Object-Relational Mapping using TMS Aurelius

Developing database applications is a common use for Delphi. However, dealing with SQL statements is typically a boring, cumbersome and error-prone task. Moreover, relying on SQL statements in your code directly can quickly make your application heavily dependent on a specific database. In general, when building database applications, steps are roughly the same:

1. Configure a connection
2. Create a TSQLQuery or any other TDataset descendant that executes SQL statements
3. Build the SQL statement and execute it with TSQLQuery
4. Open the TDataset and iterate through the records reading values using TField objects.

It's been like this since the beginning of Delphi and obviously it is a well-known method – but it has its disadvantages. We have to manually build the SQL statement and remember the names of fields and tables in the database, either when building the SQL or when reading field values. If we do a database refactor, we have to review all our SQL statements.

As an alternative, we can use an object-relational mapping framework. With such library, the process is different: we do an initial mapping, to tell the framework how each object will be saved in the database, and from there, we do not work with SQL or TDataset anymore: we just deal with the objects.

The purpose of this article is to show how to use an object-relational mapping framework to build database applications. Framework used will be TMS Aurelius (http://www.tmssoftware.com/site/aurelius.asp).

Creating a connection

First thing to do is to configure a database connection. TMS Aurelius doesn't require you to learn too much new in this step – actually it doesn't even provide components for database connection: it just uses what you have in your application. Instead, TMS Aurelius provides adapters which you use just by passing your existing database connection component to it. Listing 1 displays how to obtain an Aurelius connection from a dbExpress connect ion.

```
Listing 1 – Creating a connection
uses
  Aurelius.Drivers.Interfaces, Aurelius.Drivers.dbExpress;

var
  Conn: IDBConnection;
begin
  Conn := TDbExpressConnectionAdapter.Create(SQLConnection1, False);
```

```
end;
```

SQLConnection1 component is a dbExpress TSQLConnection component that might be already present in your application (or you could just have dropped it to use exclusively with Aurelius). It's in that component that you will configure your connection, and to use it with Aurelius, just use the adapter to obtain an IDBConnection interface. The Conn variable created in Listing 1 will be used in all examples from now on.

There are adapters for several component libraries. If you don't use dbExpress, you can use native dbGo/ADO (TADOConnection), IBExpress (TIBDatabase), native SQLite connection, or even 3rd parties like UniDac (TUNIConnection), AnyDac (TADConnection), SQL-Direct (TSDDatabase), NexusDB, ElevateDB, and several others.

TMS Aurelius will also create SQL statements according to the database to which the application is connecting to. In the example above, when using a dbExpress adapter, TMS Aurelius automatically detected the target database by reading TSQLConnection.DriverName property, but you can explicitly inform which database server you are connecting to. Options are many:

Oracle, SQL Server, Interbase, Firebird, MySQL, Nexus, DB2, SQLite, PostgreSQL, ElevateDB, etc.

Class Mapping

You can use an existing class or create a new one. TMS Aurelius has an auto mapping feature, which makes class mapping really easy. Take a look at Listing 2.

```
Listing 2 – Automatic class mapping
  [Entity]
  [Automapping]
  TPerson = class
  private
    FId: integer;
    FName: string;
    FCity: string;
    FPhone: string;
  public
    property Id: integer read FId;
    property Name: string read FName write FName;
    property City: string read FCity write FCity;
    property Phone: string read FPhone write FPhone;
  end;
```

On Listing 2, we're telling TMS Aurelius: this class can be persisted in database (Entity) and an automatic mapping of fields must be done (Automapping). TMS Aurelius will save objects of this class in table “PERSON”, which will have four columns: Id, Name, City and Phone. Id will be an INTEGER type, and remaining columns will be of type VARCHAR.

You can define all of this manually, using attributes. It can be useful if you have an existing application, with database tables already defined, and you just want to start a new module in your application using TMS Aurelius. In this case you create your class the way you want, and map tables and columns manually, as illustrated in Listing 3.

```
Listing 3 – Manual class mapping
  [Entity]
  [Automapping]
  [Table(‘TBL_PEOPLE’)]
  [Sequence(‘SEQ_PEOPLE’)]
  [Id('FId', TIdGenerator.IdentityOrSequence)]
  TPerson = class
  Private
    [Column('PS_ID', [TColumnProp.Unique, TColumnProp.Required,
TColumnProp.NoUpdate])]
    FId: integer;
    [Column('PS_PERSONNAME_A', [TColumnProp.Required], 100)]
    FName: string;
    [Column('PS_CITY_A', [TColumnProp.Required], 50)]
    FCidade: string;
    [Column('PS_PHONE_A', [TColumnProp.Required], 15)]
    FPhone: string;
  public
    property Id: integer read FId;
    property Name: string read FName write FName;
    property City: string read FCidade write FCidade;
    property Phone: string read FPhone write FPhone;
  end;
```

Creating the database – Optional

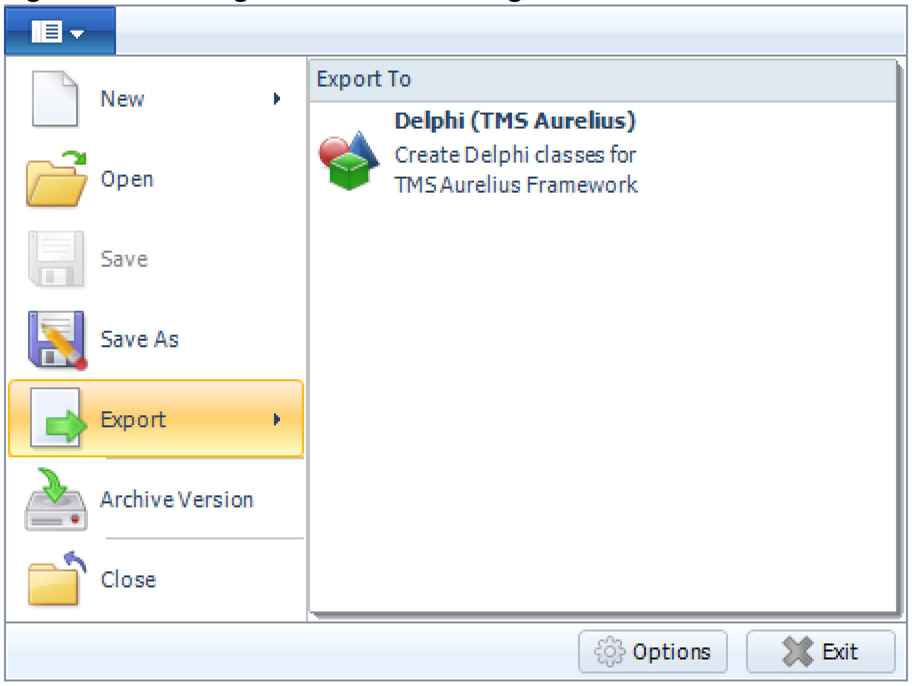
You can create database tables and columns automatically, based on the classes you have defined and mapped. Obviously, if you already have an existing database, you don’t need to do this step. Listing 4 shows how to create database scheme.

```
Listing 4 – Creating the database scheme
var
  DatabaseManager: TDatabaseManager;
begin
  DatabaseManager := TDatabaseManager.Create(Conn);
  try
    DatabaseManager.BuildDatabase;
  finally
    DatabaseManager.Free;
  end;
end;
```

Generating classes from database

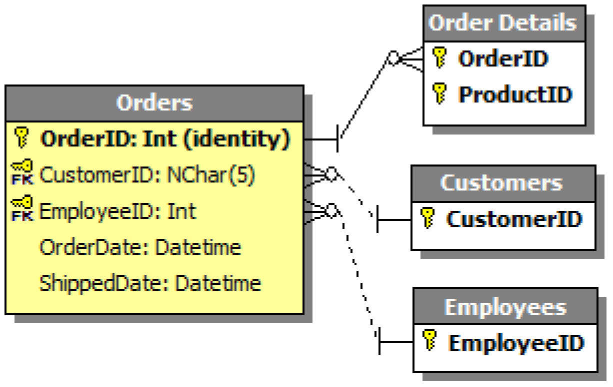
As we mentioned, if the database already exists, you can just create the classes and map them to existing tables and columns. To make this job easier, TMS Aurelius works integrated with TMS Data Modeler. This tool allows you to import the existing database scheme and, besides modeling features are scheme compare, SQL script for generate and update scheme etc., it also provides you an option to automatically generate a Delphi source code with all classes already ready for TMS Aurelius, mapped to existing tables and columns.

Figure 1 – Creating classes from existing database scheme



Class generation is very flexible – you can generate classes right away from default settings, or you can use the configuration dialog to define a specific mapping – which class will be created, what will be its name, which properties, etc. Figure 2 shows a subset of the database model imported by TMS Data Modeler, and Listing 5 displays a class that was automatically generated, mapped to table Orders.

Figure 2 – Existing tables in database



Listing 5 – Mapped class, generated automatically by TMS Data Modeler

```
  [Entity]
  [Table('Orders')]
  [Id('FOrderID', TIdGenerator.IdentityOrSequence)]
  TOrder = class
  private
    [Column('OrderID', [TColumnProp.Required, TColumnProp.NoInsert,
TColumnProp.NoUpdate])]
    FOrderID: integer;
    [Column('OrderDate', [])]
    FOrderDate: Nullable<TDateTime>;
    [Column('ShippedDate', [])]
    FShippedDate: Nullable<TDateTime>;
    [Association]
    [JoinColumn('CustomerID', [], 'CustomerID')]
    FCustomer: Proxy<TCustomer>;
    [Association]
    [JoinColumn('EmployeeID', [], 'EmployeeID')]
    FEmployee: Proxy<TEmployee>;
    [ManyValuedAssociation([], [TCascadeType.SaveUpdate, TCascadeType.Merge],
'FOrderID')]
    FOrderDetailsList: Proxy<TList<TOrderDetails>>;
    function GetCustomer: TCustomer;
    procedure SetCustomerID(const Value: TCustomers);
    function GetEmployee: TEmployee;
    procedure SetEmployee(const Value: TEmployee);
    function GetOrderDetailsList: TList<TOrderDetails>;
  public
    constructor Create;
    destructor Destroy; override;
    property OrderID: integer read FOrderID;
    property OrderDate: Nullable<TDateTime> read FOrderDate write FOrderDate;
    property ShippedDate: Nullable<TDateTime> read FShippedDate write FShippedDate;
    property Customer: TCustomer read GetCustomer write SetCustomer;
    property Employee: TEmployee read GetEmployee write SetEmployee;
    property OrderDetailsList: TList<TOrderDetails> read GetOrderDetailsList;
  end;
```

Persisting objects in database

Once the connection is defined, mapping is done, and tables are ready in database, it's time to effectively build the application. As mentioned before, from now on, the whole application will be object-based. Listing 6 shows how to create a new person in database.

```
Listing 6 – Saving an object
var
  Person: TPerson;
  Manager: TObjectManager;
begin
  Person := TPerson.Create;
  Person.Name := 'Oswald';
  Person.City := 'New York';
  Person.Phone := '(333) 222-2222';
  Manager := TObjectManager.Create(Conn);
  try
    Manager.Save(Person);
    Manager.Flush;
    IdPerson := Person.Id;
  finally
    Manager.Free;
  end;
end;
```

Until the line where TObjectManager is instantiated, code is just plain old Delphi code: just objects manipulation. Then we create a manager, the TObjectManager, to save this object in database. Note that we pass to the manager the Conn interface we have created previously.

Method Save tells we are persisting a new object, and method Flush tells the manager to effectively perform in database all changes we've made (in this case, a new object saving). Aurelius will then build the SQL statement, set its parameters according to the object properties, and execute an INSERT in table. If you are using an Id of type IdentityOrSequence, you can retrieve the generated value by reading the property Id.

Another important observation: note that we didn't destroy object Person. This is not a memory leak – once the object becomes manage by the TObjectManager (in this case, when we call Save method), it will be destroyed automatically when the instance is not needed anymore. This way we can safely use the objects, including its associations, without worrying about memory management and destroying them.

Updating objects

To update an existing object, just find it, change its properties, and call Flush method again. Listing 7 shows how to change Phone property of the Person object we have just created.

```
Listing 7 – Updating objects
var
```

```
  Manager: TObjectManager;
  Person: TPerson;
begin
  Manager := TObjectManager.Create(Conn);
  try
    Person := Manager.Find<TPerson>(IdPerson);
    Person.Phone := '(444) 555-2222';
    Manager.Flush;
  finally
    Manager.Free;
  end;
end;
```

It's interesting to note that TMS Aurelius detects the properties that have been changed (in this case, Phone) and executes the UPDATE statement to update just this field. This helps in multiuser environments and prevents one user to overwrite what another user has changed. The SQL statement to be executed will be the following one (considering the manual mapping of Listing 2):

UPDATE TBL_PEOPLE SET PS_PHONE_A=:p0 WHERE PS_ID=:p1;

Inheritance

Another very interesting feature in TMS Aurelius is inheritance support. As we are working with classes and objects, it makes sense that inheritance, a basic feature of object-oriented programming, can be used. For example, we can create the following new classes (Listing 8):

```
Listing 8 – Inherited classes
[Entity]
[Automapping]
[Inheritance(TinheritanceStrategy.JoinedTables)]
[Table('TBL_INDIVIDUAL')]
TIndividual = class(TPerson)
private
  FSocialSecurity: string;
public
  property SocialSecurity: string read FSocialSecurity write FSocialSecurity;
end;

[Entity]
[Automapping]
[Inheritance(TinheritanceStrategy.JoinedTables)]
[Table('TBL_COMPANY')]
TCompany = class(TPerson)
private
  FOfficialName: string;
public
  property OfficialName: string read FOfficialName write FOfficialName;
end;
```

In this case, TMS Aurelius will create two new tables, TBL_INDIVIDUAL and TBL_COMPANY, both related to table TBL_PERSON through a foreign key. INDIVIDUAL table will have an additional field SocialSecurity, and Company table will have OfficialName. It's possible to configure TMS Aurelius to save all classes in same table, instead of using different tables. Listing 9 shows how to save a class TCompany.

```
Listing 9 – Saving an inherited class
var
  Person: TCompany;
  Manager: TObjectManager;
begin
  Person := TCompany.Create;
  Person.Name := 'Embarcadero';
  Person.City := 'Scotts Valley';
  Person.Phone := '(111) 111-1111';
  Person.OfficialName := ‘Embarcadero Technologies, Inc.’;
  Manager := TObjectManager.Create(Conn);
  try
    Manager.Save(Person);
    Manager.Flush;
    IdCompany := Person.Id;
  finally
    Manager.Free;
  end;
end;
```

Any difference when comparing to Listing 6. None. Only difference is that we are dealing with another class, and thus we have different properties – in this case, property OfficialName. TMS Aurelius will make sure to build all needed SQL statements. In this case, it will execute an INSERT in table TBL_PERSON, saving fields Name, City and Phone, obtain the Id from this newly inserted record, and make another INSERT in table TBL_COMPANY, saving field OfficialName, and also the generated Id, to specify the foreign key with table TBL_PERSON.

One interesting thing about this approach is that when developing the application, everything becomes easy – we can organize classes and objects according to our business logic and all complex SQL statement generation will be in charge of the framework.

Associations

Every database has relationships and so we should have in classes. As each class relates to a table (in general), relationships are defined through properties which types are classes. Take a look at the class defined and mapped in Listing 10.

```
Listing 10 – Classes with Associations
  [Entity]
  [Automapping]
  TInvoice = class
  private
```

```
    FId: integer;
    FPerson: TPerson;
    FInvoiceDate: TDateTime;
    FItems: TList<TInvoiceItem>;
  public
    constructor Create;
    destructor Destroy; override;
    property Id: integer read FId;
    property Person: TPerson read FPerson write FPerson;
    property InvoiceDate: TDateTime read FInvoiceDate write FInvoiceDate;
    property Items: TList<TInvoiceItem> read FItems write FItems;
  end;
```

Class TInvoice will be mapped to an Invoice table. Note that this class has a property of type TPerson. This means that table Invoice will have a foreign key to table Person (many-to-one association). And there is also a list of itens (class TInvoiceItem, that is not present in Listing 10). Aurelius will create a new table Invoice_Item and will create a relationship with table Invoice (one invoice can have many invoice items).

It's also interesting to note that, as an invoice is related to class TPerson, we can use any object of type TPerson, and that means we can use both TCompany or TIndividual objects. Listing 11 shows how we would save an TInvoice object.

```
Listing 11 – Saving objects with associations
var
  Person: TPerson;
  Invoice: Tinvoice;
  InvoiceItem: TinvoiceItem;
  Manager: TObjectManager;
begin
  Manager := TObjectManager.Create(Conn);
  try
    Invoice := TInvoice.Create;
    Invoice.InvoiceDate := Date;
    Invoice.Person := Manager.Find<TPerson>(IdCompany.Id);
    InvoiceItem := TInvoiceItem.Create;
    InvoiceItem.Product := 'Hammer';
    InvoiceItem.Value := 10.50;
    Invoice.Itens.Add(InvoiceItem);
    Manager.Save(Invoice);
    Manager.Flush;
  finally
    Manager.Free;
  end;
end;
```

Queries

Finally, after saving and updating objects, what we need is a way to find objects in database. Aurelius provides a full-featured API to build and execute queries. It's similar to using SQL statements – you define search criteria like filtering, ordering, grouping, etc. Listing 12 shows an example of how to build a query.

```
Listing 12 – Executing a query
var
  Manager: TObjectManager;
  Invoices: TObjectList<TInvoice>;
begin
  Manager := TObjectManager.Create(Conn);
  try
    Invoices := Manager.Find<TNotaFiscal>
      .CreateAlias('Person', 'p')
      .Where(TLinq.GreaterThan('InvoiceDate', EncodeDate(2010, 1, 1))
         and (TLinq.Like('p.City', 'New%')))
      .AddOrder(TOrder.Asc('InvoiceDate'))
      .List;
    try
      // use Invoices list as you want
      // for example, for Invoice in Invoices do ...
    finally
      Invoices.Free;
    end;
  finally
    Manager.Free;
  end;
end;
```

In the example of Listing 12, we're searching for all invoices where invoice data will be greater than 01/01/2010, and also where the city's name of the person related to the invoice starts with "New". We also asked the invoices to be ordered by invoice date.

We can note two things: first, it's possible to search by association (just as we would do with SQL using LEFT or INNER JOIN), as in the example we are searching by the city of the person associated with the invoice. Second, inheritance also plays its role here. Since the invoice can be related to a TPerson, be it TIndividual or TCompany, the query will also follow this setup. So, Aurelius will execute an SQL statement that will search in the three tables (person, individual and company) and will return all invoices that match the provided criteria, be it individuals or companies.

Other features

Some other TMS Aurelius features are worth to be mentioned:

- TAureliusDataset: a full-featured TDataset descendant that allows you to visually bind objects to data-aware controls, making it easy to display lists, query results, or even edit object properties using db-aware controls. A whole new article could be written about the dataset features, and such topics are out of the scope of this article.

- Nullable properties: you can define properties as nullable (that receive null values)

- Lazy-loading: when loading an object with associations (Invoiec.Person, for example), you can configure Person property in a way that Aurelius will only retrieve Person's data when this property is really read.

- Blob fields: you can use blob fields (binary, text) and also define lazy-loading for them (only brings blob content when it's needed).

- Other Id types: besies integer types, you can use other types in primary key, as string or date types, as well composite id's (more than one field, although this is not a recommended practice).

- Complex queries: you can define queries with projections (expressions, calculated fields), grouping (Sum, Group By) and even directly set parts of the SQL statement, for more advanced scenarios.

Conclusion

The purpose of this article was to make an introduction to object-relational mapping, especially using the TMS Aurelius framework. Several other topics related to ORM frameworks could be presented and discussed, but these topics are out of the scope of this article. I hope that it has been interesting and that it was able to present you a new way of building database applications with Delphi. The full TMS Aurelius developers guide is included in the trial and registered versions of the product.

Author

Wagner Rafael Landgraf, 35, is graduated in Electronics Engineering and is has a Master of Science degree in Industrial Information Technology. Currently he's Product Manager at TMS Software, works with Delphi since its first version in 1995, and is responsible for products like TMS Scripter Studio, TMS Diagram Studio, TMS Data Modeler and TMS Aurelius.