug-2020)

- **New: TRecordCountMode.FetchAll option in TAureliusDataset**. Allows you to force dataset to load (fetch) all records to retrieve the correct value for RecordCount property.
- **Improved:** Several internal classes were refactored, bringing an average 10-20% performance increase in database operations (retrieve and update data).
- **Fixed:** Regression bug - wrong behavior when the MappedBy parameter of a ManyValuedAssociation attribute references an association which is part of the id of the associated class. For example, in the declaration *[ManyValuedAssociation([], [], 'FParent')]*, the FParent is part of the Id of TChild class. Fixed in 4.17.2, regression introduced in 4.17.

Version 4.16 (Jun-2020)

- **Improved:** Native SQL Server driver (MSSQL) performance increase when executing data modification statements.
- **Improved:** Internal changes allows updating of entities that have mapping fields/ properties to database fields with special names (like "name[0]").
- **Fixed:** Native MSSQL driver saving null even when the parameter was an empty string.
- **Fixed:** Native MSSQL driver error "Cannot insert explicit value for identity column in table <table> when IDENTITY_INSERT is set to OFF" even after an SQL statement was executed to set IDENTITY_INSERT to ON.
- **Fixed:** "Invalid referenced column name defined in join column" error in a very specific situation. Conditions: a) bidirectional association using a non-id column as reference. b) The non-id column is declared after the many-valued association field in parent class.

Version 4.15 (Jun-2020)

- **Fixed:** Batch updates using SQLite native driver was not updating field values when parameter was null.

Version 4.14 (May-2020)

- **New: Batch Updates mechanism allows inserting, updating or deleting an arbitrary number of database records using a single SQL statement, improving peformance in such cases.** TObjectManager now introduces a BatchSize property, which is used in conjunction with cached updates mechanism. When BatchSize is greater than 1, several similar cached actions are merged together in a single SQL statement, improving performance by reducing communication with the database.
- **New: TObjectManager.CachedCount property.** Provides information about how many actions are cached to be applied when ApplyUpdates is called.
- **New: Support for Delphi 10.4 Sydney.**

Version 4.13 (Apr-2020)

- **Fixed:** Sporadic Assert failure in MSSQL native driver.

Version 4.12 (Apr-2020)

- **New: Cached updates mechanism in TObjectManager.** You can now defer all SQL statements executed in the database by setting TObjectManager.CachedUpdates to true. In this mode, you can call manager methods like Save, Flush, Remove to manipulate objects, and the database will not be hit until you call TObjectManager.ApplyUpdates.
- **New: "Cached Updates" demo shows how to use new feature cached udpdates feature.**

- **Improved:** Avoided some Access Violations in Object Manager when the entities contained associate lists with nil entities in the list. This was not considered a bug because it's up to the developer to make sure there are no nil entities in lists. The behavior is unpredictable if this happens and those nil pointers should be avoided. But now the code will at least not raise Access Violations.
- **Fixed:** Adding/removing records in the detail dataset was not flagging the master record (parent) as modified.
- **Fixed:** SQL boolean literal in MSSQL dialect wrong when UseBoolean is true.
- **Fixed:** When retrieving entities having associations that are restricted by Where attribute, the entity was not being retrieved if the association didn't exist in database. For example, suppose an entity TInvoice with an associated TCustomer entity. Such TCustomer entity has a Where attribute restricting its usage. When retrieving a TInvoice entity, if the associated customer in the database was null, the TInvoice object will not be returned, instead the result will be nil (as if the invoice didn't exist in the database).
- **Fixed:** Regression with saving widememo data using FireDAC. Error was introduced in 4.11. Current error now is like [FireDAC][Phys][ODBC]-345. Data too large for variable [A_KOMUNIKAT]. Max len = [4000], actual len = [23608] Hint: set the TFDParam.Size to a greater value.
- **Fixed:** Memory leak when an exception was raised while loading properties of an entity.

Version 4.11 (Mar-2020)

- **Fixed:** Native MSSQL driver causing "Function Sequence Error" in some situations (when an IDBConnection interface is only released at the end of the application).
- **Fixed:** Native MSSQL native driver raising an error "Invalid precision value" when updating a Memo/WideMemo with null/empty string value.
- **Fixed:** Importing Firebird database schema fails when the database have indexes only with "computed by" columns.
- **Fixed:** Error "-19. Data type conversion is not supported" when saving Unicode blobs using FireDac in MySQL databases.
- **Fixed:** Rare Invalid Pointer in Win64 platform when using TAureliusDataset.
- **Fixed:** Legacy serializer/deserializer (DataSnap) raising error when serializing proxied associations with composite keys.

Version 4.10 (Nov-2019)

- **New: Support for Android 64-bit platform (Delphi 10.3.3 Rio).**
- **Fixed:** Setting TAureliusDataset.RecNo had no effect in some situations (not correctly changing the current record).
- **Fixed:** Still an extra fix for the AV/Invalid Pointer Operation issue described in version 4.9.

- **Fixed:** Using Literal projection with date time values in some databases (like SQL Server) failed in Finish systems and any other language that set the time separator to a character different than ":".

Version 4.9 (Oct-2019)

- **New: "Driver" parameter in MSSQL native driver allows to explicitly define which driver to use to connect.** By default the most recent installed driver is automatically used, but this feaure is useful for testing purposes, or if you have a specific technical reason to use a specific driver, or an older version.
- **Improved: In TAureliusDataset import field definition dialog you can now order the list of available by unit name.** Just click the column name in the list view and the list will be ordered, either by unit name or class name.
- **Fixed:** Sporadic Access Violation or Invalid Pointer Operation when using object manager to retrieve entities that inherit from TInterfacedObject or any other class that implement interface reference counting. The error was more likely to happen when compiling for Win64 platforms. This issue was introduced in TMS Aurelius 4.5. If you are using any version from 4.5 until 4.8 and if your have any Aurelius entity that inherits from TInterfacedObject (or any other class that implement automatic reference counting), it's **strongly recommended** that you **update to 4.9 version or above**.

Version 4.8 (Sep-2019)

- **New: TAureliusManager component for persisting objects using a RAD approach.** TAureliusManager component allows an even easier way to persist Aurelius entities. It encapsulates TObjectManager in a RAD way: just drop the component in the form, associate with a TAureliusConnection component and you are ready to go.
- **New: TAureliusDBSchema component for updating database schema using a RAD approach.** It's now easier than ever to create or update your database structure (tables and fields). Just drop a TAureliusDBSchema component in the form, associate it with a TAureliusConnection component and use one of this methods, like UpdateDatabase. It automatically instantiates and encapsulates the TDatabaseManager class that creates and validates database structure.
- **New: TAureliusModelEvents component for setting mapping events using a RAD approach.** Using events in Aurelius is now easier by using TAureliusModelEvents component. Simply drop it in a form and set the desired event handler(s) using the object inspector. It's as simple as that.
- **New: Where attribute allows custom filtering when retrieving entities and many-valued associations.** You can now add a Where attribute to an entity class that provides an additional SQL expression to be added to the WHERE clause of SELECT statement used to retrieve the entities. One use case for this is soft delete: you can add an SQL clause like "*{Deleted}* = '*T*'", for example. This will prevent such entities to be retrieved if the Deleted field in the database is "T". The Where attribute can also be used in a many-valued

association to filter records retrieved in the list. You could have, for example, two TAddress lists, one for valid addresses and another for invalid ones, properly filtering those using the Where attribute.

- **New: TAureliusDataset.ReadOnly property.** You can set ReadOnly property of TAureliusDataset to true to easily put the dataset in read-only mode with a single line. This prevent data for being edited in data-aware controls.
- **Improved: Design-time field loader form now persists its size and position.** When using TAureliusDataset at design-time you can right-click and choose "Load field definitions..." to open a dialog form that allows you to choose a BPL package and load fields from entities. If you position or resize that dialog form, its size and position will be persisted: if you close and reopen the dialog at later time, last size and position will be restored.
- **Fixed: SQL error when mapping fields to expressions like field arrays "fieldname[index]".** For example, you can now map to Postgres array fields, using a mapping like "*[Column('fieldname[1]', [])]*".

Version 4.7 (Jun-2019)

- **New (4.7.1.1 July-2019): macOS 64 support in Delphi Rio 10.3.2.**
- **Fixed:** TAureliusDataset.RecNo property not properly working with filtered dataset.
- **Fixed:** TAureliusDataset displaying incorrect data when dataset is filtered and some data modification is performed (like record insert or delete).
- **Fixed:** Error when inserting entities belonging to a joined-table inheritance hierarchy, when the Id in the ancestor class was declared with TColumnProp.NoInsert.
- **Fixed (4.7.1):** SQL error in some situations when using Take or Skip criteria methods in Oracle databases.

Version 4.6 (May-2019)

- **Fixed:** TAureliusDataset.RecNo property not properly working with filtered dataset.
- **Fixed:** Error when inserting entities part of a joined-table inheritance, when the Id in the ancestor class was declared with TColumnProp.NoInsert.

Version 4.5 (Mar-2019)

- **New: TSQLGenerator.UseBoolean is now avaiable for all SQL dialects.**
- **Fixed:** Unexpected wrong behavior with boolean (BIT) fields using native Aurelius MSSQL driver.
- **Fixed:** Error "cannot find datatype Computed (Identity)" when generating entities from Delphi Rio IDE.

- **Fixed:** Nullable.Create overloaded constructor which received an initial value was still keeping the nullable with HasValue flag set to True.

Version 4.4 (Jan-2019)

- **New: Support for SAP SQL Anywhere database - former Sybase SQL Anywhere, Adaptative SQL Anywhere (ASA).**
- **New: Support for NativeDB components (adapter for TASASession).**
- **New: TDatabaseManager.UseTransactions property** allows automatically start/commit of transactions when executing DDL statements.
- **New: TGlobalConfigs.GetInstance.UseTransactionsInDBManager property** provides a global way to control the UseTransactions property in TDatabaseManager.
- **Improved:** OldColumnValues in OnUpdating/OnUpdated events now includes column values for proxies even when they were not yet loaded.
- **Fixed:** Entity generator not working with "INTERBASE" dialect.
- **Fixed:** Entity generator raising an error when trying to extract schema information from PostgreSQL 11.
- **Fixed:** Workaround a bug in Delphi Rio causing error in deserialization using TDataSnapJsonDeserializer.
- **Fixed:** TAureliusConnection and UniDac adapter causing "one of the connections in the transaction is not active" error.
- **Fixed:** It's now possible to have two Aurelius entity class with same name in the same model (e.g, TCity in Unit1 and TCity in Unit2).

Version 4.3 (Dec-2018)

- **Fixed:** Entity generation from databases using TAureliusConnection failing on MySQL 8 with error "table 'mysql.proc' doesnt exist".
- **Fixed:** TAureliusConnection failing to create a cloned connection for ElevateDB connections.

Version 4.2 (Nov-2018)

- **New: Support for Delphi 10.3 Rio.**
- **Improved:** AllButRemove is default option for association cascade type when generating entities from database.
- **Fixed:** DBIndexes not being created together when a new table was created.
- **Fixed:** TDatabaseManager.ValidateDatabase reporting wrong data type for wide memo fields in DB2.

Version 4.1 (Oct-2018)

- **New: TObjectManager.HasChanges allows checking if an specific or all objects in manager have been modified.**
- **New: MSSQL driver LoginTimeout parameter.**
- **Improved:** TAureliusDataset.ForceWideTypes forces dataset to create wide string types (widestring, widememo, widefixedchar) for text-based fields.
- **Improved:** TCriteriaResult objects are now editable from the TAureliusDataset.
- **Improved:** Better error message when trying to use unsupported field/property types in mapped classes.
- **Improved:** TAureliusConnection design-time settings dialog now responds to Enter and Esc keys.
- **Fixed:** Better handling of memo fields in TAureliusDataset - TBlob unicode memo raw data is now converted to ANSI data if field type is ftMemo.
- **Fixed:** TAureliusDataset now creates memo/widememo fields when the TBlob property is flagged with DBTypeMemo/DBTypeWideMemo attributes (previously it was blob).
- **Fixed:** Entity generator now adds DBTypeWideMemo for field types in database that are explicit unicode memo fields (NText, NVarchar(max), etc.).
- **Fixed:** Blob fields not marked as "loaded" when read from AureliusDataset, causing a single lazy blob to be retrieved multiple times when navigating through the dataset.
- **Fixed:** Rare Int64 convert error when importing entities from a MySQL database using a LONGBLOB data type.
- **Fixed:** JwtAuthDemo memory leak when canceling the insertion of a new record.
- **Fixed (4.1.1):** Error when importing Firebird3 boolean fields even using FIREBIRD3 dialect, when using TAureliusConnection "Generate database entities" design-time option.

Version 4.0 (Sep-2018)

- **New: TAureliusConnection component.** This component makes it even easier to connect to a database using TMS Aurelius. It provides design-time configuration and test of database connection, using a connection dialog to visually configure the parameters. Besides supporting the existing component adapters, it also supports the new database native drivers.
- **New: Native database drivers for direct database connection.** Native **Microsoft SQL Server** connection (TMSSQLConnection) is now supported in addition to the existing SQLite driver (with a new TSQLiteConnection). You can now connect directly to SQL Server without the need for a 3rd party component (FireDAC, dbExpress, ADO, etc.), with **increased performance** (at least 20% from initial tests).

```
Conn := TMSSQLConnection.Create(
  'Server=.\SQLEXPRESS;Database=Northwnd;TrustedConnection=True');
```

- **New: Generate TMS Aurelius entities from existing database directly from the IDE.** Thanks to the new TAureliusConnection component, it's now possible to import an existing database structure and generate source code with TMS Aurelius classes mapped to the existing database, with a few clicks.
- **Improved:** Aurelius connection wizard updated to allow choosing the new native drivers.
- **Improved:** Dropped Delphi 2010 and XE support. TMS Aurelius and BCL now supports Delphi XE2 and up.
- **Fixed:** Icon in IDE splash screen not appearing.

Version 3.13 (Jul-2018)

- **New: TAureliusDataset.FieldInclusions property.** This provides more control on what types of fields will be automatically created by Aurelius Dataset. You can choose not to automatically create lists (dataset fields) or objects (entity fields), for example.
- **New: TObjectManager.DeferDestruction property.** Such property prevents immediate destruction of entities removed with Remove method, deferring their destruction to the moment when object manager is destroyed.
- **Improved:** Aurelius DBConnection Wizard using FireDAC now adds FireDac.DApt unit automatically to uses clause.
- **Improved:** TCriteria FindByAlias and FindByPath methods allows finding TSubCriteria objects created using CreateAlias or SubCriteria methods.
- **Improved:** Proxy type now sets internal proxied value to nil when DestroyValue is called.
- **Fixed:** DiscriminatorColumn attribute now ignores size parameter (when updating database schema) if discriminator type is not string.
- **Fixed:** DiscriminatorColumn now has default size of 0 (instead of 30) when DiscriminatorType is dtInteger.
- **Fixed:** Entity classes in a single-table hierarchy without DiscriminatorColumn attribute was causing errors when loading entities. Now such classes are being ignored by Aurelius.
- **Fixed:** Associations/proxies not loading correctly for inherited classes in a single-table hierarchy.
- **Fixed:** TAureliusDataset memory leak when a source (criteria, cursor) is specified but the dataset is never open (3.13.1)

Version 3.12 (May-2018)

- **Improved: Significant performance improvement in entity retrieval.** Up to 50% of speed gain in some operations, most noticeable when selecting (finding) a high number of entities with high number of properties and associations.

- **New: Proxy<T>.Key property.** Allows you to get the database value of foreign key without needing to load the proxy object.

- **Improved:** PostgreSQL generator now supports both Sequence (already supported) and Identity (serial) identifiers. If the Sequence attribute is not defined in the mapping, then it will try to retrieve the id generated by the database (if any).

- **Improved:** SQL Server dialect option: WorkaroundInsertTriggers.

- **Improved:** IDBConnectionAdapter interface allows to get the underlying adapted database-access component (TFDConnection, for example).

- **Fixed:** Access Violation in TAureliusDataset when setting property DatasetField at design-time.

- **Fixed:** ManyValuedAssociation attribute documentation was wrongly explaining the option TCascadeType.Lazy.

Version 3.11 (Feb-2018)

- **New: LINQ SqlFunction and ISQLGenerator.RegisterFunction allows creating custom SQL functions to be used in LINQ.** It's possible to use any database-specific SQL function when using Aurelius LINQ. For example, you could register UNACCENT function from PostgreSQL and use it from Criteria API:

```
TSQLGeneratorRegister.GetInstance.GetGenerator('POSTGRESQL')
  .RegisterFunction('unaccent', TSimpleSQLFunction.Create('unaccent'));

.Where(Linq.ILike(
  Linq.SqlFunction('unaccent', nil, Linq['Name']),
  Linq.SqlFunction('unaccent', nil, Linq.Value<string>(SomeValue))
 ))
```

- **New: ILike operator in LINQ.** You can now also use ILike operator in Linq expressions. It will of course only work on databases that support it:

```
.Where(Linq['Sex'].IsNotNull and Linq['Name'].ILike('M%'))
```

- **New: TCriteria.Find<T>.Open now can be iterated.** ICriteriaCursor now implements GetEnumerator which allows you to iterate easily through the returned entities of a criteria this way:

```
for Customer in Manager.Find<TCustomer>
  .Where(Linq['City'] = 'London').Open do
  { use Customer object here }
```

- **Improved:** More detailed info when exception EAssociationReferencesTransientObject is raised ("Association references a transient object"), indicating now the context: the name of the association property that caused the issue, the id of the object, etc.

- **Improved:** TAbstractSQLGenerator.EnforceAliasMaxLength allows avoiding issues when field names in database are at the maximum size and might cause "field not found" errors when executing LINQ queries. This was more frequent with Firebird databases.

```
(TSQLGeneratorRegister.GetInstance.GetGenerator('Firebird') as TAbstractSQLGenera
tor)
  .EnforceAliasMaxLength := True;
```

- **Improved:** No more UPDATE SQL statements executed when inserting child (many-valued association) items. When inserting an object tree with many-valued associations items (Parent + Child Items), Aurelius was executing INSERT SQL statements for Parent record and for child records, and then after that UPDATE SQL statements were being executed to update the foreign-key field from child to parent table. Now this is optimized and the UPDATE SQL statements are not executed anymore, as the INSERT statements already set the foreign-key of child records.
- **Improved:** TCriteria<T>.Open now returns ICriteriaCursor<T> instead of TCriteriaCursor<T>. This is a minor breaking change.
- **Improved:** TAureliusDataset is not "Sequenced" anymore when RecordCount mode is set to Retrieve. This means that a data control like a grid will show the correct scrollbars (size and position relative to total of records) even when using fetch-on-demand mode and not all entities were retrieved.
- **Fixed:** Aurelius Dataset fields not notifying visual controls when subproperties were being automatically updated due to SyncSubprops behavior.

Version 3.10 (Oct-2017)

- **Improved: Significant performance increase** when retrieving entities from database. The specific scenario is when an entity being retrieved from database is already in the manager. Speed gains are more noticeable when lots of associated entities are retrieved in eager mode and have same id, and when cached entities have many mapped properties.
- **New: TAureliusDataset.RecordCountMode property.** When using dataset in paged mode, you can ask dataset to perform an extra statement in the database to grab the total number of records in advance and return it in RecordCount property, even before all pages are fetched.
- **Fixed:** SQLite driver refactored to use static library on Android due to Android 7 Nougat error: "unauthorized access to "libsqlite.so".
- **Fixed:** Design-time wizard icon not showing correctly in Delphi 10.2 Tokyo.
- **Fixed:** TCriteria.Refreshing state was lost when TCriteria was cloned.

Version 3.9 (Jul-2017)

- **New: TCriteria.Refreshing method.** Using Refreshing method when creating an Aurelius query will force entities returned by the query to be refreshed even if they are already cached in Object Manager.

- **New: DBIndex attribute.** In addition to unique indexes, you can now specify non-unique index (for optimization purposes) with this attribute and Aurelius will create it automatically upon database schema update.

- **New: TAureliusDataset.SyncSubprops property** allows automatic update of associated fields. When an entity field (e.g., "Customer") of the TAureliusDataset component is modified, all the subproperty fields (e.g., "Customer.Name", "Customer.Birthday") will be automatically updated with new values if this property is set to True.

- **New: TAureliusDataset.SubpropsDepth property** allows automatic loading of subproperty fields. When loading field definitions for TAureliusDataset at design-time, or when opening the TAureliusDataset without persistent fields, one TField for each property in object will be created. By increasing SubpropsDepth to 1 or more, TAureliusDataset will also automatically include subproperty fields for each property in each association, up to the level indicated by SubpropsDepth.

- **New: TAureliusDataset.DefaultsFromObject property** brings field default values with object state. When inserting a new record in TAureliusDataset, all fields come with null values by default. By setting this property to True, default (initial) value of the fields will come from the property values of the underlying object.

- **New: TObjectManager.FindCached and IsCached methods.** Those methods allow checking if an object of specified class and id is present in the object manager cache, without hitting the database to load the object.

- **New: TAureliusDataset popup menu option at design-time for quick reloading field definitions.** At design-time, if you right-click TAureliusDataset component, a new menu "Reload from <class>" appear for quickly reloading the field definitions for a previously loaded class.

- **Improved:** Faster lazy-loading of proxied associations in some situations. When the association has a JoinColumn attribute with a explicity param value for ReferencedColumnName, the manager was always hitting the database to load associated proxy. Now if the referenced column is an id column, the manager will first check if associated object is already in cache.

- **Improved:** TAureliusDataset doesn't automatically call Flush anymore on Insert and Delete operations, when Manager property is set. Only Save and Remove methods are called, respectively. This fixes performance and unexpected behaviors in some scenarios, but might break existing code. It's a breaking change.

- **Improved:** When targeting DB2 databases, TDatabaseManager now retrieves schema of database objects and updates/creates them accordingly.

- **Improved:** Updating ElevateDB database schema (TDatabaseManager.UpdateDatabase) is significantly faster now.

- **Fixed:** Calling TAureliusDataset.Delete was raising an exception in some specific situations.
- **Fixed:** Argument out of range on specific Merge operations. This error was happening when merging an object A with a proxied list of B objects. If the B objects happen to have a reference back to A, then another instance of A would be loaded, the proxied list would be loaded, and such list would override the list of original object A being merged, causing this error.
- **Fixed:** Firedac + Oracle on Delphi Tokyo was causing "Data Too Large" error on fixed-sized parameters.
- **Fixed:** Calling TAureliusDataset.RecordCount on a closed dataset was raising an Access Violation.

Previous Versions

Version 3.8 (May-2017)

- Fixed: Using AureliusDataset, during an insert, if a Post operation failed, an Access Violation would be raised if user cancels insertion of record.
- Fixed: Access Violation when loading a lazy blob in the handler of OnDeleted event.

Version 3.7 (Mar-2017)

- New: Linux platform support together with Rad Studio 10.2 Tokyo support.
- Fixed: Memory leaks in mobile platforms.
- Fixed: Error when loading entities with inheritance where a lazy blob field is declared in an inherited class.
- Fixed: TGlobalConfigs.GetInstance.SimuleStatements not working.
- Fixed: Better transactions handling on UIB (Universal Interbase) driver.

Version 3.6 (Feb-2017)

- New: Manager events OnInserting, OnUpdating, OnDeleting.
- Improved: Not equal (<>) operator support in Linq queries.
- Fixed: Firebird schema update was trying to generate sequences even though they already existed in database (regression).
- Fixed: Error inserting records in SQL Server when table name ends with "Values".
- Fixed: JSON Deserializer failed when deserializing nullable enumerated values.
- Fixed: DB2 dialect was not supporting schemas (regression).

Version 3.5 (Jan-2017)

- New: Firebird3 dialect support.
- New: MSSQL dialect UseBoolean property allows using BIT data type for boolean fields in SQL Server.
- Improved: Column names can now be mapped using double quotes.
- Improved: Demos rewritten to better show use more recent Aurelius features.
- Improved: Better error handling when SQLite DLL is not available.
- Fixed: Error with field names containing spaces.
- Fixed: Wrong behavior and cast errors in TAureliusDataset when moving dbgrid field columns linked to the dataset.
- Fixed: Cast error in Aurelius Dataset when setting a nullable enumerated field to null.
- Fixed: Aurelius Dataset Locate method accepts variant array as search value even when locating for a single field.
- Fixed: IBExpress adapter not working if using the overloaded Create constructor that receives a TComponent parameter.
- Fixed: Memory leaks on nextgen (mobile) platforms when using FireDac (version 3.4.1)

Version 3.4 (Sep-2016)

- New: Linq query syntax improved with support for relational operators: *Linq['Name'] = 'Mia'*. All query examples in this documentation updated to newer syntax.
- New: Arithmetic projections Add, Subtract, Multiply and Divide, also supporting operators: *Linq['Total'] + Linq['Additional']*.
- New: In clause in Linq queries.
- New: Linq "type-helper" version all existing functions, like Upper or Year: *(Linq['Name'].Upper = 'MIA') and (Linq['CreatedAt'].Year = 2015)*.
- New: Cross-database Concat function: *Linq.Concat(Linq['FirstName'], Linq['LastName'])*.
- New: Linq functions Contains, StartsWith, EndsWidth now support projections: *Linq['Name'].StartsWith(Linq['OtherField'])*.
- New: TDatabaseManager.IgnoreConstraintName property for better control of database schema update and validation.
- Fixed: ZeosLib depending on unnecessary units.

Version 3.3 (Aug-2016)

- New: TObjectManager.Flush method can now receive an entity as parameter allowing flushing a single entity.
- New: Support for ZeosLib database-access components.

- New: TCascadeType.Flush cascade type allows control of how associated objects will be flushed when flushing a single entity.
- Improved: When retrieving Int64 values from database, it now tries to handle the value even when the underlying db access component provides the value as float.
- Fixed: TAureliusDataset.RecNo returning wrong value when in insert mode.
- Fixed: When using bidirectional associations, in some rare situations the many-to-one side of association was being cleared.
- Fixed: TAureliusDataset displaying wrong records when using Filter in a detail dataset (DatasetField pointing to another dataset).

Version 3.2 (Jul-2016)

- New: TCriteria.Clone method allows cloning an existing Aurelius criteria.
- New: TAureliusDataset.IncludeUnmappedObjects property to allow object and list fields even if they are not mapped in class.
- New: TManagerEvents.OnSQLExecuting event that is fired for every SQL statement executed in database.
- Improved: Mapping table and field names with spaces is now allowed, without needing to quote the names in quotes in mapping.
- Improved: Online Resources updated with links for new videos and articles.
- Fixed: **Breaking change:** Merging transient objects with proxy collections was ignoring the collection content. TObjectManager.MergeListLegacyBehavior.
- Fixed: **Breaking change:** Updating/Merging objects with proxied associations that were not modified was not clearing the value.
- Fixed: "Duplicate Field Name" error in Aurelius Dataset when loaded object had properties that have been redeclared from an ancestor class.
- Fixed: Inheritance using discriminator failed in some situations with SQLite due to int32/int64 type mismatch.
- Fixed: DB Connection Wizard failed when using AnyDac connection.
- Fixed: TProjections.Count failed for counting GUID fields.
- Fixed: TDateTime field values losing time part when using dbGO and ODBC driver.

Version 3.1 (May-2016)

- New: Delphi 10.1 Berlin support.
- New: Explorer.ObjectFactory and Manager.ObjectFactory properties allows defing a custom object factory for creating entity classes.
- Fixed: Database update using table schema now working with PostgreSQL and MS SQL Server.

Version 3.0 (Feb-2016)

- New: Design-time wizard "New TMS Aurelius Connection" makes it very straightforward to create Aurelius database connections (IDBConnection).
- New: TObjectManager.Replicate method.
- Improved: Automapping now sets generator to SmartGuid if field FId is of type TGuid.
- Improved: TObjectManager.Find has a new overload that accepts TGuid value for id.
- Improved: Saving an object with user-assigned id was calling SQL to retrieve ID without need.
- Improved: TDatabaseManager can receive a TArray<TMappingExplorer>, allowing to create the database structure for all of them at once.
- Fixed: Merging an object with a lazy-loaded list wouldn't delete removed items on Flush if the object being merged was not loaded from TObjectManager.
- Fixed: After Mapping Explorer raised an error about wrong mapping when retrieving columns for a class, it could later not raise that error anymore.
- Fixed: Wrong error message (AV) when opening a cursor and SQL dialect is not registered.
- Fixed: Sporadic AV when destroying TAureliusDataset without closing it.

Version 2.9 (Oct-2015)

- New: Optimistic versioned concurrency control of entities using Version attribute.
- New: TObjectManager.UseTransactions property allows control whether manager uses transactions to perform internal operations. This is a **breaking change**.
- Improved: More detailed error message when loading a proxy fails due to duplicated records.

Version 2.8.1 (Sep-2015)

- New: Delphi 10 Seattle support.

Version 2.8 (Aug-2015)

- New: Cross-database, high-level projection functions in Aurelius queries. Date/time functions added: Year, Month, Day, Hour, Minute, Second. String functions added: Upper, Lower, Substring, Position, Length, ByteLength.
- New: Additional TLinq conditions for string comparison: Contains, StartsWith, EndsWith.
- New: OnInserted event parameters now include Master that hold the parent instance in case of unidirectional items being inserted.

Version 2.7.1 (May-2015)

- Fixed: AV when using Update event listener for objects in manager without previous state (using Update method).

Version 2.7 (Apr-2015)

- New: Events system allows subscribing listeners to respond to several events (e.g, when an entity is inserted, updated, etc.).
- Improved: When deserializing objects from JSON, properties unknown to the entity will now be ignored, instead of raising an error.
- Improved: Music Library demo includes an audit log viewer that illustrates usage of the events system.
- Fixed: FireDAC driver not compiling on XE8.

Version 2.6.3 (Apr-2015)

- New: Delphi XE8 support.

Version 2.6.2 (Mar-2015)

- Improved: TBlob handling of data (especially using AsBytes property) improved for better performance.
- Improved: TBlob.Data property removed. Breaking change.
- Fixed: Flush not updating properties modified if lazy proxy/blob is loaded after properties were modified.
- Fixed: Setting a lazy TBlob content that was not yet loaded didn't change blob content.
- Fixed: TAureliusDataset now retrieves correct value for RecordCount when dataset is filtered.
- Fixed: Rare Access Violation when reloading associated object lists that exist in object manager.

Version 2.6.1 (Feb-2015)

- Improved: TAureliusDataset design-time dialog now makes it much easier to find a class by providing a search box.
- Improved: TAureliusDataset makes it easy to reload fields from classes at design-time by remembering the last class used to load fields.
- Fixed: TObjectManager.Merge was not updating collections when none of parent object properties was changed.
- Fixed: AV when loading a proxy value after an object refresh.

- Fixed: Error when inserting records with identity Id on tables with INSERT triggers in MS SQL Server.
- Fixed: Access Violation when destroying entity objects before destroying a TAureliusDataset component.
- Fixed: Rare error when inserting records in MS SQL Server, using SQL-Direct and native SQL Server client.

Version 2.6 (Dec-2014)

- New: TObjectManager.Evict method allows removing an object instance from the manager without destroying it.
- New: TFetchMode option in CreateAlias allows per-query setting for eager-loading associations to improve performance.
- New: TAureliusDataset.Current now returns an object even in insert state.
- New: TAureliusDataset.ParentManager allows fine-grained control over the manager used in detail datasets.
- New: TCriteria.OrderBy provides an easier, alternative way to TCriteria.AddOrder to specify criteria order.
- Improved: Automatic destruction of TCriteriaResult objects in TAureliusDataset when using SetSourceCriteria or SetSourceCursor.
- Improved: Removed an extra final SQL being executed in paged queries using TAureliusDataset.
- Fixed: Design-time error using TAureliusDataset when recompiling packages with entities.
- Fixed: TAureliusDataset.BookmarkValid was wrongly returning true after the bookmarked record was deleted.
- Fixed: Blobs and associations being loaded in lazy mode were causing objects to be updated on flush.
- Fixed: Json serialization using SuperObject was providing wrong boolean value.
- Fixed: Saving child objects using unidirectional ManyValuedAssociation when parent has composite key.

Version 2.5 (Oct-2014)

- New: Multi-model design architecture allows different mapping models in a single application with a few lines of code, just by using attributes.
- New: SmartGuid generator allows using identifiers with sequential GUID for better database performance.
- New: OrderBy attribute allows defining a default order for many-valued associations.
- New: Model attribute to specify the model where the class belongs to.

- New: RegisterEntity procedure helps registering a mapped class avoiding linker optimization to remove it from application.
- New: Proxy<T>.Available property.
- Improved: More detailed manager error messages when trying to save objects that are already persistent.
- Fixed: Identity conflict when using MS SQL Server with multiple simultaneous sessions inserting in the same table.
- Fixed: Trailing semi-comma from some PostgreSQL commands were causing errors when using FireDac with automatic record count.
- Fixed: Wrong data for fields OldValue property when dataset is empty.
- Fixed: Incompatibility between TAureliusDataset and FastReport design-time editor.

Version 2.4.1 (Sep-2014)

- New: Delphi XE7 support.

Version 2.4 (Jul-2014)

- New: TObjectManager.Refresh method allows refreshing object state from database.
- New: ForeignKey attribute to define the name of foreign keys in the database.
- New: TCascadeType.RemoveOrphans allow automatic deletion/removal of child entities on Flush if they are removed from a parent collection.
- New: TCustomJsonDeserializer.Entities property allows retrieving the list of objects created by the JSON deserializer.
- New: TDriverConnectionAdapter<T>.Connection property allows referencing the original database component used for the connection.
- New: TBlob.Available property.
- New: TFirebirdSQLGenerator.WideStringCharSet property allows defining specific column character set for WideString properties in Firebird.
- Improved: Merge now can receive objects with no id. This will automatically create a copy of the object and save it. This is a breaking change.
- Improved: Better performance and memory consumption using unidirectional datasets to fetch data with some specific component adapters.
- Fixed: Error when updating objects with composite id in SQLite and one of id values is null.
- Fixed: Error when serializing a newly created entity (not loaded with manager) with a TBlob property that has not been initialized.
- Fixed: ElevateDB driver compile error when using latest ElevateDB versions.
- Fixed: Error when deserializing empty dynamic array properties.

Version 2.3.1 (Apr-2014)

- New: Delphi XE6 Support.
- Improved: MappedClasses.RegisterClass now checks if the class being registered is a valid entity ([Entity] attribute present).
- Improved: CascadeTypeAllButRemove constant makes easier to define association cascade with all options except TCascadeType.Remove.
- Fixed: Using [Automapping] attribute with classes that inherit from non-entity classes was causing "Id attribute not found" error.
- Fixed: Wrong TAureliusDataset behavior with db visual controls that rely on CompareBookmarks method.

Version 2.3 (Feb-2014)

- New: Support for Android platform.
- New: Support for FireDac components.
- New: Overloaded constructor for connection component adapters allows easier memory management when using data modules.
- Improved: Property TIBObjectsConnectionAdapter.Transaction allows you to change the default transaction in an IBObjects connection adapter.
- Fixed: TAureliusDataset.Current method was returning an invalid value when it was in insert state.
- Fixed: "Duplicates not allowed" when retrieving objects in a inheritance tree where different descendant classes had associations with same name.
- Fixed: TAureliusDataset missing the current record position in some situations.
- Fixed: Memory leak when trying to save unmapped objects.

Version 2.2 (Oct-2013)

- New: Increased querying capabilities with new TExpression/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.

Licensing and Copyright Notice

Licensing Information

Trial version of this product 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/distribution of the application.

For registered version of this proudct, three types of licenses apply:

- Single Developer License
- Small Team License
- Site License

Main Copyright

Unless in the parts specifically mentioned below, all files in this distribution are copyright (c) Wagner Landgraf and licensed under the terms detailed in Licensing Information section above. The product 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.

Third Party Copyrights

This distribution might also contain the following licensed code:

File Bcl.Collections: Hash Set data structure.

Copyright (c) 2017 by Grijjy, Inc.
Those parts are licensed under the following terms:

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
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.

Breaking Changes

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

Version 4.18

If you use undocumented internal classes *TInsertCommand* or *TUpdateCommand*, be aware that the type of the insert and update fields have changed. You now have to create a *TInsertSQLField* and *TUpdateSQLField* respectively, and you must define a value for the *ParamName* property.

Old behavior:

```
Command := TInsertCommand.Create;
Command.InsertFields.Add(TSQLField.Create(SQLTable, 'Name'));
```

New behavior:

```
Command := TInsertCommand.Create;
Command.InsertFields.Add(TSQLInsertField.Create(SQLTable, 'Name', 'ParamName'));
```

The same applies for TUpdateCommand and TSQLUpdateField.

Version 3.11

TCriteria.Open<T> now returns ICriteriaCursor<T>.

This is a change that should not affect any existing code. But in any case you have a type mismatch error when retrieving a cursor with *Open* and saving the reference to a variable, just change the type of the variable and everything should work as expected.

Version 3.2

- Merging transient objects with proxy collections was ignoring the collection content (*TObjectManager.MergeListLegacyBehavior*).
- Updating/Merging objects with proxied associations that were not modified was not clearing the value.

More info here.

Version 2.9

- Object manager now uses transactions by default. More info here.

Version 2.4

The process of merging objects (*Merge* method) has improved, but this created a breaking change. In previous versions, if you tried to merge an object without id, an exception would be raised. But if you tried to merge an object which had an association that pointed to an object with no id, nothing would happen and that association property would remain unchanged. It was an inconsistent behavior but no exception was raised. Starting from version 2.4, if you try to merge an object with no id, a copy of that instance will be saved. If it's an association, the instance will be replaced. This is a breaking change.

For example, consider the following code:

```
Customer := TCustomer.Create;
Customer.Id := 1;
Country := TCountry.Create;
Country.Name := 'New Country';
Customer.Country := Country;
MergedCustomer := Manager.Merge<TCustomer>(Customer);
```

Customer has an id but *Country* has not. Customer will be merged and a different instance will be returned and put in *MergedCustomer* variable. Previous to version 2.4, MergedCustomer.Country will point to the same instance pointed by Country variable, and nothing would happen in database. From version 2.4 and on, a copy of Country object will be saved in database, and MergedCustomer.Country will point to that new instance, which is different from the instanced referenced by Country variable. You should destroy the Country instance.

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`).

Version 3.2 - Breaking Changes

Merging transient objects with proxy collections was ignoring the collection content.

This versions fixes a bug that might break existing code that was relying on such bug to work.

Suppose you have a list with a property using lazy-loaded association (using *Proxy*):

```
TCustomer = class
{...}
  FAddresses: Proxy<List<TAddress>>;
```

If you initialize such class and *Merge* it using an existing customer Id:

```
Customer := TCustomer.Create;
Customer.Id := 5;
Manager.Merge<TCustomer>(Customer);
Manager.Flush;
```

Expected behavior would be that all the existing Addresses associated with Customer which Id=5 would be disassociated from it (or deleted if the association cascade included RemoveOrphan type.

However, for versions below 3.2, the property was being ignored when merging and the addresses were kept. So you must be sure that your code doesn't rely on such behavior, otherwise you might get some changes in data.

If you want to keep the old behavior, you can set a specific property in the object manager:

```
Manager.MergeListLegacyBehavior := True;
```

This will keep the old (and wrong) behavior.

Updating/Merging objects with proxied associations that were not modified was not clearing the value.

On the other hand, suppose you have the same situation but with a single entity association:

```
TCustomer = class
{...}
  FCountry: Proxy<TCountry>;
```

If you create a new instance and update (or merge) it, leaving Country blank:

```
Customer := TCustomer.Create;
Customer.Id := 5;
Manager.Update(Customer); // or Manager.Merge<TCustomer>(Customer);
Manager.Flush;
```

Expected behavior would be that Country of customer with id = 5 in the database would be cleared.

However, for versions below 3.2, the value was being ignored and Country property was left unchanged. So be careful with the update because after updating existing code might behave differently (even though it was relying on a bug).

Version 2.9 - TObjectManager.UseTransactions

As of version 2.9, *TObjectManager* includes a property UseTransactions. This property is **true** by default, meaning the behavior is different from previous versions. When true, the manager will create transactions for its internal operations (for example, when you call *Save* or *Remove*). This is to make sure that all SQL performed by the internal operations are executed successfully or all is reverted in case of error at any point.

In our (huge) test suite, we didn't detect any problem with backward compatibility, no regressions. But in any case you find an issue with version 2.9, please be aware of this change and consider if that can be the cause of the problem.

You can switch to previous behavior by setting that property to false, or globally using the global configuration.

Online Resources

This topic lists some links to internet resources - videos, articles, blog posts - about TMS Aurelius.

Official Online Documentation

Intensive Delphi video series
(Portuguese Audio, English Subtitles)

- Introduction to TMS Software products: TMS Business, Aurelius, XData, Scripter
- TMS Aurelius ORM for Delphi: Basic Demo Showcase
- TMS Aurelius ORM for Delphi: Music Library Demo Showcase
- TMS Data Modeler Database Modeling integrated with TMS Aurelius Delphi ORM
- TMS XData Showcase: REST/JSON server for Delphi from scratch
- TMS XData for Delphi: Features of Rest/JSON Server, filter, orderby, PUT, POST
- TMS Scripter: Add scripting capabilities to your Delphi application with full IDE/debugging support

Rest/Json Server On Linux with TMS XData and Delphi - video series
(English Subtitles)

- Part 1: Installing Ubuntu Linx
- Part 2: Installing PAServer
- Part 3: WebBroker Apache Module
- Part 4: TMS Sparkle with Apache
- Part 5: TMS XData/Aurelius Server

"My Top 10 Aurelius Features" blog post and video series
(videos have both English and Portuguese subtitles)

- Introduction (05-Dec-2016)
- #10 - Automapping (video link) (05-Dec-2016)
- #9 - Plain Old Delphi Objects (video link) (12-Dec-2016)
- #8 - Lazy Loading (video link) (22-Dec-2017)
- #7 - Schema Update (video link) (03-Jan-2017)
- #6 - Legacy Databases (video link) (12-Jan-2017)
- #5 - LINQ Expressions and Paging (video link) (30-Jan-2017)
- #4 - Aurelius Dataset (video link) (09-Feb-2017)
- #3 - Inheritance (video link) (21-02-2017)

- #2 - LINQ Projections (video link) (06-03-2017)
- #1 - Maturity (video link) (19-03-2017)

Malcolm Groves' series of articles "Storing your Objects in a Database" about TMS Aurelius (include videos)

- Introduction (16-Jun-2016)
- Getting Started (16-Jun-2016)
- Extending the Model (11-Jul-2016)

Aurelius Crash Course (blog posts)

- Getting Started
- AnyDAC or dbExpress
- Associations (Foreign Keys)
- Using Blobs
- Inheritance and Polymorphism
- Visual Data Binding using TAureliusDataset

Conference/Webinar Videos

- TMS Aurelius Free Edition - An Overview (CodeRage XI Session) (21-Nov-2016)
- TMS Aurelius session at CodeRage 8 (download source code used in video)
- Introducing TMS Aurelius, a Delphi ORM - Vendor Showcase

Portuguese Resources - Links em português

Vídeos em português

- TMS Aurelius - Usando TAureliusDataset
- TMS Aurelius - Criando uma Aplicação
- Grupo DCORM - reunião sobre TMS Aurelius/XData
- TMS Aurelius e TMS XData - DCORM group meeting - 2014 (português) (download source code)
- TMS Aurelius e TMS XData - Embarcadero Conference 2013 (português) (download source code)

Curso Rápido TMS Aurelius (português)

- Primeiros Passos
- FireDac ou dbExpress?
- Associações (Chaves Estrangeiras)

Artigos em Revistas

- Artigo revista DevMedia - Mapeamento ORM com TMS Aurelius