European Commission

BioMA

Data Layer

Reference Documentation

Joint Research Centre

Release	Issue	Date
1	2	June 2019

Copyright

Disclaimer

European
Commission

Contents

About this document

1

This document describes the **BioMA Data Layer**, one of the auxiliary libraries of the BioMA Framework developed at JRC.

Tip:

For further information of the architecture and the components of the BioMA framework, please visit the Agri4Cast Software Portal.

This document is targeted to software developers who want to create an application or a model based on the BioMA Framework and need to connect to a database for reading or writing data.

Although it is always possible, in an application or a model, to connect to a database in a standard way (e.g., directly using the proper database driver), we recommend to use the Datalayer because it allows to decouple from the single database vendor. Many of the data providers or data persisters developed in the BioMA Framework are based on the Datalayer.

The normal user of the BioMA platform (e.g., the Bioma Spatial user or an agronomical modeller) does not need to read this document in order to use the platform unless he/she needs information about the mapping mechanism and the XML mapping file format. In this case, it is suggested to read “Generating the mapping file” on page 18.

The topics are organized as follows:

Topic	Description
“About the Data layer” on page 5	• What the Data layer is with respect to the BioMA framework architecture • Main purposes of the Data layer

Topic	Description
"Data Layer main libraries" on page 9	• How to map database tables and how to access the database • How to create tables, domain classes, and mapping file • How to operate on data • How to set locking strategies
"DataSetDBLayer library" on page 37	• What is it and how can be used the DataSetDB library • Example of usage

About the Data layer

2

This chapter is organized into the following sections:

Related topics:

- “Data Layer main libraries” on page 9
- “DataSetDBLayer library” on page 37

Purposes of the Data layer

The DataLayer Framework is a set of C# libraries that allows a programmer to manage the query and the modification of the database records abstracting the physical implementation of tables and hiding the real database structure (e.g., table and column names). This way, the C# program developer can use directly his own C# classes (domain classes) to query and modify the database records.

Domain classes and database tables are related each other via XML mapping files. These files can be automatically generated from the database tables or from the domain classes.

The framework is composed by three distinct libraries:

JRC.IPSC.MARS.DB (main DataLayer library)

It contains the core of the framework: objects and interfaces that allow the programmer to manage the mapping between domain classes and DB tables, as well as the most common operations on records (i.e., insert a new record, delete or modify an existing record, select a set of records).

JRC.IPSC.MARS.DatabaseUtility

It contains a set of tools that allow the programmer to automatically create tables in a pre-existent database, from a strongly-typed DataSet or from domain classes. Moreover, these allow the programmer to automatically generate the XML mapping files.

These first two libraries are described in this document in chapter “Data Layer main libraries” on page 9.

EC.JRC.MARS.DataSetDBLayer

It contains an alternative version of the DataLayer that does not require the usage of domain classes. It uses **System.Data.DataTable** objects instead of the domain classes. It can also use **EC.JRC.MARS.ModelLayer.Data.Table** objects, which are part of the **BioMA Model Layer** framework (go to the **Agri4Cast Software Portal** to download the Model Layer document). This library is less powerful than the main DataLayer library and is described in the last chapter of this document “DataSetDBLayer library” on page 37.

The framework also includes some tools that allow the programmer to generate the C# code of the domain classes starting from pre-existing database tables. These tools use the open-source software DBLinq and the Microsoft application SQLMetal.

Data Layer main libraries

3

This chapter is organized into the following sections:

Related topics:

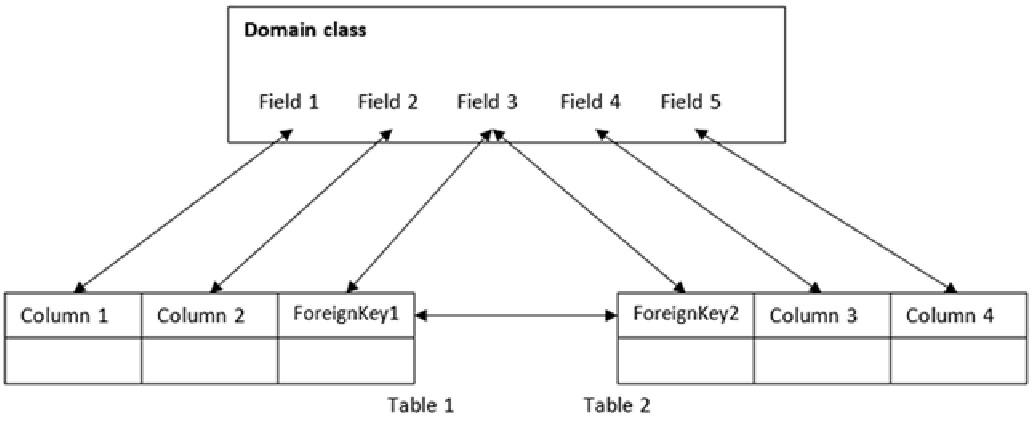
- “DataSetDBLayer library” on page 37

Mapping between tables and domain classes

The main concept of the Data Layer framework is the mapping between database tables and domain classes.

In the simplest case, a domain class (with certain public Fields and/or Properties) is associated to a database table. The table columns are associated one-to-one with the public Fields and/or Properties of the domain class. It is not necessary that all Fields and/or Properties are associated with the table columns and vice-versa.

In a more complicated case, a domain class can be associated to more than one table. So a Field of the class can be mapped into a column of a table, while another Field can be mapped into a column of another table. In this case it is mandatory that a one-to-one relation exists between the two tables through a foreign key (see Figure below):



See also:

- “Accessing the database” on page 11
- “Creation of tables, domain classes, and mapping file” on page 13
- “Data operations” on page 21
- “Records locking strategies” on page 32

Accessing the database

The DataLayer framework is based on the ADO.NET Disconnected Layer framework and is used to access and modify database data.

The two main actors of this framework are:

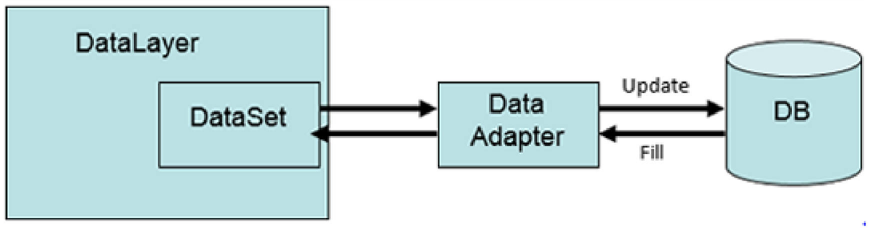
- **DataSet class**: This class allows including a copy of the database data and schema in the memory space of the application. The "DataSet" type is a container for any number of "DataTable" objects, each of which contains a collection of "DataRow" and "DataColumn" objects.
- **DataAdapter class**: This class allows copying the data and schema of the database to the DataSet and committing data changes to the database. The DataAdapter object of the specific data provider handles the database connection automatically.

  Unlike the connected layer, the data obtained via a DataAdapter are not processed using data reader objects.

DataAdapters keep the connection open for the shortest possible time. Once the caller receives the DataSet object, the calling tier is completely disconnected from the database and left with a local copy of the remote data. The caller is allowed to insert, delete, or update rows from a given ”DataTable”, but the physical database is not updated until the caller explicitly passes the DataSet to the data adapter for updating.

The following summarizes how the Disconnected Layer works:

1. A DataAdapter retrieves data from a database table through its **Fill** command. The fill method is invoked with a SQL select string parameter. This SQL string is used to select the records to retrieve from the database.
2. The DataAdapter inserts the retrieved data into a DataSet object. It can be initially empty, or filled with data from other tables.
3. The application reads and modifies the records, acting on the tables contained in the DataSet.
4. The DataAdapter transfers the changes to the database through its **Update** command.



As shown in the figure above, the DataLayer framework uses the ADO.NET Disconnected Layer to access the database data.

This provides two benefits:

- The ADO.NET Disconnected Layer is database provider independent (the framework provides a different DataAdapter for each database type). So, the DataLayer framework is database provider independent too.
- The number of access to the physical database is minimized.

Setting up a C# project

To use the DataLayer in a C# project the programmer needs to make the project compatible with version 3.0 (or more) of .NET libraries.

The use of .NET 3.5 framework is suggested because this way LINQ can be used within the project. LINQ allows the programmer to simplify the access to data structures managed by the DataLayer.

The programmer needs also to set the project references to the two DLLs that compose the DataLayer: JRC.IPSC.MARS.DB and JRC.IPSC.MARS.DatabaseUtility.

See also:

- “Mapping between tables and domain classes” on page 10
- “Creation of tables, domain classes, and mapping file” on page 13
- “Data operations” on page 21
- “Records locking strategies” on page 32

Creation of tables, domain classes, and mapping file

In order to start using the DataLayer to manage data, a programmer must have references to the database tables, to (create and) import domain classes and to generate the XML mapping file.

So, before writing the application code, the programmer needs to properly create and set-up the entities.

Here you find two common cases:

- The programmer has at his disposal a set of domain classes (or a strongly typed DataSet) but the database tables do not exist.
- The programmer has at his disposal a set of database tables but there are no C# classes to map table data in a C# application.

The following sections describe the solutions for managing these two cases, for creating database tables from domain classes and vice-versa. Moreover, the procedure to create the XML mapping file is described.

See:

- "Generating database tables from a strongly-typed" on page 14
- "Generating database tables from a set of domain classes" on page 15
- "Generating domain classes code from database tables" on page 17
- "Generating the mapping file" on page 18

Generating database tables from a strongly-typed

The DatabaseUtility library provides methods to generate the tables and their primary and foreign keys from an ADO.NET DataSet object.

For each DataTable object included in the DataSet, the library creates a database physical table. For each Column object in a DataTable, the library creates a column in the table. The column type will be created on the basis of the Column value data type.

Data type association rules are different depending on the database provider. The following table depicts the relations between C# Column types and database types for the two database provided in this version:

C# Type	Oracle Column Type	SQLServer Column Type
`System.Int16`	INTEGER	int
`System.Int32`	INTEGER	int
`System.Int64`	INTEGER	int
`System.Decimal`	NUMBER	numeric (28, 10)
`System.Double`	NUMBER	numeric (28, 10)
`System.String`	VARCHAR2 (data column max length)	nvarchar (data column max length)
`System.DateTime`	DATE	datetime

To generate a table from a DataSet:

1. Import the DataSet object into the project (even as a "reference"). This way, the represented C# class can be referenced.
2. Ensure that the DataSet contains all (and only) the tables that must be created. In the generation phase, the program will try to create all tables in the DataSet.
3. Create one instance of the DatabaseUtility façade.
4. Call the `CreateDBTablesFromDataset` method, as shown below:

```
//get an instance of the DatabaseUtilityFacade
            JRC.IPSC.MARS.DatabaseUtility.DatabaseUtilityFacade
facade = new DatabaseUtilityFacade();
            //get an instance of the strongly typed DataSet
            DataSet dataset = new ExampleDataSet();
            //use the facade method: CreateDBTablesFromDataset
            facade.CreateDBTablesFromDataset(connstring,
JRC.IPSC.MARS.DatabaseUtility.Provider.SQLServerCompact, dataset);
               Console.WriteLine("tables created from dataset!");
```

5 Ensure that the tables were actually created in the database with their primary and foreign keys.

If a table already exists in the database, an excelption will be thrown.

To check this, the programmer can invoke the `CheckExistsTable` method of the same façade, as follows:

```
//get an instance of the DatabaseUtilityFacade
if (facade.CheckExistsTable("ExampleTable", connstring,
JRC.IPSC.MARS.DatabaseUtility.Provider.SQLServerCompact))
           {
                       //do something
 }
```

Generating database tables from a set of domain classes

The DatabaseUtility library provides methods to generate tables and their primary and foreign keys from an array of C# Type objects, representing a set of domain classes.

The library uses the ADO.NET DataSet paradigm as the underlying layer to access the database. So, the generation procedure uses the same procedure as described in the previous paragraph (see “To generate a table from a DataSet:” on page 14), and the data type conversion follows the same rules.

The generation procedure creates a database table for each domain class provided, with the same name of the domain class. Every table is created with columns corresponding to every public Property/Field objects of the domain class. The table column name will be equal to the Property/Field name.

Example:

The class `Prova:`

```
public class Prova
        {
            private string miachiave;
```

```
                public string miachiave
                {
                    get { return miachiave; }
                    set { miachiave = value; }
                }
                private string miodato;

                public string mydata
                {
                    get { return miodato; }
                    set { miodato = value; }
                }
                private Decimal miodecimal;

                public Decimal miodecimal
                {
                    get { return miodecimal; }
                    set { miodecimal = value; }
                }
            }
```

generates the following table:

Test class	
Miachiave	nvarchar(256)
Miodato	nvarchar(256)
Miodecimal	numeric(28,10)

To generate the tables from a set of classes:

1. Import the classes in the project, even as “references”.
2. Ensure that you are using all (and only) the classes to be converted into tables. In the generation phase, the program will try to create a table for each class.
3. Create an instance of the DataUtility façade.

4 Call the `CreateDBTablesFromDomainClasses` method, as shown below:

```
//get an instance of the DatabaseUtilityFacade
            JRC.IPSC.MARS.DatabaseUtility.DatabaseUtilityFacade
facade = new DatabaseUtilityFacade();
            //create an array with the domain classes types
           Type[] domainClasses = new Type[] {typeof (ProvaRelease),
typeof (ProvaRelease)};
            //use the facade method: CreateDBTablesFromDomainClasses
facade.CreateDBTablesFromDomainClasses(connstring,
JRC.IPSC.MARS.DatabaseUtility.Provider.SQLServerCompact,
domainClasses);
               Console.WriteLine("tables created!");
```

5 Ensure that the tables were actually created in the database.

Generating domain classes code from database tables

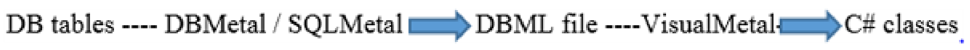
In the open-source/freeware software community there is a number of softwares that allow programmers to generate class code from database tables.

In the DataLayer framework, two programs are used, depending on the database provider:

- DBLinq’s DBMetal for Oracle, PostgreSQL, MySql databases
- Microsoft’s SQLMetal for SQLServer databases

Both these tools act in the same way: they read the database schema and then code it into an XML file in the DBML (Database Markup Language) standard.

From the DBML file, the VisualMetal application generates the corresponding C# class for each table.



Tools references:

- Microsoft's SQLMetal: it is part of the .NET release 2.0 (or more). Documentation: http://msdn.microsoft.com/it-it/library/bb386987.aspx

- DBLinq's DBMetal and VisualMetal: tools created by the open source project DBLinq
- DBML standard: Microsoft standard to describe a database schema: http://msdn.microsoft.com/en-us/library/bb399400.aspx

Example of usage of SQLMetal (with a SQLServerCe ".sdf" file):

```
SQLMetal.exe /dbml:dbmlfile.dbml "D:\....\mydb.sdf"
```

Example of usage of DBMetal with an Oracle DB:

```
DbMetal.exe /provider:Oracle   /conn:"Data Source=XXX; User ID=xx;
Password=xx" /database:xxx ./dbmlfile.dbml" /culture:IT
```

Usage of VisualMetal:

1. Open the **VisualMetal GUI** application. (To install it, launch the VisualMetal.exe file that can be found in the .NET installation directory).
2. From the menu, select **File > Open DBML**.
3. Open the DBML file. As a result, the tables list will be shown in the middle of the program interface.
4. From the menu, select **File > Generate C#**.
5. Choose the name and the path of the .CS file that will be created. The file will contain the code of the classes generated from all the tables in the DBML.

Generating the mapping file

The XML mapping file describes the association between the database tables and the domain classes, as depicted in section "Mapping between tables and domain classes" on page 10.

The mapping file schema is formally described in the XSD schema `mappingschema_x.x.xsd` (where `x.x` is the version number). The `mappingTemplate.xml` file is a template that can be used to manually create a file. Both these files are provided in the DataLayer release, in the **mappingSchema** folder.

To manually create a mapping file:

1. Create a file header like the following:

```
<?xml version="1.0" encoding="utf-8"?>
```

2 Create a root tag like the following:

```
<mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="JRC.IPSC.MARS.DB" xmlns="JRC.IPSC.MARS.DB">
```

3 For each domain class to map, create a **domainMapping** tag, with one mandatory attribute: **DomainClassName**. This attribute must be the same name of the domain class as used in the table method of class DBResolver.

For example, if the application uses the domain class `Prova`:

```
IDBTable<Prova> myTable = resolver.table<Prova>();
```

Meaning that, in this case, the **DomainClasseName** must be **`Prova`**.

4 For each table associated to the domain class, create a **tableMapping** element, with one mandatory field **TableName**. This attribute must be the name of the DB table. The element can also define the attribute **where**. This optional attribute can contain the SQL filter expression (without the keyword WHERE) used to fill table in DataSet with a filter, to avoid the loading of all rows.

5 For each column of the table mapped by the **tableMapping** element, create a **Column** tag. This element has got 2 mandatory attributes:

- **Name** represents the exact real table column name.
- **Member** represents the exact propriety or field name of the domain class.

6 If a domain class is associated to more than one DB table (see “Mapping between tables and domain classes” on page 10), it is mandatory to insert the relation information using the **relations** tag:

- Insert the **relations** tag inside the **domainMapping** tag.
- Inside the **relations** tag, insert one or more **relation** tags. This tag describes the single relation between tables.
- Insert the attributes of **relations** tag: **TableName1**, **TableName2**.
- Inside the **relation** tag, insert a **relationcolumns** tag.
- In the **relationcolumns** tag, insert one or more **relationcolumn** tags that enumerate the one-to-one column associations between table1-columns and table2-columns.
- Insert the attributes of **relationcolumn** tag: **TableColumnName1**, **TableColumnName2**.

Using the CreateXMLMapping tool to create the mapping file:

The framework provides the custom tool **CreateXMLMapping** that allows the programmer to avoid manually writing the file. The tool automatically generates the XML mapping file from a DBML file (to create one, see the previous section "To manually create a mapping file:" on page 18).

For a single table described in the DBML, the tool creates the corresponding XML mapping tag. The mapping from a table to a domain class with the same name is auto-generated, having fields corresponding to the table column names.

If table or columns names are different from C# object names, then the programmer must manually modify the XML so as to create the right associations.

In this version, the generator cannot manage neither the relations between a domain class and more than one table, nor the relations between tables (foreign keys). These associations must be manually created.

To run the **CreateXMLMapping** tool from a command line the programmer must provide two parameters: the DBML file name and the XML file name, as shown below:

```
CreateXMLMappingFromDBML.exe ".\temp\file.dbml" ".\temp\file.xml"
```

See also:

- "Mapping between tables and domain classes" on page 10
- "Accessing the database" on page 11
- "Data operations" on page 21
- "Records locking strategies" on page 32

Data operations

Once all the required entities have been set up through the mapping as described in the chapter “Creation of tables, domain classes, and mapping file” on page 13, the programmer can start to develop a C# application in order to access database data, via the JRC.IPSC.MARS.DB library of the DataLayer framework.

In the following sections are provided some examples of the most common functionalities of the framework.

These examples use the dummy C# class `Prova` as shown below:

```
public class Prova
        {
            private string miachiave;

            public string Miachiave
            {
                get { return miachiave; }
                set { miachiave = value; }
            }
            private string miodato;

            public string Miodato
            {
                get { return miodato; }
                set { miodato = value; }
            }
            private Decimal miodecimal;

            public Decimal Miodecimal
            {
                get { return miodecimal; }
                set { miodecimal = value; }
            }
          }
```

This class is mapped to the physical database table **TabellaProva** through the mapping file shown below (please, note that in the examples the physical names of the table and of his columns were never used):

```
<mappings>
  <domainMapping DomainClassName="Prova">
    <tableMapping TableName="TabellaProva">
      <Column Member="Miachiave" Name="MIA_CHIAVE" />
      <Column Member="Miodato" Name="MIO_DATO" />
      <Column Member="Miodecimal" Name="MIO_DECIMAL" />
    </tableMapping>
  </domainMapping>
</mappings>
```

For further information, see:

- “Creating the IDB Resolver object and obtaining a reference to a table” on page 23
- “Multi-tables management by the same resolver” on page 23
- “Loading filtered data” on page 24
- “Reading records” on page 26
- “Adding a new record” on page 27
- “Deleting a record” on page 28
- “Committing changes to the physical database” on page 29
- “Rolling back changes” on page 30

Creating the IDB Resolver object and obtaining a reference to a table

The reference to an underlying table is provided from the IDBResolver interface. To create an instance of an `IDBResolver` implementation, the programmer must call the `getDbResolver` method of the factory class `DBResolverFactory`. Method parameters:

- The name of the XML mapping file relative to the domain classes used in the program.
- The database connection string.
- The database provider identifier. The `Provider` enumeration allows the programmer to choose between the supported providers (currently Oracle, PostreSQL, SQLServer, or SQLServer compact edition).

```
IDBResolver resolver = DBResolverFactory.getDbResolver("./
miofile.xml",connstring,JRC.IPSC.MARS.DB.Provider.SQLServerCompact)
;
```

Once created the `IDBResolver` object, the programmer can use it to retrieve the reference to an `IDBTable` object, through the `table` method. The `IDBTable` object is the "logical" reference to the physical table(s) associated to the domain class.

In the example below, the program obtains the reference to the table linked to domain class `Prova`, using the `table` method:

```
IDBTable<Prova> myTable = resolver.table<Prova>();
```

Multi-tables management by the same resolver

Each resolver can handle only one instance of a table related to a certain domain class. So, if the method is called with the same domain class type parameter, the resolver returns the same `IDBTable` instance .

The following shows an example:

```
IDBTable<Prova> myTable1 = resolver.table<Prova>();

IDBTable<Prova> myTable2 = resolver.table<Prova>();
```

In this case `myTable1` and `myTable2` are references to the same instance of `IDBTable.`

However, the resolver can handle many `IDBTable` objects that relate to different domain classes.

The following shows an example:

```
IDBTable<Prova1> myTable1 = resolver.table<Prova1>();
IDBTable<Prova2> myTable2 = resolver.table<Prova2>();
```

In this case `myTable1` and `myTable2` are references to different instances of `IDBTable`.

The name of the domain class, that is used as type parameter in the table method, must be the same name used in the XML configuration file: the domain class type name must match exactly the value in the XML file (that is, the value that has been written in the **DomainClassName** attribute of the **domainMapping** element tag).

If, for some reasons, this is not the case, it is possible to specify in the table method the name of the object as used in the **DomainClassName** attribute of the **domainMapping** element tag.

For example, if this is the mapping file related to the domain class `Prova`:

```
 <domainMapping DomainClassName="MyProva">
        <tableMapping TableName="PROVARELEASE" >
        … columns …
        </tableMapping>
</domainMapping>
```

the table object must be created as follows:

```
IDBTable<Prova> myTable1 = resolver.table<Prova>("MyProva");
```

Loading filtered data

When the `table` method is called, the DataLayer has to fill the database table data into the application memory. To do this, the ADO.NET framework is used, as depicted in section “Accessing the database” on page 11: the DataAdapter object fills the DataSet with database table data using a SQL Select command.

All the records of the table are retrieved and stored into the application memory.

For example, if the `Prova` domain class is associates whit the **TabellaProva** table, the SQL Select command used by the DataAdapter is the following:

```
Select * from TabellaProva
```

This behavior is not very convenient if the table has got many records, because the storing procedure requires a big amount of memory and this is really useless when the application needs to read/modify only a little set of records. Luckily, the DataAdapter allows filtering the records to retrieve using a **where** clause in his SQL Select command. This is supported by the DataLayer framework.

Please, note that the application does not know the record that were not loaded. These will not appear in queries and cannot be modified.

To fill the tables using filters:

1 For each table to filter, do the following:

- Create a **WhereCondition** object using the constructor. Constructor parameters: Physical table name, SQL filter condition (without the keyword **where**).
- Insert the **WhereCondition** object in a **WhereCondition** array.

2 Call the **getDbResolver** method of the factory class DBResolverFactory overloaded with the **WhereCondition** parameter.

3 Obtain the reference to the **IDBTable** object, through the table method, as usual.

Example:

```
                //create a WhereCondition object for every table to
filter (only “TabellaProva” in this example)

                WhereCondition[] whereConditions = new
WhereCondition[1];

            whereConditions[0] = new WhereCondition("TabellaProva",
"Miodecimal< 100");


                //pass the WhereCondition[] parameter in the
getDbResolver method of the DBResolverFactory

                IDBResolver resolver =
DBResolverFactory.getDbResolver

                 ("./
miofile.xml",connstring,JRC.IPSC.MARS.DB.Provider.SQLServerCompact,
whereConditions);
```

```
//obtain the (filtered) table
IDBTable<Prova> myTable1 = resolver.table<Prova>();
```

Reading records

Once created the `IDBTable` object relative to the domain class we want to manage, it is possible to read the underlying associated table(s).

To read, for example, read all records of the associated table(s) the programmer must call the `SelectAllRows` method of `IDBTable` class, as shown below:

```
IList<IDBRow<Prova>> allRows = myTable.SelectAllRows();
```

The method returns a list (`IList`) of `IDBRow` objects. These represent the reference to the rows of physical associated table(s). Each `IDBRow` object provides the `DomainClassObject` public property. This property allows the programmer to access to the instance of the domain class associated to the row.

For example, using this property, the programmer can retrieve and print the list of all `Miachiave` fields of all rows in the associated **ProvaRelease** physical table, as shown below:

```
        IList<IDBRow<Prova>>allRows = myTable.SelectAllRows();

        foreach (var row in allRows)
        {
             Console.WriteLine(row.DomainClassObject.Miachiave);
        }
```

To filter the records, the programmer can use the `Select` method and insert a filter expression (`IBooleanExpression` class) that uses the names of the domain class fields as the filter parameters :

```
            IBooleanExpression filterExpr =
myTable.getBooleanExpression("Miachiave", "LIKE", "%xxx%");
            IList<IDBRow<Prova>> filteredRows =
myTable.Select(filterExpr);
```

The programmer can combine more than one filter expression with logical operators (`BooleanOperator` class):

```
            IBooleanExpression filterExpr1 =
myTable.getBooleanExpression("Miachiave", "LIKE", "%xxx%");

            IBooleanExpression filterExpr2 =
myTable.getBooleanExpression("Miodecimal", "<", "100");

            IBooleanExpression filterExpr = new
BooleanExpression(filterExpr1,BooleanOperator.AND,filterExpr2);

            IList<IDBRow<Prova>> filteredRows =
myTable.Select(filterExpr);
```

In any case, to filter the records, it is better and simpler to use the Microsoft LINQ library. Since the `IList<T>` class implements the `IEnumerable<T>` interface, it is possible to apply LINQ functions on the result of the `SelectAllRows` method of `IDBTable` class, as shown below:

```
IList<IDBRow<Prova>> allRows = myTable.SelectAllRows();

            IList<IDBRow<Prova>>filteredRows = from a in allRows
where a.DomainClassObject.Miodecimal< 100 select a;
```

Please remember that, in order to use LINQ in a C# project, the project must be compatible with .NET 3.5 Framework. Moreover, the programmer needs to import the `System.Linq` namespace.

Adding a new record

The `IDBTable` class provides a method to insert a new row in the associated table(s), managing a new instance of the domain class.

To add a new record the programmer must:

1 Create a new instance of the class containing the values that will be written in the table(s).

2 Invoke the `createAndInsertNewRow` method of the `IDBTable` class.

3 Call the `commit` method of `IDBTable` class to commit data to the database (see “Committing changes to the physical database” on page 29). Note that the `IDBTable` object must be the one containing the `IDBRow` object.

```
IDBTable<Prova> myTable = resolver.table<Prova>();

IList<IDBRow<Prova>> allRows = myTable.SelectAllRows();
```

```
            IList<IDBRow<Prova>>filteredRows = from a in allRows
where a.DomainClassObject.Miachiave.Equals("key") select a;

            IDBRow<Prova> myRow = filteredRows.Single();

            myRow.DomainClassObject.Miodato = "new string";

               myRow.Update();

            myTable.commit();
```

Note the usage of the LINQ `Single()` method: It ensures that one and only one instance is in the `filteredRows` list, and returns it. The LINQ `Single()` method throws an exception if the list is empty or if there is more than one instance.

In the example above, we suppose that the field `Miachiave` is a unique primary key; that is, filtering using this field, the query will return one (or zero) record.

Deleting a record

The `IDBRow` class provides a method to delete the physical row(s) in the domain class associated table(s).

The delete procedure is similar to the modify procedure as described in the previous paragraph. Firstly, the programmer must retrieve the right instance of the `IDBRow` class.

Once the `IDBRow` object is properly valued, the programmer can delete the row(s) by completing the following steps.

To delete a record:

1. Call the `Delete` method of the `IDBRow` object. This method deletes the corresponding row(s) of the table(s) stored in program memory.
2. Call the `commit` method of IDBTable class to commit data changes to the database (see commit paragraph for further details). Please note that the `IDBTable` object must be the one that contains the `IDBRow` object.

```
IDBTable<Prova> myTable = resolver.table<Prova>();

            IList<IDBRow<Prova>>allRows = myTable.SelectAllRows();

            IList<IDBRow<Prova>>filteredRows = from a in allRows
where a.DomainClassObject.Miachiave.Equals("key") select a;

            IDBRow<Prova> myRow = filteredRows.Single();
```

```
myRow.Delete();

myTable.commit();
```

Committing changes to the physical database

The changes commit is managed at domain class level by the `IDBTable` object. The `IDBTable` class provides the `commit` method that commits to the database every change that affected the table stored in memory.

Once the `IDBTable` object is properly valued and changes are made, the programmer can commit changes by completing the following step.

To commit changes:

- Call the `commit` method of `IDBTable` class to commit data changes to the database.

Note one

if the same `IDBResolver` object is used to manage more than one domain class, the commit will affect only the physical tables associated to the `IDBTable` object that invoked the commit method, as shown below:

```
               //changes on domain class Prova

IDBTable<Prova> myTable = resolver.table<Prova>();

IList<IDBRow<Prova>> allRows = myTable.SelectAllRows();

            IList<IDBRow<Prova>>filteredRows = from a in allRows
where a.DomainClassObject.Miachiave.Equals("key1") select a;

            IDBRow<Prova> myRow = filteredRows.Single();

            myRow.DomainClassObject.Miodato = "new string";

            myRow.Update();


               //changes on domain class OtherProva

IDBTable<OtherProva>myOtherTable = resolver.table<OtherProva>();

IList<IDBRow<Prova>> allOtherTableRows =
myOtherTable.SelectAllRows();

IList<IDBRow<OtherProva>> otherRows = from a in allRows where
a.DomainClassObject.Miachiave.Equals("key2") select a;

            IDBRow<OtherProva> myOtherRow = otherRows.Single();

myOtherRow.DomainClassObject.Miodato = "new other string";
```

```
            myOtherRow.Update();


               //it commits only the first change to the db, the one
applied on the "myTable" object
myTable.commit();
```

Note two

In the previous paragraphs, the commit was called just after the insert/modify/delete command. This is not mandatory: it is possible to execute an undefined number of changes on a table, calling the commit only at the end of the last command, as shown below:

```
//change 1 on domain class Prova
IDBTable<Prova> myTable = resolver.table<Prova>();
IList<IDBRow<Prova>> allRows = myTable.SelectAllRows();
            IList<IDBRow<Prova>>filteredRows = from a in allRows
where a.DomainClassObject.Miachiave.Equals("key1") select a;
            IDBRow<Prova> myRow = filteredRows.Single();
            myRow.DomainClassObject.Miodato = "new string";
            myRow.Update();

//change 2 on domain class Prova
            IList<IDBRow<Prova>>otherRows = from a in allRows where
a.DomainClassObject.Miachiave.Equals("key2") select a;
            IDBRow<Prova> myOtherRow = otherRows.Single();
myOtherRow.DomainClassObject.Miodato = "new other string";
            myOtherRow.Update();

               //it commits 2 changes to the db
myTable.commit();
```

Rolling back changes

Roll-back changes is managed at domain class level by the `IDBTable` object. The `IDBTable` class provides the `rollback` method that rolls back any change that affected the table stored in memory.

Once the `IDBTable` object is properly valued and changes are made, the programmer can roll back changes by completing the following step.

To roll-back changes:

- Call the `rollback` method of `IDBTable`:

```
myTable.rollback();
```

See also:

- “Mapping between tables and domain classes” on page 10
- “Accessing the database” on page 11
- “Creation of tables, domain classes, and mapping file” on page 13
- “Data operations” on page 21

Records locking strategies

An application created on the DataLayer framework, runs in the Disconnected Layer context described in section “Accessing the database” on page 11.

In this context, it is necessary to have a look at what happens when another application tries to modify the same records that are stored (in disconnected mode) in the memory (DataSet object) of the first application.

The following is an example of a test case:

1. The application DataAdapter retrieves data from a database table through his **Fill** command.
2. The application DataAdapter inserts the retrieved data into a DataSet object.
3. The application modifies a record of the table, acting on the tables that are included in the DataSet.
4. Another application modifies the same record directly on the database.
5. The DataAdapter transfers the changes to the database through his **Update** command.

In this case, Step 5 will fail because the **Update** command of the DataAdapter does not commit changes if the record on the database has changed since the execution of the **Fill** command.

To avoid these problems, the DataAdapter can lock the records retrieved with the **Fill** command, releasing the lock only after the **Update** command is called. In the meantime, no other application can modify these records. This is called "pessimistic lock strategy".

If the DataAdapter does not lock the records it follows the "optimistic lock strategy": that is, a lock is in fact set, but only for the (brief) time necessary to commit (roughly from the start to the end time of the **Update** command).

This locking strategy choice is supported by the DataLayer in this way: when an `IDBTable` object is created from an `IDBResolver` object, the locking strategy parameter can be set, as show in the following example:

```
IDBTable<ProvaRelease>myTable =
resolver.table<ProvaRelease>(LockingStrategy.PESSIMISTIC_LOCK);
```

or

```
IDBTable<ProvaRelease>myTable =
resolver.table<ProvaRelease>(LockingStrategy.OPTIMISTIC_LOCK);
```

If the parameter is not set, the default strategy will be used, which is named “optimistic locking”.

For further information, see the following sections:

- “Usage of optimistic lock” on page 33
- “Usage of pessimistic lock” on page 33

Usage of optimistic lock

The optimistic lock is used by default. No additional code has to be written by the programmer.

All the examples shown in the previous paragraphs implement the optimistic lock.

Usage of pessimistic lock

Pessimistic locking is suitable for multi-application / multi-user environment, where many applications or many instances of the same application run simultaneously.

If the programmer chooses the pessimistic locking strategy, he/she must try to reduce at minimum the lock-time, releasing the lock as soon as possible.

Please note that if the table was loaded in the DataSet using some filter (see “Loading filtered data” on page 24), only the filtered and loaded records are locked.

The lock can be released using in two ways, as described in the following.

Way one

The programmer can release the lock committing the table changes by using the `commit` method of `IDBTable`, as shown in section “Committing changes to the physical database” on page 29.

Please note that, starting from when the locks are released, the `IDBTable` object works with optimistic strategy. It is not possible to have the same object keep working with pessimistic strategy because, in order to create new locks, the object must be totally re-filled with data. The only way to do this is by re-calling the **Fill** method of the DataAdapter object, and so creating a new `IDBTable` object.

```
//open the table with pessimistic locking

            IDBTable<Prova> myTable =
resolver.table<Prova>(LockingStrategy.PESSIMISTIC_LOCK);


            //modify/insert/delete rows of the table

            //....


            //commit changes and release the lock on records

            myTable.commit();


           //hereafter the myTable object, since released his locks,
works with optimistic stategy


            //modify/insert/delete rows of the table

            //....


            //simply commit changes in optimistic way

      myTable.commit();


             //if we want to work again with pessimistic locking on
the same table, create a new table on the domain class Prova

            IDBTable<Prova> myTable2 =
resolver.table<Prova>(LockingStrategy.PESSIMISTIC_LOCK);
      //now we work again with pessimistic locking
```

Way two

The programmer can release the lock without committing changes by using the `releaseLockWithoutCommit` method of `IDBTable`.

After this method is called, the lock is released, and the `IDBTable` object starts working in optimistic locking strategy. Previous data changes are not committed without being lost in the table stored in memory; this way, the programmer can further commit them on the application.

```
//open the table with pessimistic locking
```

```
            IDBTable<Prova> myTable =
resolver.table<Prova>(LockingStrategy.PESSIMISTIC_LOCK);

            //modify/insert/delete rows of the table
            //....

            //Releaese the records locks. Changes are neither
committed nor rolled back
            myTable.releaseLockWithoutCommit();

           //hereafter the myTable object, since released his locks,
works with optimistic stategy

            //simply commit changes in optimistic way
            myTable.commit();
```

Note that the rollback method of the `IDBTable` object does not affect the lock release, because it operates at a memory level, without interacting with the database.

Example:

```
//open the table with pessimistic locking
            IDBTable<Prova> myTable =
resolver.table<Prova>(LockingStrategy.PESSIMISTIC_LOCK);
            //modify/insert/delete rows of the table
            //....
            //rollback changes (the lock is NOT released)
            myTable.rollback();

            //the myTable keep working with pessimistic locking
```

See also:

- “Mapping between tables and domain classes” on page 10
- “Accessing the database” on page 11
- “Data operations” on page 21
- “Creation of tables, domain classes, and mapping file” on page 13

DataSetDBLayer library

4

This chapter is organized into the following sections:

Related topics:

- “Data Layer main libraries” on page 9

Introduction

The **EC.JRC.MARS.DataSetDBLayer** library is another Datalayer library that is aimed at saving the content of a DataSet object to a DB.

Unlike the **JRC.IPSC.MARS.DB** library, which requires the existence of a Domain class associated to the DB table, the DataSetDBLayer library associates a DataSet' DataTable to a DB table, without the need of creating a C# class for every table. This is useful when the structure of the C# objects is not known in advance.

Many of the concepts described for the main DataLayer library are valid for the DataSetDBLayer too.

For example, the mapping mechanism is the same: the user must create the XML mapping files that, in this case, will map the DB table columns to the DataSet table columns (rather than the domain class properties).

The EC.JRC.MARS.DataSetDBLayer library works with DB tables only if these define a primary key.

The main object of the EC.JRC.MARS.DataSetDBLayer library is the DataSetManager class. It contains the methods to save a DataSet table to the database.

To easily integrate the library in the BioMA Framework, the DataSetManager class also provides a method to save an EC.MARS.ModelLayer.Data.Table object to a database. The EC.MARS.ModelLayer.Data.Table is the class of the BioMA Model Layer that represents a table containing the model output (see the **BioMA Model Layer Documentation** for more details, which you can download from the Agri4Cast Software Portal).

Description of the DataSetManager public methods

In the following is provided a short description of the public methods of the DataSetManager.

DataSetManager (DataSet dataSet, string connectionString, string providerName)

Constructs an instance of this class using an existing DataSet.

bool *TestConnection ()*

Returns *true* if the connection can be opened, *false* otherwise. If the connection is already open, returns *true*. If the connection is not already open the method tries to open it. If the connection open succeeds, then the program closes it again.

void *SaveTableToDB (Table table, IDBMapper mapper)*

Saves a EC.JRC.MARS.ModelLayer.Data.Table to the database, using a specified mapper to map the Table columns to the DB Columns. To convert (if necessary) the values to the Type of the DB Column, it uses one of the registered ValueConverters.

void *SaveTableToDB (Data Table table, IDBMapper mapper)*

Saves a DataTable to the database, using a specified mapper to map the DataTable columns to the DB Columns. To convert (if necessary) the values to the Type of the DB Column, it uses one of the registered ValueConverters .

void *CloseEverything ()*

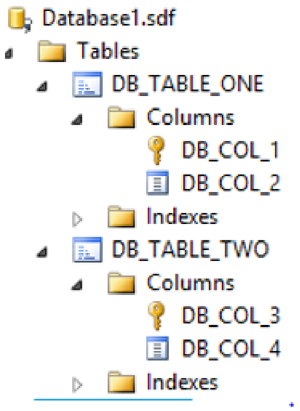
See also:

- “Example of usage” on page 40

Example of usage

In this example we will use the DataSetDBLayer library to save the content of a sample DataSet into a database.

The next figure shows the structure of the sample database. In this case we use a SQLServer compact edition database:



The DataSet structure is similar. We use a dataset composed by two tables, containing two columns each. Note that both tables have a primary key (mandatory).

To map the DataSet structure into the DB structure, we use a mapping file. The structure of the DataSetDBLayer mapping files is identical to the structure of the main DataLayer library mapping files.

Content of our sample mapping file

```
<?xml version="1.0" encoding="utf-8" ?>

<mappings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="JRC.IPSC.MARS.DB mappingschema.xsd"
xmlns="JRC.IPSC.MARS.DB">

  <domainMapping DomainClassName="FirstTable" >

    <tableMapping TableName="DB_TABLE_ONE"  >

      <Column Name="DB_COL_1"  Member="Col1" />

      <Column Name="DB_COL_2"  Member="Col2" />

    </tableMapping>
  </domainMapping>
  <domainMapping DomainClassName="SecondTable" >
    <tableMapping TableName="DB_TABLE_TWO" >
      <Column Name="DB_COL_3" Member="Col3" />
      <Column Name="DB_COL_4" Member="Col4"  />
    </tableMapping>
  </domainMapping>
</mappings>
```

The following is the code to use the library starting from a DataSet object (collection of System.Data.DataTable objects):

```
     class Program
    {
        static void Main(string[] args)
        {
            //create and fill the sample dataset (2 tables)
            DataSet ds=new DataSet();
            ds.Tables.Add("FirstTable");
            ds.Tables.Add("SecondTable");
            ds.Tables["FirstTable"].Columns.Add("Col1");
            ds.Tables["FirstTable"].Columns.Add("Col2");
            ds.Tables["SecondTable"].Columns.Add("Col3");
            ds.Tables["SecondTable"].Columns.Add("Col4");
            ds.Tables["FirstTable"].Rows.Add(new object[] {1, 3.4});
            ds.Tables["FirstTable"].Rows.Add(new object[] { 2, 5.4 });
           ds.Tables["SecondTable"].Rows.Add(new object[] { "abc", 4.2 });
           ds.Tables["SecondTable"].Rows.Add(new object[] { "def", 1.4 });

            string ConnString = "insert here the connection string";
            string ProviderName = "System.Data.SqlServerCe.3.5";//name of
the sqlserver ce driver
```

```
            //create the DataSetManager object (please note it is based on
another new dataset object)
            DataSetManager dsm = new DataSetManager(new DataSet()
, ConnString, ProviderName);

            //listen to DataSetManager error/warning messages
            dsm.errorMessageEvent += ShowMessage;
            dsm.warningMessageEvent += ShowMessage;

            //test DB connection
            if (!dsm.TestConnection())
            {
                ShowMessage("Error connecting to DB");
                return;
            }

            //create the mapper object based on the XML mapping file
            string MappingFilePath = "SampleMapping.xml";
            IDBMapper mapper = new
TableColumnsToDomainClassFields(MappingFilePath);

            foreach (DataTable table in ds.Tables)
            {
               if (mapper.getDomainClassNames().Contains(table.TableName))
                {
                    //save the table
                    dsm.SaveTableToDB(table, mapper);
                    ShowMessage("Table '" + table.TableName + "' saved to
the database");
                }
            }

            Console.WriteLine("End");
```

```
        }

        private static void ShowMessage(string msg)
        {
            Console.WriteLine("Message from data layer:"+msg);
        }
    }
```

The following is the code to use the library starting from a DataCollection object (collection of EC.MARS.ModelLayer.Data.Table objects):

```
            string ConnString = "insert here the connection string";
            string ProviderName = "System.Data.SqlServerCe.3.5";//name of
the sqlserver ce driver

//create and fill the DataCollection
            DataCollection dc= new DataCollection();
            dc.AddTable("FirstTable");
            dc.GetTable("FirstTable").AddColumn("Col1",typeof(double));
            dc.GetTable("FirstTable").AddColumn("Col2", typeof(double));
            dc.AddTable("SecondTable");
            dc.GetTable("SecondTable").AddColumn("Col3", typeof(string));
            dc.GetTable("SecondTable").AddColumn("Col4", typeof(double));
            dc.GetTable("FirstTable").AddRow(new object[] { 5, 5.4 });
            dc.GetTable("FirstTable").AddRow(new object[] { 6, 6.4 });
           dc.GetTable("SecondTable").AddRow(new object[] { "xyz", 5.4 });
           dc.GetTable("SecondTable").AddRow(new object[] { "mno", 6.4 });

            //create the DataSetManager object
            DataSetManager dsm3 = new DataSetManager(new DataSet(),
ConnString, ProviderName);



            //listen to DataSetManager error/warning messages
            dsm3.errorMessageEvent += ShowMessage;
            dsm3.warningMessageEvent += ShowMessage;

            //test DB connection
            if (!dsm3.TestConnection())
            {
                ShowMessage("Error connecting to DB");
                return;
            }

            //create the mapper object based on the XML mapping file
            string MappingFilePath = "SampleMapping.xml";
            IDBMapper mapper = new
TableColumnsToDomainClassFields(MappingFilePath);


            foreach (Table table in dc.Tables)
            {
                if (mapper.getDomainClassNames().Contains(table.Name))
                {
                    //save the table
                    dsm3.SaveTableToDB(table, mapper);
                    ShowMessage("Table '" + table.Name + "' saved to the
database");
                }
            }
```