Evaluant Universal Storage Services  
Reference Documentation

[1. Introduction to Euss 7](#_Toc206412061)

[1.1. Overview 7](#_Toc206412062)

[1.2. Open Source 7](#_Toc206412063)

[1.3. Features 7](#_Toc206412064)

[1.4. A Basic Euss Application 7](#_Toc206412065)

[1.4.1. Defining the class 7](#_Toc206412066)

[1.4.2. Adding a configuration file 8](#_Toc206412067)

[1.4.3. Loading and storing objects 9](#_Toc206412068)

[1.4.4. Creating a Mapping file 10](#_Toc206412069)

[1.5. Conclusion 11](#_Toc206412070)

[2. Architecture 12](#_Toc206412071)

[2.1. Overview 12](#_Toc206412072)

[2.1.1. Object Services 12](#_Toc206412073)

[2.1.2. Core Services 12](#_Toc206412074)

[2.1.3. Persistence Engines 13](#_Toc206412075)

[2.2. Contexts 13](#_Toc206412076)

[3. Configuration 14](#_Toc206412077)

[3.1. XML Configuration 14](#_Toc206412078)

[3.1.1. Usage 14](#_Toc206412079)

[3.1.2. Configuration Files 14](#_Toc206412080)

[3.2. Application Settings 15](#_Toc206412081)

[3.2.1. Declaring the section 15](#_Toc206412082)

[3.2.2. Defining the parameters 15](#_Toc206412083)

[3.2.3. Instantiating an ObjectService 16](#_Toc206412084)

[3.3. Programmatic Configuration 16](#_Toc206412085)

[3.4. Logging 16](#_Toc206412086)

[3.4.1. Overview 16](#_Toc206412087)

[3.4.2. Choosing the trace destination 17](#_Toc206412088)

[4. Persistent Classes 18](#_Toc206412089)

[4.1. Plain Old Classes 18](#_Toc206412090)

[4.1.1. Implement a default constructor 18](#_Toc206412091)

[4.1.2. Provide an identifier property (optional) 19](#_Toc206412092)

[4.1.3. Use non-sealed classes 19](#_Toc206412093)

[4.1.4. Declare fields as protected (optional) 19](#_Toc206412094)

[4.1.5. Declare property getter and setter on fields 19](#_Toc206412095)

[4.1.6. Declare reference properties as virtual 19](#_Toc206412096)

[4.1.7. Use IList for collections 19](#_Toc206412097)

[4.2. Generating the Classes 20](#_Toc206412098)

[4.2.1. Model files 20](#_Toc206412099)

[4.2.2. Euss Domain Model files and Visual Studio 20](#_Toc206412100)

[4.2.3. Xmi files and Visual Studio 21](#_Toc206412101)

[4.2.4. Using the command line utility 21](#_Toc206412102)

[5. Working With Objects 23](#_Toc206412103)

[5.1. Object States 23](#_Toc206412104)

[5.1.1. Transient 23](#_Toc206412105)

[5.1.2. Persistent 23](#_Toc206412106)

[5.1.3. Detached 23](#_Toc206412107)

[5.2. Managing Transactions 23](#_Toc206412108)

[5.3. Making Objects Persistent 24](#_Toc206412109)

[5.4. Loading an Object 24](#_Toc206412110)

[5.5. Querying 24](#_Toc206412111)

[5.5.1. Executing queries 24](#_Toc206412112)

[5.5.2. Scalar results 25](#_Toc206412113)

[5.5.3. Pagination 25](#_Toc206412114)

[5.5.4. Ordering results 25](#_Toc206412115)

[5.6. LINQ 26](#_Toc206412116)

[5.6.1. Referencing LINQ extension methods 26](#_Toc206412117)

[5.6.2. Executing queries 26](#_Toc206412118)

[5.6.3. Filtering on properties 26](#_Toc206412119)

[5.6.4. Loading references 26](#_Toc206412120)

[5.6.5. Constraining relationships 27](#_Toc206412121)

[5.6.6. Ordering results 27](#_Toc206412122)

[5.6.7. Paging results 27](#_Toc206412123)

[5.6.8. Aggregates 28](#_Toc206412124)

[5.6.9. Inferred loading 28](#_Toc206412125)

[5.6.10. Applying projections 28](#_Toc206412126)

[5.7. Modifying persistent objects 28](#_Toc206412127)

[5.8. Deleting persistent objects 28](#_Toc206412128)

[5.9. Getting and setting the identifier 29](#_Toc206412129)

[5.10. Using metadata 29](#_Toc206412130)

[5.10.1. Data about data 29](#_Toc206412131)

[5.10.2. Metamodel 32](#_Toc206412132)

[5.11. ADO.NET Data Services 32](#_Toc206412133)

[5.11.1. Implementing a DataContext 32](#_Toc206412134)

[5.11.2. Notes on implementation 33](#_Toc206412135)

[6. Euss in the Application Architecture 34](#_Toc206412136)

[6.1. Domain Layer 34](#_Toc206412137)

[6.2. Service Layer 34](#_Toc206412138)

[6.2.1. Contracts 35](#_Toc206412139)

[6.2.2. Implementations 36](#_Toc206412140)

[6.2.3. Service Factory 36](#_Toc206412141)

[6.3. Applying the Architecture 37](#_Toc206412142)

[6.3.1. Usage in ASP.NET 38](#_Toc206412143)

[7. Object-Relational Mapping 40](#_Toc206412144)

[7.1. Overview 40](#_Toc206412145)

[7.2. Configuration 40](#_Toc206412146)

[7.2.1. XML tag definitions 40](#_Toc206412147)

[7.2.2. Connections strings 41](#_Toc206412148)

[7.3. Generating the mapping file 41](#_Toc206412149)

[7.4. Mapping Declaration 41](#_Toc206412150)

[7.4.1. Entity mapping 41](#_Toc206412151)

[7.4.2. Identifier mapping 42](#_Toc206412152)

[7.4.3. Attribute mapping 44](#_Toc206412153)

[7.4.4. Reference mapping 47](#_Toc206412154)

[7.5. Mapping Relationships 47](#_Toc206412155)

[7.5.1. One-To-One relationships 47](#_Toc206412156)

[7.5.2. One-To-Many relationships 48](#_Toc206412157)

[7.5.3. Many-To-Many relationships 48](#_Toc206412158)

[7.6. Inheritance mapping 49](#_Toc206412159)

[7.6.1. The three strategies 49](#_Toc206412160)

[7.6.2. Table per class hierarchy 49](#_Toc206412161)

[7.6.3. Table per subclass 50](#_Toc206412162)

[7.6.4. Table per concrete class 51](#_Toc206412163)

[8. Other Storage Providers 53](#_Toc206412164)

[8.1. Xml Provider (Evaluant.Uss.Xml.XmlProvider) 53](#_Toc206412165)

[8.2. Sql Provider (Evaluant.Uss.Sql.SqlProvider) 53](#_Toc206412166)

[8.3. Memory Provider (Evaluant.Uss.Memory.MemoryProvider) 54](#_Toc206412167)

[9. Infrastructure Providers 55](#_Toc206412168)

[9.1. Hub Provider (Evaluant.Uss.Hub.HubProvider) 55](#_Toc206412169)

[9.2. Cache Provider (Evaluant.Uss.Cache.CacheProvider) 55](#_Toc206412170)

[9.3. Remoting Provider (Evaluant.Uss.Remoting.RemotingProvider) 56](#_Toc206412171)

[10. OPath 58](#_Toc206412172)

[10.1. Case Sensitivity 58](#_Toc206412173)

[10.2. Overview 58](#_Toc206412174)

[10.2.1. Select all clause 58](#_Toc206412175)

[10.2.2. Selecting references 58](#_Toc206412176)

[10.2.3. Constraints 58](#_Toc206412177)

[10.2.4. Referring to identifier property 58](#_Toc206412178)

[10.2.5. Filtering on the existence of a reference 59](#_Toc206412179)

[10.3. Logical Expressions 59](#_Toc206412180)

[10.3.1. Operators 59](#_Toc206412181)

[10.3.2. Functions 60](#_Toc206412182)

[10.3.3. Literals 60](#_Toc206412183)

[10.4. Referring to the Identifier 61](#_Toc206412184)

[10.5. OPath Examples 61](#_Toc206412185)

[10.6. Scalar Opath 62](#_Toc206412186)

[10.6.1. Aggregate functions 62](#_Toc206412187)

[10.6.2. Arithmetic expressions 62](#_Toc206412188)

[11. Tools and Utilities 63](#_Toc206412189)

[11.1. Command Line Tools 63](#_Toc206412190)

[11.1.1. Location 63](#_Toc206412191)

[11.1.2. Conventions 63](#_Toc206412192)

[11.2. mapping.exe 63](#_Toc206412193)

[11.3. domain.exe 64](#_Toc206412194)

[11.4. initialize.exe 65](#_Toc206412195)

[11.5. migrate.exe 66](#_Toc206412196)

[11.6. xmi2domain.exe 67](#_Toc206412197)

[11.7. scriptddl.exe 67](#_Toc206412198)

[12. License 68](#_Toc206412199)

# Introduction to Euss

## Overview

Evaluant Universal Storage Services (EUSS) is an extensible programming model and runtime components for building data aware solutions on the .NET platform. With EUSS you can access any type of data (XML, RDBMS, Objects, ...) with the same code and make production ready applications in record time.

Several advanced services like database neutrality, distributed caching or generic data stores plus a flexible licensing policy make EUSS the ideal solution for both open-source projects and commercial applications.

## Open Source

As a true open source project (based on the [MIT License](http://en.wikipedia.org/wiki/MIT_License)), you are free to use or distribute it, in the context of an open source but also commercial products. However, in order to use Euss in critical applications Evaluant (<http://www.evaluant.com>) provides professional support, making Euss a reliable solution for any type of use.

## Features

- **Open Source** under the MIT License, allowing commercial use  
- **Professional Support** for industry needs  
- **Object-Relational mapping** enabling you to reuse an existing database  
- **Generic relational data store** making relational databases simple as object databases  
- **In-Memory Database** (IMDB) providing never reached performance  
- **Synchronization Framework** for disconnected applications and staging scenarios  
- **Remote storage** (using .NET Remoting channels) for making distributed applications in record time    
- **XML based configuration** for easy deployment and seamless evolution of the data store  
- **Code generation** with ability to import major UML case tools files   
- **Dynamic Proxy** for importing your own domain classes   
- **Provider based architecture** allowing flexible configurable of the "persistence chain"   
- First and **second level cache** provider for improving performances  
- **RDBMS provider independence** to allow easy migration of you application  
- Dedicated versions for the .NET Frameworks 1.1, 2.0 and 3.5 (generics, nullable types, **LINQ** ...)  
- Persistence of **Interfaces** and **Generic types  
- Visual Studio .NET** integration.

## A Basic Euss Application

First we’ll create a basic console application storing some customer information in an XML file. Then we will use the same application for storing those information in a relational database.

Once you have downloaded and installed Euss you can create a new Console Application in Visual Studio.

### Defining the class

In the project you just created, add this class:

public class Customer

{

protected string name;

public string Name

{

get { return name; }

set { name = value; }

}

protected string address;

public string Address

{

get { return address; }

set { address = value; }

}

}

It is mandatory to use protected members as Euss will access them directly instead of using the public property.

### Adding a configuration file

Euss needs to know how to load and store objects of the persistent class. This is done using a configuration file.

This file contains information on the logic to use to load and store objects. This logic is handled by objects called **Persistence Engine**. A persistence engine is an instance of a **Persistence Provider** class, standing as a factory for them.

In a Euss configuration file we have to define what are the available Persistence Engines, i.e. what is the Persistence Provider and their inner properties.

You can create a new one by using a file template installed in Visual Studio with Euss: **Project, Add New Item, Euss Engine Configuration File**

By default the file is named engines.config and should look like this:

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

<PersistenceEngines

xmlns="http://euss.evaluant.com/schemas/EngineConfiguration.xsd"

DefaultEngine="Xml">

<PersistenceEngine Name="Xml" Factory="Evaluant.Uss.Xml.XmlProvider">

<FileName>euss.xml</FileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

</PersistenceEngines>

In this file, one Persistence Engine is configured using the provider XmlProvider. This provider is able to load and store objects using XML files. By default, the FileName property of this persistence engine is set, here euss.xml.

Any persistence engine needs some information about what are the different classes it will have to handle. This is done using the Metadata tag. Here we define that the information will be found in the assembly named BasicApplication, which is the name of the console application.

### Loading and storing objects

In you project, add a reference to Euss. You will find it listed among the standard assemblies.

Here is the code for the Main method:

using System;

using System.Collections.Generic;

using System.Text;

using Evaluant.Uss.ObjectContext;

namespace BasicApplication

{

class Program

{

static void Main(string[] args)

{

// Initilizes the configuration provider

ObjectService os = new ObjectService("engines.config");

// Creates a dedicated engine for the job

ObjectContext oc = os.CreateObjectContext();

// Deletes any already existing content and creates

// necessary resources

oc.InitializeRepository();

Customer c1 = new Customer();

c1.Name = "Customer One";

c1.Address = "1 Customer One avenue";

Customer c2 = new Customer();

c2.Name = "Customer Two";

c2.Address = "1 Customer Two avenue";

oc.BeginTransaction();

oc.Serialize(c1);

oc.Serialize(c2);

oc.CommitTransaction();

foreach(Customer c in oc.Load<Customer>(typeof(Customer)))

{

Console.WriteLine("Customer: {0} [{1}]", c.Name, c.Address);

}

}

}

}

In this sample we create two Customer instances, store them in the configured engine, and finally load them to display the information on the console.

Before executing the application, you’ll have to mark the engines.config file to **Copy Always** for the file to be copied in the same folder as the compiled program.

Here is the result:

Customer: Customer Two [1 Customer Two avenue]  
Customer: Customer One [1 Customer One avenue]

### Creating a Mapping file

Using an XML file to store the objects is somehow very simple but does not really correspond to a real scenario of an enterprise application. Instead we should use a dedicated relational database schema.

Euss is able to store and load objects from relational database prevented we define which tables and columns to use for each class we want to use. The file containing such information is called the **Mapping File**.

If you want to use a brand new schema you can ask Euss to generate a dedicated one using the mapping.exe command line utility. Some detailed information on how to use this tool can be found in the section [Command Line Tools](#_migrate.exe).

The mapping file is made of distinct Entity tags, one per class to describe. In our example it is as simple as this:

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

<Mapping

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<Entity type="BasicApplication:Customer" table="Customer">

<Id field="CustomerId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name"

db-type="String" size="50" not-null="false" />

<Attribute name="Address" field="Address"

db-type="String" size="50" not-null="false" />

</Entity>

</Mapping>

You should notice that this file is vendor independent, i.e. there is no reference to any database name of proprietary value. Here the Customer objects will be stored in the Customer table. Its table will be given a CustomerId primary key, plus a Name and Attribute columns

Now we need to add a new Persistence Engine section in the configuration file to inform the application it should use a relational database to store and load objects. This behavior is held by a specific Persistence Provider, named SqlMapperProvider.

We add this section to the engines.config:

<PersistenceEngine Name="Sql" Factory="Evaluant.Uss.SqlMapper.SqlMapperProvider">

<ConnectionString>

Server=.;DataBase=BasicApplication;UID=sa;PWD=

</ConnectionString>

<Driver>Evaluant.Uss.SqlMapper.MsSqlDriver</Driver>

<Dialect>Evaluant.Uss.SqlMapper.MsSqlDialect</Dialect>

<MappingFileName>BasicApplication.eum.xml</MappingFileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

There are two more things to do before launching the application again. First you have to create a database corresponding to the connection string defined in the configuration. In the example above the database is named BasicApplication. Second you have to change the default used Persistence Engine, in the DefaultEngine attribute of the PersistenceEngines tag.

Launch the application, the result should be the same.

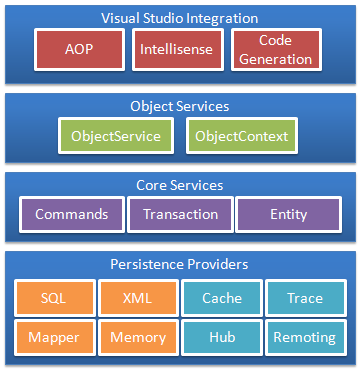
## Conclusion

Euss is provides a simple way to switching persistence logics. In this document you will find more in depth information about each Persistence Provider.

# Architecture

## Overview

Here is an overview of the Euss architecture:



Euss can be divided in three main functional layers:

* Object Services
* Core Services
* Persistence Engines

### Object Services

This layer is responsible of handling classes by transforming them into a more friendly data structure called Entity Relationship Attribute (ERA).

There are two main classes in this layer:

#### ObjectService (Evaluant.Uss.ObjectContext.ObjectService)

A threadsafe cache of compiled configuration for a single repository and a factory for ObjectContext.

#### ObjectContext (Evaluant.Uss. ObjectContext.ObjectContext)

A single-threaded, short-lived object representing a conversation between the application and the repository. It also holds a mandatory first-level cache of persistent objects (IdentityMap design pattern), used when navigating the object graph or looking up objects by identifier.

### Core Services

This layer realizes an abstraction between data sources and final client applications by providing a unified data model based on ERA elements:

#### Entity

It is the core element about the data to represent. They are described by a type. Entities are in general recognizable, concrete or abstract, like *a person*, *a place*, *a thing*.

#### Relationship

*Relationships* allow creating a link between two entities. They are in a general manner typed so that they can associate two same entities with different *Relationships*

#### Attribute

The attributes are associated with an entity to describe it. Each *Attribute* owns a name and its associated value.



This layer is also capable of decomposing any ERA graph into atomic commands (Units of Work) which will be executed by Persistence Engines in the scope of **transactions**.

### Persistence Engines

A Persistence Engine is a concrete class capable of executing atomic commands (Units of Work).

Each Persistence Engine is specifically for a repository. For instance there will be one for persisting data in XML files, another for Relational Database Systems, and so on and so forth. Using a common contracts infrastructure it is possible to enhance the list of available Persistence Engines.

## Contexts

An ObjectService is an expensive-to-create, threadsafe object intended to be shared by all application threads. It is created once, usually on application startup, from a configuration file.

An ObjectContext is an inexpensive, non-threadsafe object that should be used once, for a single request, a conversation, single unit of work, and then discarded.

From those statements you should retain that only one ObjectService instance should be created by application, and as many ObjectContext as threads using this unique ObjectService.

# Configuration

Because Euss is designed to operate in many different environments, there are a large number of configuration parameters.

## XML Configuration

### Usage

In order to simplify deployment and repository neutrality, Euss provides a way to load all the information for the serialization process from XML documents.

The convention expects you to store the configuration in a file named engines.config but you can decide to use any other one. Notice that using .config extensions will protect your configuration files from being downloaded in ASP.NET applications.

You can create an ObjectService instance directly from the configuration file:

ObjectService os = new ObjectService("engines.config");

Optionally you can specify which named configuration to use as an argument:

ObjectService os = new ObjectService("engines.config", "Sql");

This means the Persistence Engine named Sql in his file will be used. If the name argument is not provided, the one marked as Default will be used instead.

### Configuration Files

The simplest method is to use Visual Studio by adding a new item based on the template **Euss Engine Configuration File**. Doing like this will give you Intellisense integration within Visual Studio.

#### Sample XML configuration file

The following sample declares one Persistence Engine configuration section named Xml, using the class named Evaluant.Uss.Xml.XmlProvider as factory. Inside this tag the properties are defined, specifically for the type of the factory which is used here.

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

<PersistenceEngines

xmlns="<http://euss.evaluant.com/schemas/EngineConfiguration.xsd>"  
 DefaultEngine="Xml">

<!-- Persistence Engine declarations -->

<PersistenceEngine Name="Xml" Factory="Evaluant.Uss.Xml.XmlProvider">

<!-- Properties -->

<FileName>euss.xml</FileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

</PersistenceEngines>

For a list of all available properties, refer to the corresponding provider’s [configuration section](#_Xml_Provider_(Evaluant.Uss.Xml.XmlP) .

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <PersistenceEngines> | required | Encapsulates the file and references the current protocol standard. |
| DefaultEngine | required | Name of the Persistence Engine configuration section to load if none is specified. |
| <PersistenceEngine> | required | Parent tag for each Persistence Engine configuration. |
| Name | required | Local name. Used to reference this section. |
| Factory | optional | Name of the class to use. Allowed values:  Evaluant.Uss.Xml.XmlProvider  Evaluant.Uss.SqlMapper.SqlMapperProvider  Evaluant.Uss.Sql.SqlProvider  Evaluant.Uss.Memory.MemoryProvider  Evaluant.Uss.Cache.CacheProvider  Evaluant.Uss.Hub.HubProvider  Evaluant.Uss.Trace.TraceProvider  Evaluant.Uss.Remote.RemoteProvider  If not set, this represents a reference to another existing section in the same file. |

## Application Settings

The configuration information can be defined right inside the configuration system of the .NET framework. This can be either the Web.configfile or an App.config when.the application is not an ASP.NET application.

### Declaring the section

The first thing that has to be done is to declare the configuration section like this :

<configSections>

<section name="evaluant.uss"

type="Evaluant.Uss.Configuration.EussConfiguration, Evaluant.Uss"/>

</configSections>

This will inform the configuration manager of a section named evaluant.uss. This name is mandatory, you can’t use another one.

### Defining the parameters

Here is an example of a working configuration section for the Xml persistence engine.

<evaluant.uss>

<engines defaultEngine="wrigley">

<engine name="wrigley" factory="Evaluant.Uss.Xml.XmlProvider">

<add name="FileName" value="~/euss.xml"/>

<metadata>

<add type="model" value="~/demo.model.xml"/>

</metadata>

</engine>

</engines>

</evaluant.uss>

Engines are defined in the engines tags. Each of them can declare provider specific properties using an add tag.

Tip : The SqlProvider and the SqlMapperProvider can benefit from the existing connectionStrings section and link to the connection strings declared inside. This allows you to reuse the same connection strings among different tools. This can also be very useful to encrypt them.

<connectionStrings>

<add name="myDb" connectionString="Server=.;Database=uss;UID=sa;PWD="/>

</connectionStrings>

<evaluant.uss>

<engines defaultEngine="demo">

<engine name="wrigley" factory="Evaluant.Uss.SqlMapper.SqlMapperProvider" connectionStringName="myDb">

If you set the connection string provider name as the name of the engine, it would also take the right connection string.

<connectionStrings>

<add name="myDb" providerName="demo" connectionString="Server=.;Database=uss;UID=sa;PWD="/>

</connectionStrings>

<evaluant.uss>

<engines defaultEngine="demo">

<engine name="demo" factory="Evaluant.Uss.SqlMapper.SqlMapperProvider" />

For the SqlMapperProvider, you may provide a mapping file from a resource in an assembly :

<evaluant.uss>

<engines defaultEngine="demo">

<engine name="demo" factory="Evaluant.Uss.SqlMapper.SqlMapperProvider">

<add name="mapping" value="assembly:Evaluant.Uss.NLinq.Tests.Resources.mapping\_specific,Evaluant.Uss.NLinq.Tests" />

</engine>

</engines>

</evaluant.uss>

In that case, the mapping file is named *mapping\_specific*, it is registered in the resource file *Evaluant.Uss.NLinq.Tests.Resources*, in the assembly *Evaluant.Uss.NLinq.Tests*. You may also declare you mapping as an Embedded Resource file. In that case, the mapping value seen before points to the file *mapping\_specific*, in the folder Resources, in the project Evaluant.Uss.NLinq.Tests.

### Instantiating an ObjectService

Getting a new ObjectService instance using the default configuration inside the configuration sections is as simple as creating a new object using the default constructor.

ObjectService os = new ObjectService();

## Programmatic Configuration

Programmatic configuration is a way to define a provider’s configuration by code. This can be useful when the properties are defined dynamically and thus can’t be set in a static XML file.

The ObjectService class can be instantiated using an existing persistence engine object (IPersistenceEngine). This object can be configured by hand before getting initialized.

// Creates a new XmlProvider isntance

XmlProvider provider = new XmlProvider();

// <FileName>euss.xml</FileName>

provider.FileName = "euss.xml";

// <Metadata Type="assembly">ProgrammaticConfiguration</Metadata>

provider.RegisterMetaData(

MetaDataFactory.FromAssembly("ProgrammaticConfiguration")

);

// Initilizes the provider using the given configuration properties

provider.InitializeConfiguration();

ObjectService os = new ObjectService(provider);

Here you can see how a provider instance is created and configured. This example is the programmatic configuration corresponding to previous Xml samples.

## Logging

### Overview

Euss logs various messages depending on the used provider.

Logging is configured in application settings files. This means in web.config for ASP.NET application and App.config for the others.

The standard logging infrastructure provided by the .NET Framework is used. This way you can configure where the log should be written declaratively.

This sample show how to enable logging in order to see all executed SQL statements.

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

<configuration>

<system.diagnostics>

<switches>

<add name="Evaluant.Uss.SqlMapper.Sql" value="1"/>

</switches>

</system.diagnostics>

</configuration>

To enable SQL DDL statements logs you must set the switch named Evaluant.Uss.SqlMapper.Sql to 1.

For more information on TraceSwitch configuration in the Microsoft .NET Framework please refer to the official documentation (<http://msdn2.microsoft.com/en-us/library/system.diagnostics.traceswitch.aspx>).

The available trace switches are listed below:

|  |  |  |
| --- | --- | --- |
| Switch | Values | Description |
| Evaluant.Uss.SqlMapper.Sql | 0, 1 | If 1, logs every SQL DDL statement |
| Evaluant.Uss.OPath | 0, 1 | If 1, logs every OPath query sent by the application |

### Choosing the trace destination

You can also change the logs destination directly from the configuration file.

In the sample below, the trace is configured to be written in a text file.

<configuration>

<system.diagnostics>

<switches>

<add name="Evaluant.Uss.SqlMapper.Sql" value="1"/>

</switches>

<trace autoflush="true" indentsize="4">

<listeners>

<add name="myListener"

type="System.Diagnostics.TextWriterTraceListener"

initializeData="TextWriterOutput.log" />

<remove name="Default" />

</listeners>

</trace>

</system.diagnostics>

</configuration>

For more information on TraceListener configuration in the Microsoft .NET Framework please refer to the official documentation (<http://msdn2.microsoft.com/en-us/library/system.diagnostics.tracelistener.aspx>).

# Persistent Classes

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an e-commerce application). Not all instances of a persistent class are considered to be in the persistent state - an instance may instead be [transient](#_Transient) or [detached](#_Detached).

Euss needs these classes to follow some simple rules. Depending on the rules you comply with, performances and functionalities can differ somehow.

In Euss, there are fundamentally two ways to create those classes: by hand (Plain Old Classes) or by code generation. The differences reside in the fact that code generation needs some information on the classes to generate, in the form of an Xml file.

## Plain Old Classes

Plain old classes are the one that each programmer is used to creating every day. Here is such a class, compatible with Euss.

public class Customer

{

protected string id;

public string Id

{

get { return id; }

set { id = value; }

}

protected string name;

public string Name

{

get { return name; }

set { name = value; }

}

protected Address address;

public virtual Address Address

{

get { return address; }

set { address = value; }

}

protected IList<Order> orders = new List<Order>();

public IList<Order> Orders

{

get { return orders; }

}

}

### Implement a default constructor

The class Customer has a default constructor (by essence as no constructor is defined here). All persistent classes must have a default constructor (at least protected) so that Euss can instantiate them during loads.

### Provide an identifier property (optional)

Customer has a property called id. This property maps to the primary key of the type of repository used. This property has to be of type string (System.String) or int (System.Int32). It will be automatically converted if the underlying repository stores other types of values.

By default, if no Id property is found, Euss will look for one named with the name of the class followed by Id (e.g. PersonId for a class named Person).

The identifier property is strictly optional. You can leave it off and let Euss keep track of object identifiers internally. We do not recommend this, however.

### Use non-sealed classes

A central feature of Euss, proxies, depends upon the persistent class being non-sealed. Dynamically created sub-classes are used to handle Euss specific behavior like lazy-loading.

### Declare fields as protected (optional)

For the same reason as previously, fields can be declared as protected. In this case Euss is not using getters and setters to read and write the object states, but on the contrary accesses directly the fields. This allows you to add any necessary code to accessors and be sure only the final end user fires those methods.

### Declare property getter and setter on fields

Customer declares the properties setter and getter methods for all its persistent fields. Internally Euss recognizes the relationship between the names of a field and its property in the form of \_Foo, foo, and m\_Foo.

In .NET 3.5 you can define properties without members. This is handled by Euss too.

### Declare reference properties as virtual

Customer declares its Address property as virtual. This is necessary for Euss so as it can handle lazy-loading on “to-one” relationships.

Not to mark those properties as virtual can also be a mean to prevent lazy-loading to be used.

### Use IList for collections

Customers has a property named Orders declared as a generic list IList<Order>. Internally Euss is using a specific IList implementation for lazy-loading and optimized management of collections of objects.

You should also initialize this field with any implementation of IList so that your object is fully functional for the programmer.

You should never implement the setter for collections as no-one should be able to replace your implementation with another one. In a general manner, collections only need to bet read-accessible as they contain methods for their modification.

## Generating the Classes

### Model files

In order to describe which classes, property and relationships must be generates, Euss needs some Xml model files.

Euss accepts two types of Xml files:

* XMI (Xml Metadata Interchange) generated by UML modeling tools
* Euss Domain Model files created directly Visual Studio or any Xml file editor

If you intend to use XMI files, here is a list of the compatible tools:

* Objecteering
* PowerDesigner
* PowerAMC
* Rose
* Visio
* Together
* VisualParadigm
* Poseidon
* StarUML
* Enterprise Architect

### Euss Domain Model files and Visual Studio

You can create a new Euss Domain Model file from Visual Studio by clicking on the following menu items: **Project**, **Add New Item**, **Euss Domain Model File**.

Here is a sample model file:

<Model xmlns="http://euss.evaluant.com/schemas/GenerationModel.xsd">

<Package name="EussDemo.Domain">

<Class name="Customer">

<Property name="Name" type="System.String" />

</Class>

<Class name="Order">

<Property name="Amount" type="System.Int32" />

</Class>

<Relationship type="composition">

<Parent name="Customer" role="Customer" multiplicity="1" />

<Child name="Order" role="Orders" multiplicity="\*" />

</Relationship>

</Package>

</Model>

Each time you save this file the code generation process is launched and all the corresponding classes are generated automatically. If you configure Visual Studio to see hidden files you will be able to see them.

You can add your own elements in the generated classes by using partial classes. If you still use Visual Studio 2003 you can add you own code in dedicated regions without worrying about loosing them. They will be inserted during next generation.

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Model> | required | Encapsulates the file and references the current protocol standard. |
| <Package> | required | Encapsulates a set of Class and Reference declarations. |
| name | required | Name of the namespace the classes will b generated in. |
| <Class> | optional | Declares a Class. Encapsulates a set of Property declarations. |
| name | required | Local name of the class. |
| <Property> | optional | Declares a property. |
| name | required | Name of the property. |
| type | required | Type of the property. It must be a valid type in the targeted programming language. |
| <Reference> | optional | Declares a relationship between two classes. Declares a property. Encapsulates a Parent and a Child declarations. |
| type | required | Type of the reference.  Allowed values:  composition, aggregation |
| <Parent> | required | Declares the parent class. |
| name | required | Name of the parent class. |
| role | optional | Name of the parent property from the child. Currently not used during generation. |
| multiplicity | required | Multiplicity of the child-parent relationship.  Allowed values:  1, \* |
| <Child> | required | Declares he child class. |
| name | required | Name of the child class. |
| role | required | Name of the child property from the parent. |
| multiplicity | required | Multiplicity of the parent-child relationship.  Allowed values:  1, \* |

### Xmi files and Visual Studio

Euss can also handle Xmi files right inside Visual Studio. This can be done by adding he file to your project, and setting the **Custom Tool** property of the file (in the **Property Window**) to PersistenceManagerGenerator.

Once this is done, each time this file will change the code generation process will be launched.

### Using the command line utility

Euss includes a set of command line utilities to manage repeated tasks. One (domain.exe) is responsible for generating the persistent classes from an Xml model file. This utility takes either a Euss Domain Model file or an Xmi file directly.

Here is a sample command to generate the code from a model file:

domain.exe /model:crm.domain.xml

You can find details on its usage in the [corresponding section](#_domain.exe).

# Working With Objects

## Object States

Euss supports the following object states:

### Transient

An object is transient if it has just been instantiated using the new operator, and it is not associated with an ObjectContext. It has no persistent representation in the repository and no identifier value has been assigned. Transient instances will be destroyed by the garbage collector if the application doesn't hold a reference anymore. Use the ObjectContext to make an object persistent.

### Persistent

A persistent instance has a representation in the repository and an identifier value. It might just have been saved or loaded, however, it is by definition in the scope of an ObjectContext. Euss will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes.

### Detached

A detached instance is an object that has been persistent, but is not attached to an ObjectContext anymore. The reference to the object is still valid, of course, and the detached instance might even be modified in this state. A detached instance can be reattached to a new ObjectContext at a later point in time, making it (and all the modifications) persistent again using the Import() method.

## Managing Transactions

Transactions are demarcated in Euss using the methods BeginTransaction(), CommitTransaction() and RollbackTransaction() of the class ObjectContext.

The standard usage is to enclose any communication with the repository in a Try Catch clause like this:

try

{

oc.BeginTransaction();

oc.Serialize(p);

oc.CommitTransaction();

}

catch

{

// Handle failure here

return String.Empty;

}

There is no need to call RollbackTransaction() in the Catch section. Indeed, internally the underlying persistence engine takes care of managing connection states by itself. The RollbackTransaction() method should instead be seen as a *cancel current transaction* method that could be called in place of CommitTransaction() if needed.

## Making Objects Persistent

Newly instantiated instances of a persistent class are considered *transient* by Euss. We can make a transient instance *persistent* by using an ObjectContext and its Serialize() method.

Customer c = new Customer();

c.Name = "Customer One";

c.Orders.Add(new Order("Product One"));

oc.BeginTransaction();

oc.Serialize(c);

oc.CommitTransaction();

If the object you make persistent has associated objects (e.g. the Orders collection in the previous example), these objects are also made persistent automatically.

Multiple calls on the Serialize() method can be made during between a call to BeginTransaction() and CommitTransaction().

## Loading an Object

The LoadWithId<T>() methods of ObjectContext gives you a way to retrieve a persistent instance if you already know its identifier. LoadWithId<T>() takes a Type parameter and will load the state into a newly instantiated instance of that class, in persistent state.

Customer c = oc.LoadWithId<Customer>("123");

If the object with the specified identifier has already been loaded with this ObjectContext instance then no query will be sent to the repository but will be taken from the first level cache it embeds instead. It also ensures that multiple queries on the same objects return the same used reference in your application and prevents you from modifying some properties from different objects.

If you want to refresh some persistent objects you can call the method Clear() on ObjectContext and load this object again. The Clear() method will delete the first level cache and force reading the repository directly.

## Querying

If you don't know the identifiers of the objects you are looking for, you need a query. Euss supports an easy-to-use but powerful object oriented query language, *OPath*.

For queries requiring special parameters like paging and sorting a special Query object can be used.

### Executing queries

OPath queries are executed using the Load<T>() method of the class ObjectContext. This method returns a typed collection containing the objects corresponding to the query’s criteria.

IList<Customer> custs = oc.Load<Customer>(

typeof(Customer),

"[Name = 'Erik']" );

IList<Order> orders = oc.Load<Order>(

typeof(Customer),

"[Name = 'Erik'].Orders");

IList<Customer> buyers = oc.Load<Customer>(

typeof(Customer),

"[ exists(Orders[ id('CASDS')] ]");

The first parameter to define is the type of the class on which the query will be applied. There is no link between the returned type and the one given as the first parameter.

When a query is executed the result is generally a collection. This collection is strongly typed in .NET 2.0 using *generics* by setting the return type right after the method name like in Load<Customer>().

The LoadSingle<T>() method offers a shortcut if you know your query will only return a single object.

Customer c = oc.LoadSingle<Customer>(

typeof(Customer),

"[Name = 'Erik']");

If no object corresponds to the constraint the result is null.

### Scalar results

Queries may specify an aggregate on a property of a class. Aggregates are considered "scalar" results (and not entities in persistent state). Thus the result is a value type. The LoadScalar() method is dedicated to this function.

int count = (int)oc.LoadScalar("count(Domain:Customer)");

double avgAmount = (double)oc.LoadScalar("avg(Domain:Orders.Amount)");

The LoadScalar() uses pure OPath queries, by setting the name of classes using the fully specified construction. This means that the full name of the type is used, each namespace separated by a colon (:).

### Pagination

If you need to specify bounds upon your result set (the maximum number of objects you want to retrieve and / or the first row you want to retrieve) you should use the Query class:

Query query = new Query(typeof(Customer));

query.FirstResult = 50;

query.MaxResults = 10;

IList<Customer> pagedCustomers = oc.Load<Customer>(query);

The query will be optimized by the underlying persistence engine.

### Ordering results

The Query class is also used to order your result set.

Query query = new Query(typeof(Customer));

query.OrderBy.Add("Name");

IList<Customer> orderdCustomers = oc.Load<Customer>(query);

## LINQ

LINQ is a set of extensions to the .NET Framework that encompass language-integrated query, set, and transform operations. It extends C# and Visual Basic with native language syntax for queries and provides class libraries to take advantage of these capabilities.

Euss implements method extensions for LINQ, allowing the developer to use LINQ as a primary language for querying objects from any repository.

The LINQ runtime can only be used with Microsoft Visual Studio 2008.

### Referencing LINQ extension methods

In order to be able to use LINQ within Euss, the extension methods must be imported. This is done by adding the using keywork on them. In Euss they are declared in the namespace Evaluant.Uss.Linq.

using Evaluant.Uss.Linq;

### Executing queries

With LINQ, queries are expressed directly on the domain classes.

ObjectService os = new ObjectService();

ObjectContext oc = os.CreateObjectContext();

var customers = from Customer c in oc

where c.Name == "Erik"

select c;

foreach (Customer c in customers)

{

Console.WriteLine(c.Name);

}

This example iterates through any Customer with the name Erik.

The query is executed when the enumeration starts.

### Filtering on properties

Any operator can be used on properties. Moreover you can use additive methods on System.String like Contains(), StartsWith(), and EndsWith()

var customers = from Customer c in oc

where c.Name.StartsWith("A")

select c;

### Loading references

Applying a “join “ is done by navigating the references in the domain model.

var orders = from Customer c in oc

where c.Name.StartsWith("A")

from Order o in c.Orders

select o;

Two from clauses are applied to navigate a relationship.

### Constraining relationships

Several methods can be applied on relationships to constrain the queries.

The method Count() can be used to filter depending on the number of elements of the relationship. This method can also be used in conjunction with a predicate.

var customers = from Customer c in oc

where c.Orders.Count() > 0

select c;

This query returns any Customer with at least one Order.

var customers = from Customer c in oc

where c.Orders.Count(o => o.Description = "Screws") > 0

select c;

This one also applies a predicate on the Count().

You can also use the Contains() method to test the existence of a specific object inside a collection.

var customers = from Customer c in oc

where c.Orders.Contains(myOrder)

select c;

### Ordering results

Ordering is done using the orderby keyword.

var customers = from Customer c in oc

orderby c.Name

select c;

The ordering clauses can be mixed, and also descendant.

var customers = from Customer c in oc

orderby c.Name descending

select c;

### Paging results

Paging is done using two extension methods, Skip() and Take().

foreach (Customer c in customers.Skip(5).Take(10))

{

Console.WriteLine(c.Name);

}

Optionally, you can also use the First() method to only get the first result of a query, using an underlying optimized Euss query.

Customer c = (from Customer c in oc orderby c.Name descending select c).First();

This query will be executed when First() is called.

### Aggregates

In order to execute aggregation queries, Euss comes with a set of extension methods to apply on LINQ queries. They are Count(), Max(), Min() and Avg().

int custs = (from Customer c in oc

where c.Name.StartsWith("A")

select c).Count();

### Inferred loading

In order to optimize the queries, you can define which relationships will be used. Specifying those relationships is done using the Infer() method.

var customers = (from Customer c in oc

orderby c.Name

select c).Infer(c => c.Orders);

Multiple calls to Infer() can be joined for a single query.

### Applying projections

With the implementation of LINQ in Euss you can still use the “LINQ To Memory” implementation inside the select statements. This is also true for anonymous types.

var names = from Customer c in oc

orderby c.Name

select c.Name.ToUpper();

## Modifying persistent objects

Modifying a persistent object is simple as loading it, changing its properties and references, and finally calling the Serialize() method, like for making a transient object persistent.

Customer c = oc.LoadSingle<Customer>(typeof(Customer), "[Name = 'Erik']");

c.Name = "John";

c.Orders.Add(new Order("Product Two"));

oc.BeginTransaction();

oc.Serialize(c);

oc.CommitTransaction();

As you can see in this example whether modifications are applied to properties or references, they will be both persisted with a single call to Serialize().

If you call the Serialize() method on an unmodified instance, then the serialization engine will not execute any query. Moreover, only the modified attributes will be used during queries, for an optimized execution.

## Deleting persistent objects

ObjectContext.Delete() will remove an object's state from the database. Of course, your application might still hold a reference to a deleted object. It's best to think of Delete() as making a persistent instance transient.

Customer c = oc.LoadSingle<Customer>(typeof(Customer), "[Name = 'Erik']");

oc.BeginTransaction();

oc.Delete(c);

oc.CommitTransaction();

When deleting an object, all references to it and also from it will be removed from the data store. It is also possible to specify that direct references objects should be deleted as well. This information can be set using [metadata information](#_Data_about_data) during configuration of the persistence provider.

## Getting and setting the identifier

If you followed the recommendations on designing a persistent class, you can use the identifier property, commonly named Id. If not, you can still use the ObjectContext instance to access it with the methods GetId() and SetId().

## Using metadata

The word *metadata* can be used to describe two different concepts.

On one side it is simply *data about data*, more specifically information (data) about a particular content (data). On the other side this can stand for *data coming from a metamodel*.

In Euss both metadata concepts take an important part.

From time to time, this model is very useful to the application itself. For example, the application might use Euss's metadata to create model driven generic applications.

### Data about data

Metadata in the context of data about data is used by Euss to know in advance all the entities it will have to handle.

The whole set of metadata is made of the list of persistent classes, their properties and references. Internally they are described as *entities*, *relationships* and *attributes* (ERA). It also contains information on *multiplicity* and *composition*.

#### Metadata sources

Metadata can be registered in Euss directly from the configuration files, in Metadata tags.

There are three possible types of metadata sources:

* Assemblies
* Model files (the same used for [Code Generation](#_Generating_the_Classes))
* Metadata specific Xml files

If you intend to code plain old classes using the assembly declaration should be used. On the contrary if you decide to use code generation then using the same model file as the metadata source will be more adapted.

#### Registering metadata in configuration files

In the sample below these three methods are shown:

<Metadata Type="assembly">BasicApplication</Metadata>

<Metadata Type="model">crm.domain.xml</Metadata>

<Metadata Type="metadata">crm.metadata.xml</Metadata>

Tip : When specifying an assembly metadata, you may also filter the assembly for a specific namespace :

<Metadata Type="assembly">MyNamespace:BasicApplication</Metadata>

You may also specify multiple namespaces cascading them:

<Metadata Type="assembly">MyNamespace1:MyNamespace2:BasicApplication</Metadata>

#### Registering metadata programmatically

As shown in the section [Programmatic Configuration](#_Programmatic_Configuration), it is possible to add metadata information from different sources programmatically. During the persistence provider’s configuration you can use the method RegisterMetada() to add some metadata.

provider.RegisterMetaData(

MetaDataFactory.FromAssembly("ProgrammaticConfiguration")

);

In this example the metadata is taken from an existing assembly. In order to provide a common set of information coming from different sources the class MetaDataFactory can be used. You can also add those information directly without using the factory class, and instead use the classes TypeMetaData, PropertyMetaData and ReferenceMetaData.

#### Creating metadata definition files

Whatsoever, if you need to add information which is not comprised in the source you specified then it will be necessary to use a supplementary *metadata definition file*.

You can create this type of file using a template in Visual Studio, by clicking on **Project**, **Add, New Item**, **Euss Metadata File**.

Here is a sample metadata file:

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

<?evaluant-application progid="EUSS.MetaData"?>

<Model xmlns="http://euss.evaluant.com/schemas/MetaDataModel.xsd">

<Entity type="BasicApplication:Customer">

<Attribute name="Name" type="System.String" />

<Reference name="Address" type="BasicApplication:Address"

fromMany="false" toMany="false" composition="true" />

<Reference name="Orders" type="BasicApplication:Orders"

fromMany="false" toMany="true"/>

</Entity>

<Entity type="BasicApplication:Address">

<Attribute name="City" type="System.String" />

</Entity>

<Entity type="BasicApplication:Order">

<Attribute name="Amount" type="System.Int32" />

</Entity>

</Model>

Please notice that the names of the classes have their namespaces separated using a colon (:).

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Model> | required | Encapsulates the file and references the current protocol standard. Encapsulates a set of Entity declarations. |
| <Entity> | required | Declares an Entity (a class). Encapsulates a set of Attribute and Reference declarations. |
| type | required | Full type of the entity to declare. |
| ignore | optional | If true, the entity will be removed from discovered entities. |
| <Attribute> | optional | Declares an Attribute (property) in an Entity. |
| name | required | Name of the declared Attribute. |
| type | required | Type of the Attribute. It must be a valid type in the .NET CLI (Common Language Infrastructure). |
| inherit | optional | Full name of the base for this type. |
| ignore | optional | If true, the attribute will be ignored during serialization. |
| implement | optional | Comma separated list of all interfaces the type implements. It should contain the whole inheritance tree. |
| <Reference> | optional | Declares a relationship between two classes. Declares a property. Encapsulates a Parent and a Child declarations. |
| name | required | Name of the declared Reference. |
| type | required | Type of the reference.  Allowed values:  composition, aggregation |
| fromMany | optional | Whether the child Entity of the reference can be referenced by several other ones. Allowed values:  0, 1, false, true. By default it is “true”. |
| toMany | optional | Whether the parent Entity of the reference can reference several other ones. Allowed values:  0, 1, false, true. |
| composition | optional | Type of the reference.  Allowed values:  true, false. By default it is “false”. |
| ignore | optional | If true, the reference will be ignored during serialization. |

Note: It is possible to force a reference to be persisted as an attribute by adding a reference metadata with and ignore attribute, and an attribute metadata on this property. This will lead Euss to serialize the object using binary serialization.

#### Adding metadata using .NET attributes

Some metadata can be declared right inside the domain classes. The only drawback is that the assembly containing those classes will need a reference on Euss, adding some sort of dependency.

Still, here is the list of existing attribute classes:

| Class/Properties | Scope | Description |
| --- | --- | --- |
| **PersistentProperty** | Property | Encapsulates the file and references the current protocol standard. Encapsulates a set of Entity declarations. |
| FieldName |  | A string representing the member in the class that the property is encapsulating |
| Type |  | The member’s declaration type |
| Composition |  | **True** if this property is a relationship which should be seen as a composition (as opposed to an aggregation) |
| SerializeAsAttribute |  | True to force this property to be serialized as an attribute and not a relationship. Useful for transforming referenced objects using binary serialization. |
| **NotSerialized** | Class | Ignores the specified class in the metadata. This will cause Euss not to take it into account. |
| **PersistentIdAttribute** | Property | Set on a property to define it as the identifier of the class. By default, the property named *Id*o *ClassNameId* is used. |
| FieldName |  | A stringrepresenting the name of the member containing the identifier. |

### Metamodel

Behind the scene every object is converted into a higher level object in Euss, an Entity instance. Entity objects are the ones the persistence providers actually manipulate. They represent abstractions of data which are more manageable.

Together with the metadata they provide a repository independent data storage infrastructure. The *Object Services* layer is just a wrapper on it, adding the services to manipulate CLR classes.

IPersistenceProvider provider = XmlConfigLoader.LoadXmlConfig("engines.config");

IPersistenceEngine engine = provider.CreatePersistenceEngine();

Entity e = new Entity("Customer");

e["Name"] = "Customer One";

Transaction t = new Transaction();

t.Serialize(e);

t.Commit(engine);

In this example an Entity object of type *Customer* is created in the repository. This has exactly the same effect as using the Customer class we used previously, but here we don’t even need this class to exist.

Note that any functionnality demonstrated in the Object Services layer is somehow available in the metadata layer, like for instance *Querying*.

## ADO.NET Data Services

ADO.NET Data Services (also known as Project code name “Astoria”) consists of a combination of patterns and libraries that enables any data store to be exposed as a flexible data service, naturally integrating with the Web, that can be consumed by Web clients within a corporate network or across the Internet. ADO.NET Data Services uses URIs to point to pieces of data and simple, well-known formats to represent that data, such as JSON and ATOM/APP. This results in data being exposed to Web clients as a REST-style resource collection, addressable with URIs that agents can interact with using standard HTTP verbs such as GET, POST, or DELETE.

### Implementing a DataContext

A Data Context is a class describing all entity sets which can be exported. In Euss, a base Data Context implementation is provided, letting you focus on the exported entities only. This implementation also implements both IUpdatable and IExpandProvider.

To create a Data Context, inherit from Evaluant.Uss.DataServices.DataContext like this:

public class MyDataContext : Evaluant.Uss.DataServices.DataContext

{

private static ObjectService os = new ObjectService();

public MyDataContext():base(os.CreateObjectContext())

{

}

public IQueryable<Customer> Customers

{

get { return CreateIQueryable<Customer>(); }

}

public IQueryable<Address> Addresses

{

get { return CreateIQueryable<Address>(); }

}

}

This example uses a standard Euss configuration based in the App.Config file, yet you can use any kind of configuration instead. Each DataContext instance must own a dedicated ObjectContext instance.

Then the Data Service is exposed like this:

public class SampleDataService : DataService<MyDataContext>

{

public static void InitializeService(IDataServiceConfiguration config)

{

config.SetEntitySetAccessRule("\*", EntitySetRights.All);

config.SetServiceOperationAccessRule("\*", ServiceOperationRights.All);

}

}

### Notes on implementation

In order to provide entities to external services, your domain classes must have the following properties:

* Each class must have an identifier named ID. Otherwise ADO.NET Data Services will refuse to expose your entities. You can define another property by using the System.Data.Services.Client.DataServiceKey attribute on your classes.
* Each class must be marked as [Serializable].

# Euss in the Application Architecture

This section provides a sort of best practice in utilizing Euss in your applications. Even more if you are concerned in making your application independent from the ORM tool you use or you just want separate concerns.

## Domain Layer

The *domain layer* is a logical layer containing all the classes you will manipulate in your application. They are the *subjects* of your application.

The main concern with this layer is to make it completely unaware of any other framework, so that you can use another tool at any time without major impact on your application.

Physically this layer is made of a *Class Library* (or several) containing all your classes. You don’t need to add a reference to Euss from it.

In this project, we will add for instance classes like Person and Address.

Here is an example Person class:

public class Person

{

protected string id;

public string Id

{

get { return id; }

set { id = value; }

}

protected string firstname;

public string Firstname

{

get { return firstname; }

set { firstname = value; }

}

protected string lastname;

public string Lastname

{

get { return lastname; }

set { lastname = value; }

}

}

## Service Layer

The service layer is another logical layer. You will place all your interactions needed for the application. This layer need a reference to you domain layer and also Euss. Some more complex scenarios can separate this layer in several physical layers (assemblies) which would not all have a reference to Euss.

### Contracts

First we need to create an interface which will describe what our service will have to do, the *contract*. We call it IService.

public interface IService

{

/// <summary>

/// Creates the person.

/// </summary>

/// <param name="id">The id.</param>

/// <param name="firstname">The firstname.</param>

/// <param name="lastname">The lastname.</param>

/// <returns>The id if the operation was succcessful,

/// otherwise String.Empty</returns>

string CreatePerson(string id, string firstname, string lastname);

/// <summary>

/// Deletes the person.

/// </summary>

/// <param name="id">The id.</param>

/// <returns>True is the operation was successful</returns>

bool DeletePerson(string id);

/// <summary>

/// Gets all person.

/// </summary>

/// <returns></returns>

IList<Person> GetAllPerson();

/// <summary>

/// Gets all person.

/// </summary>

/// <param name="sortExpression">The sort expression.</param>

/// <param name="firstResult">The first result.</param>

/// <param name="maxResults">The max results.</param>

/// <returns></returns>

IList<Person> GetAllPerson(string sortExpression, int firstResult,

int maxResults);

/// <summary>

/// Gets the person by id.

/// </summary>

/// <param name="id">The id.</param>

/// <returns></returns>

Person GetPersonById(string id);

/// <summary>

/// Gets the person count.

/// </summary>

/// <returns></returns>

int GetPersonCount();

/// <summary>

/// Initializes the repository.

/// </summary>

void Initialize();

/// <summary>

/// Updates the person.

/// </summary>

/// <param name="id">The id.</param>

/// <param name="firstname">The firstname.</param>

/// <param name="lastname">The lastname.</param>

/// <returns>True is the operation was successful.</returns>

bool UpdatePerson(string id, string firstname, string lastname);

}

These are all the method we should provide from a service point of view. Actually you will add many other ones, this is just an example.

### Implementations

Then we create a class EussService implementing this interface. It will represent our concrete implementation for the service.

public class EussService : IService

We create a valued constructor taking an ObjectContext instance in parameter.

protected ObjectContext oc;

public EussService(ObjectContext oc)

{

this.oc = oc;

}

Then the implementation of the methods themselves.

public string CreatePerson(string id, string firstname, string lastname)

{

Person p = new Person();

p.Firstname = firstname;

p.Lastname = lastname;

try

{

oc.BeginTransaction();

oc.Serialize(p);

oc.CommitTransaction();

}

catch

{

return String.Empty;

}

return p.Id;

}

### Service Factory

Finally we create a *service factory* class in order to hide the implementation used.

public class ServiceFactory

{

protected static ObjectService os;

/// <summary>

/// Initializes the <see cref="ServiceFactory"/> class.

/// </summary>

static ServiceFactory()

{

// Determines whether the service is hosted in ASP.NET or not

string path = System.Web.Hosting.HostingEnvironment.IsHosted

? System.Web.Hosting.HostingEnvironment.MapPath("~/engines.config")

: "engines.config";

os = new ObjectService(path);

}

/// <summary>

/// Creates the service.

/// </summary>

/// <returns></returns>

public static IService CreateService()

{

return new EussService(os.CreateObjectContext());

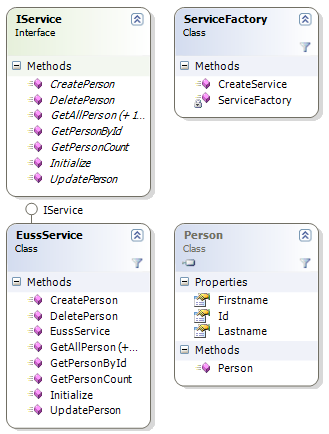
}

}

This class is statically initialized and detects whether it is hosted in the ASP.NET environment or not to select the correct path of the engines.config configuration file. The CreateService() method will return an IService instance, hiding the real EussService implementation.

## Applying the Architecture

Here is a partial view of the current architecture:

****

### Usage in ASP.NET

#### General usage

The key information in ASP.NET is that the ObjectService instance must be unique in the whole application domain. That’s why it is instantiated in the static constructor of the ServiceFactory.

On the contrary, ObjectContext should be instantiated in each context, here the ASP.NET request. For this reason it is advised to instantiate the IService as a protected member in the declaration of a **Web Form**.

IService service = ServiceFactory.CreateService();

Person p = null;

protected void Page\_Load(object sender, EventArgs e)

{

Person p = service.GetPersonById(Request["id"]);

}

#### Sample usage

Create a new **Web Application Project** and add a reference to both Application.Domain and Application.Services project.

First we add an Euss configuration file. You can refer to [Configuration](#_Configuration) to see how to do it.

<PersistenceEngines

xmlns="http://euss.evaluant.com/schemas/EngineConfiguration.xsd"

DefaultEngine="Xml">

<PersistenceEngine Name="Xml" Factory="Evaluant.Uss.Xml.XmlProvider">

<FileName>euss.xml</FileName>

<Metadata Type="assembly">Application.Domain</Metadata>

</PersistenceEngine>

</PersistenceEngines>

Here is the one used, a simple Xml file storage.

In a new **Web Form** named DisplayPerson.aspx, we add a GridView and use the data source wizard to setup a new ObjectDataSource.

Select the EussService as the service provider and the **Select Method**. Allow paging and sorting.

<asp:GridView ID="PersonGridView" AllowPaging="True"

AutoGenerateColumns="False"

DataSourceID="PersonObjectDataSource" AllowSorting="True"

runat="server" >

<Columns>

<asp:BoundField DataField="Id" HeaderText="Id"

SortExpression="Id" />

<asp:BoundField DataField="Firstname" HeaderText="Firstname"

SortExpression="Firstname" />

<asp:BoundField DataField="Lastname" HeaderText="Lastname"

SortExpression="Lastname" />

</Columns>

</asp:GridView>

On the ObjectDataSource, add a new event handler to ObjectCreating in order to setup how the service is instantiated.

protected void PersonObjectDataSource\_ObjectCreating(

object sender,

ObjectDataSourceEventArgs e)

{

e.ObjectInstance = Services.ServiceFactory.CreateService();

}

Here is the final code for the ObjectDataSource configuration.

<asp:ObjectDataSource ID="PersonObjectDataSource" runat="server"

TypeName="Application.Services.EussService"

DeleteMethod="DeletePerson"

InsertMethod="CreatePerson"

SelectMethod="GetAllPerson"

UpdateMethod="UpdatePerson"

SortParameterName="sortExpression"

OnObjectCreating="PersonObjectDataSource\_ObjectCreating">

<SelectParameters>

<asp:Parameter Name="sortExpression" Type="String" />

<asp:Parameter Name="firstResult" Type="Int32" />

<asp:Parameter Name="maxResults" Type="Int32" />

</SelectParameters>

</asp:ObjectDataSource>

# Object-Relational Mapping

## Overview

From Wikipedia: “***Object-Relational mapping*** *(aka* ***O/RM****,* ***ORM****, and* ***O/R mapping****) is a programming technique for converting data between incompatible type systems in databases and Object-oriented programming languages. This creates, in effect, a "virtual object database" which can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.*”

## Configuration

Here is a sample configuration file for this provider:

<PersistenceEngine Name="Sql" Factory="Evaluant.Uss.SqlMapper.SqlMapperProvider">

<ConnectionString>

Server=.;Database=euss;UID=sa;PWD=

</ConnectionString>

<Driver>Evaluant.Uss.SqlMapper.MsSqlDriver</Driver>

<Dialect>Evaluant.Uss.SqlMapper.MsSqlDialect</Dialect>

<MappingFileName>domain.eum.xml</MappingFileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <ConnectionString> | required | The connection string representing the database to use. The syntax depends on the Driver. |
| <Driver> | required | The .NET driver used to communicate with the database. Allowed values:  Evaluant.Uss.SqlMapper.MsSqlDriver  Evaluant.Uss.SqlMapper.MsOracleDriver  Evaluant.Uss.SqlMapper.OracleDPDriver  Evaluant.Uss.SqlMapper.Db2Driver  Evaluant.Uss.SqlMapper.MySqlDriver  Evaluant.Uss.SqlMapper.OdbcDriver  Evaluant.Uss.SqlMapper.OleDbDriver  Evaluant.Uss.SqlMapper.SQLiteDriver |
| <Dialect> | required | Represents the logic used to create the SQL which will be sent to the database. Allowed values:  Evaluant.Uss.SqlMapper.MsSqlDialect  Evaluant.Uss.SqlMapper.OracleDialect  Evaluant.Uss.SqlMapper.AccessDialect  Evaluant.Uss.SqlMapper.MySqlDialect  Evaluant.Uss.SqlMapper.SQLiteDialect  Evaluant.Uss.SqlMapper.Db2Dialect |
| <MappingFileName> | required | The location of the file containing the mapping information. |
| <TablePrefix> | optional | If specified, defines the table prefix to add to each table name. |
| <MappingStringSize> | optional | If specified, defines the size of the ValueString field for the Sql provider in the Attribute table |
| <Options> | optional | A space separated list of options. Allowed values:   * INSENSITIVE\_LIKE When used with OracleDialect, Euss will use case insensitive LIKE clauses. |

### Connections strings

Depending on the driver you use, the connection string will differ. Here are some example of connection strings you can use for each Euss driver. This website (<http://www.connectionstrings.com/>) contains a complete list of connection strings examples for each driver.

Here is the correspondence between Euss drivers and their real classes:

| Euss Driver | .NET Driver |
| --- | --- |
| MsSqlDriver | System.Data.SqlClient |
| MsOracleDriver | System.Data.OracleClient |
| MySqlDriver | MySql.Data.MySqlClient |
| OdbcDriver | System.Data.Odbc |
| OleDbDriver | System.Data.OleDb |
| SQLiteDriver | System.Data.SQLite |
| OracleDPDriver | Oracle.Data |
| Db2Driver | BM.Data.DB2.iSeries |

## Generating the mapping file

Euss can generate a specific mapping file using your model metadata. In most cases it will be sufficient for new projects. If you are using an existing database you will need to create it by yourself.

To have more information on how to generate this file refer to the migrate.exe command line utility documentation. You can also read the sample at the beginning of this document.

## Mapping Declaration

Here is a sample mapping file:

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

<Mapping

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<Entity type="BasicApplication:Customer" table="Customer">

<Id field="CustomerId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name"

db-type="String" size="50" not-null="false" />

<Attribute name="Address" field="Address"

db-type="String" size="50" not-null="false" />

</Entity>

</Mapping>

### Entity mapping

You may declare the mapping for a persistent class using the Entity element:

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Entity> | required | Declares the mapping for an Entity (a class). Encapsulates a set of Id, Attribute and Reference mapping declarations.  Example: BasicApplication:Customer |
| type | required | Full type of the entity to map. |
| table | required | Name of the database table. |
| copyNode | optional | Full type of an entity mapping. Its mapping properties will be copied in the current node. |
| discriminator-field | optional | If set, defines the column in the table which will contain the entity’s type when several ones can be stored in the same table (c.f. *One Table per Hierarchy* mapping in the Inheritance Mapping section). If set, *discriminator-value* is mandatory. |
| discriminator-value | optional | If set, defines which value will be stored in the column defined in the *discriminator-field* attribute for the current entity. If not, the full type name of the current will be used. |

### Identifier mapping

Mapped entities *must* declare the primary key column of the database table. The <Id> element defines the mapping to the primary key column:

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Id> | required | Declares the mapping for the primary key. Encapsulates a Generator declaration. |
| Field | required | Name of the column to use as a primary key for the current table. |
| <Generator> | required | Declares the primary key strategy for the current table. Encapsulates a set of Property declarations. |
| Name | required | Name of the strategy to use. Allowed values:  guid  native  assigned  inherited  business |
| <Property> | optional | Declares a property for the current Generator. |
| Name | required | Name of the property to declare. |

#### guid

Generates a *guid* value at runtime. The value is mapped to a variable char column type of 36 characters.

<Id field="PersonId" >

<Generator name="guid" />

</Id>

In this example, the technical identifier of the entity is copied in a variable char column type of 36 characters.

#### native

Use the *identity* (or *sequence*) value generated by the database.

Here is the list of available properties for this generator:

| Property |  | Description |
| --- | --- | --- |
| dbType | required | Euss dbType for the column containing the primary key. (c.f. dbType list) |
| seed | optional | First value in the table. |
| increment | optional | Value to increment the id with for each new entity in the table |

In the example below, the class Person contains a property Id of type Int32. Though you can still use a string property.

public class Person

{

protected int id;

public int Id

{

get { return id; }

set { id = value; }

}

}

And a sample corresponding mapping.

<Id field="PersonId">

<Generator name="native">

<Property name="dbType">Int32</Property>

<Property name="seed">2</Property>

</Generator>

</Id>

#### assigned

Let the application to assign an identifier to the object before Serialize() is called. This can be used is you want to be able to map the default identifier property to a special field type.

Here is the list of available properties for this generator:

| Property |  | Description |
| --- | --- | --- |
| dbType | required | Euss dbType for the column containing the primary key. (c.f. [dbType list](#_Database_Types_(dbtype))) |
| size | optional | Size of the column if the dbType needs one. |

<Id field="name">

<Generator name="assigned">

<Property name="dbType">String</Property>

<Property name="size">50</Property>

</Generator>

</Id>

In this example, the default identifier value is mapped to a variable length char field of 50 characters.

#### inherited

In the case of an entity inherits from another one, the inherited generator will use the same identifier mapping as the one declared for the parent entity mapping. This is used in *One Table per Concrete Class* mapping strategy to link multiple tables if the attributes of one entity are spanned over several tables.

Example:

<Entity type="Domain:Person" table="Person">

<Id field="PersonId">

<Generator name="guid" />

</Id>

<Attribute name="Name" db-type="String"/>

</Entity>

<Entity type="Domain:Employee" table="Employee">

<Id field="PersonId">

<Generator name="inherited" />

</Id>

<Attribute name="Salary" db-type="Int32"/>

</Entity>

#### business

Using this generator, the identifier value will be taken from a different attribute than the default identifier property. This is also used to define composite identifiers.

<Entity type="NUnit:Mapping:IdBusiness:Person" table="Person">

<Id field="name">

<Generator name="business" />

</Id>

<Attribute name="Name" field="name" db-type="String" size="50" />

<Attribute name="Age" field="age" db-type="Int32" />

</Entity>

In the example above, the default identifier is ignored, and the property named Name is use instead.

### Attribute mapping

You may declare the mapping for the attributes using the Attribute element:

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Attribute> | required | Declares the mapping for an Attribute. |
| name | required | Name of the attribute to declare. |
| field | optional | The name of the mapped database table column. The default value is the attribute’s name. |
| db-type | required | Euss database type identifier. (c.f. [dbType list](#_Database_Types_(dbtype))) |
| size | optional | Maximum length of the column. |
| scale | optional | Numerical column resolution. |
| precision | optional | Numerical column precision. |
| table | optional | Name of the table, if the attribute must be stored in another one. |
| parentField | optional | If *table* is set, defines the foreign key column name containing the link with the other table. |
| not-null | optional | Whether the column can’t be null. Allowed values:  true, false |
| default-value | optional | If not-null is set, contains the default value of the column. |

In Euss mapping files, database types (dbType) are used to define the column types with the same values whatever the RDBMS is. This allows mapping files to be database neutral, and make them reusable.

#### Database Types (db-type)

Here is the list of all available database types in Euss and their corresponding mapping in different RDBMS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| .NET type*[[1]](#footnote-1)* | db-type | Access | SQLServer | Oracle | MySQL | SQLite | DB2 |
| System.Byte[] | Binary | BINARY | IMAGE | BINARY | BLOB | BLOB | BLOB |
| System.Boolean | Boolean | BIT | BIT | NUMBER(1) | TINYINT | INTEGER | CHAR FOR BIT DATA |
| System.Byte | Byte | TINYINT | TINYINT | NUMBER(3) | TINYINT UNSIGNED | INTEGER | CHAR FOR BIT DATA/BLOB |
| *System.Decimal* | Currency  or Decimal(14, 28) | MONEY | MONEY | NUMBER(19,1) | MONEY | NUMERIC | DECIMAL |
| *System.DateTime* | Date | DATETIME | DATETIME | DATE | DATE | TIME | DATE |
| System.DateTime | DateTime | DATETIME | DATETIME | DATE | DATETIME | TIME | DATETIME |
| System.Decimal | Decimal | DECIMAL | DECIMAL | NUMBER | NUMERIC | NUMERIC | FLOAT |
| System.Double | Double | DOUBLE | DOUBLE | DOUBLE PRECISION | DOUBLE | NUMERIC | DOUBLE PRECISION |
| System.Guid | Guid | UNIQUEIDENTIFIER | UNIQUEIDENTIFIER | NVARCHAR2(36) | VARCHAR | UNIQUEIDENTIFIER | CHAR |
| System.Int16 | Int16 | SMALLINT | SMALLINT | NUMBER(5) | SMALLINT | INTEGER | SMALLINT |
| System.Int32 | Int32 | INTEGER | INT | NUMBER(10) | INTEGER | INTEGER | INTEGER |
| System.Int64 | Int64 | TEXT | BIGINT | NUMBER(20) | BIGINT | INTEGER | BIGINT |
| System.Object | Object | IMAGE | SQL\_VARIANT | BLOB | BLOB | BLOB | BLOB |
| System.SByte | SByte |  | TINYINT | NUMBER(3) |  | INTEGER |  |
| System.Single | Single | REAL | REAL | FLOAT | FLOAT | NUMERIC | FLOAT |
| Sytem.String[[2]](#footnote-2) | String | NVARCHAR | NVARCHAR/NTEXT | NVARCHAR/CLOB | NATIONAL VARCHAR /NATIONAL TEXT | TEXT | VARCHAR/CLOB |
| *Sytem.String* | AnsiString | VARCHAR | VARCHAR/TEXT | VARCHAR/CLOB | VARCHAR/TEXT | TEXT | VARCHAR/CLOB |
| *Sytem.String* | StringFixedLength | NCHAR | NCHAR | NCHAR | NATIONAL VARCHAR | TEXT | CHAR |
| *Sytem.String* | AnsiStringFixedLength | CHAR | CHAR | CHAR | CHAR | TEXT | CHAR |
| System.Timespan | Time |  |  | DATETIME | TIME | TIME | TIME |
| System.UInt16 | UInt16 |  | SMALLINT | NUMBER(5) |  | INTEGER |  |
| System.UInt32 | UInt32 |  | INT | NUMBER(10) |  | INTEGER |  |
| System.UInt64 | UInt64 |  | BIGINT | NUMBER(20) |  | INTEGER |  |
| *Sytem.Decimal* | VarNumeric |  |  | NUMBER |  | NUMERIC |  |

### Reference mapping

You may declare the mapping for the references using the Reference element:

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Reference> | required | Declares the mapping for a Reference. Encapsulates Rule mapping declarations. |
| name | required | Name of the reference to map. |
| entityChild | required | Full type of the entity the reference is actually linking. |
| discriminator | optional | Column name containing the nameof the reference in case the table handles several references. |
| <Rule> | required | Declares a Rule mapping. |
| parentField | optional | The column name to use as a foreign key in the table of the parent entity. |
| parentTable | optional | The table containing the *parentField* (if there can be an ambiguity). |
| childField | optional | The column name to use as a foreign key in the table of the referenced entity. |
| childTable | optional | The table containing the *childField* (if there can be an ambiguity). |
| constraint | optional | Name of the constraint to create in the database. |

## Mapping Relationships

### One-To-One relationships

An association to another persistent class is defined using a single rule. The relational model is a many-to-one association: a foreign key in one table is referencing the primary key column of the target table.

<Mapping>

<Entity type="Domain:Customer" table="Customer">

<Id field="CustomerId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String" size="50" />

<Reference name="Address" entityChild="Domain:Address">

<Rule parentField="FK\_AddressId"

childTable="Address"

childField="FK\_CustomerId" />

</Reference>

</Entity>

<Entity type="Domain:Address" table="Address">

<Id field="AddressId">

<Generator name="guid" />

</Id>

<Attribute name="City" field="City" db-type="String" size="50" />

</Entity>

</Mapping>

In this example the table Customer contains a foreign key FK\_AddressId referencing the Address entity it is holding.

### One-To-Many relationships

An ordinary association to another persistent class is declared using a single rule. The relational model is a many-to-one association: a foreign key in one table is referencing the primary key column of the target table.

<Mapping>

<Entity type="Domain:Customer" table="Customer">

<Id field="CustomerId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String" size="50" />

<Reference name="Orders" entityChild="Domain:Order">

<Rule parentField="CustomerId"

childTable="Order"

childField="FK\_CustomerId" />

</Reference>

</Entity>

<Entity type="Domain:Order" table="Order">

<Id field="OrderId">

<Generator name="guid" />

</Id>

<Attribute name="Amount" field="Amount" db-type="Int32" />

</Entity>

</Mapping>

In this example the table Order contains a foreign key FK\_CustomerId referencing the Customer entity holding it.

### Many-To-Many relationships

Many-To-Many relationships are declared using two rule declarations. The relational model is made using one *index table* and two associations.

<Mapping>

<Entity type="Domain:Customer" table="Customer">

<Id field="CustomerId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String" size="50" />

<Reference name="Addresses" entityChild="Domain:Address">

<Rule parentField="CustomerId"

childTable="CustomerAddress"

childField="FK\_CustomerId" />

<Rule parentField="FK\_AddressId"

childTable="Address"

childField="AddressId" />

</Reference>

</Entity>

<Entity type="Domain:Address" table="Address">

<Id field="AddressId">

<Generator name="guid" />

</Id>

<Attribute name="City" field="City" db-type="String" size="50" />

</Entity>

</Mapping>

In this example the table CustomerAddress contains two foreign keys: FK\_CustomerId and FK\_AddressId.

## Inheritance mapping

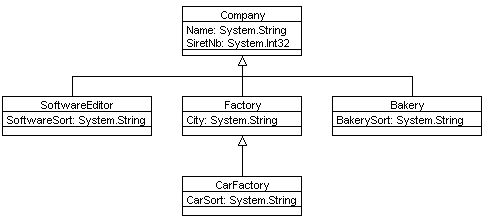
### The three strategies

Euss supports all the three inheritance mapping strategies:

* Table per class hierarchy
* Table per subclass
* Table per concrete class

### Table per class hierarchy

This strategy maps all the classes of one hierarchy is a single table. To handle to correct type of a row, a discriminator column is used.



In this example, a single Company table is created.

<Entity type="Domain:Company" table="Company" discriminator-field="Type"

discriminator-value="Company">

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String"/>

<Attribute name="SiretNb" field="SiretNb" db-type="Int32"/>

</Entity>

<Entity type="Domain:Factory" table="Company"

discriminator-field="Type"

discriminator-value="Factory" >

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="City" field="City " db-type="String"/>

</Entity>

<Entity type="Domain:CarFactory" table="Company"

discriminator-field="Type"

discriminator-value="CarFactory" >

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="CarSort" field="CarSort" db-type="String"/>

</Entity>

<Entity type="Domain:Bakery" table="Company"

discriminator-field="Type"

discriminator-value="Bakery" >

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="BakerySort" field="BakerySort" db-type="String"/>

</Entity>

<Entity type="Domain:SoftwareEditor" table="Company"

discriminator-field="Type"

discriminator-value="SoftwareEditor" >

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="SoftwareSort" field="SoftwareSort" db-type="String"/>

</Entity>

This first entity mapping declaration maps the Company class and the Company table. We use a discriminator (Type) in order to store several Companyentities from different types into the same table.

All other entity mappings jus need to declare the additional attributes they can handle.

### Table per subclass

A table per subclass mapping would look like:

<Entity type="Domain:Company" table="Company">

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String"/>

<Attribute name="SiretNb" field="SiretNb" db-type="Int32"/>

</Entity>

<Entity type="Domain:Factory" table="Factory">

<Id field="CompanyId">

<Generator name="inherited" />

</Id>

<Attribute name="City" field=" City " db-type="String"/>

</Entity>

<Entity type="Domain:CarFactory" table="CarFactory" >

<Id field="CompanyId">

<Generator name="inherited" />

</Id>

<Attribute name="CarSort" field="CarSort" db-type="String"/>

</Entity>

Four tables are required. The three subclass tables have primary key associations to the super class table (so the relational model is actually a one-to-one association). This association is done automatically using the inherited generator strategy.

Here is the result in the database for several instances from different types:

#### Company table



#### Bakery table



#### Factory table



#### CarFactory table



#### SoftwareFactory table



### Table per concrete class

With this strategy, each table defines columns for all properties of the class, including inherited properties.

This is done by using the copyNode attribute on entity mappings.

<Entity type="Domain:Company" table="Company">

<Id field="CompanyId">

<Generator name="guid" />

</Id>

<Attribute name="Name" field="Name" db-type="String"/>

<Attribute name="SiretNb" field="SiretNb" db-type="Int32"/>

</Entity>

<Entity type="Domain:Factory" table="Factory" copyNode="Domain:Company">

<Id field="FactoryId">

<Generator name="guid" />

</Id>

<Attribute name="City" field=" City " db-type="String"/>

</Entity>

<Entity type="Domain:CarFactory" table="CarFactory" copyNode="Domain:Factory">

<Id field="CarFactoryId">

<Generator name="guid" />

</Id>

<Attribute name="CarSort" field="CarSort" db-type="String"/>

</Entity>

# Other Storage Providers

## Xml Provider (Evaluant.Uss.Xml.XmlProvider)

This provider is used to persist data in a simple Xml file. It is helpful to start new projects as there is no configuration depending on the class you will try to persist.

#### Configuration

Here is a sample configuration file for this provider:

<PersistenceEngine Name="Xml" Factory="Evaluant.Uss.Xml.XmlProvider">

<FileName>euss.xml</FileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <FileName> | required | The filename of the Xml file to use as a data store. |

## Sql Provider (Evaluant.Uss.Sql.SqlProvider)

Compared to Xml data stores, Relational Database Management Systems (RDBMS) are transactional. The SqlProvider can use any RDBMS to store some data, but it is special in the database schema it uses.

This provider, as for the XmlProvider, is generic and thus can store any domain model’s data without requiring any configuration. Actually it creates three tables representing a metamodel.

#### Configuration

Here is a sample configuration file for this provider:

<PersistenceEngine Name="Sql" Factory="Evaluant.Uss.Sql.SqlProvider">

<ConnectionString>

Server=.;Database=euss;UID=sa;PWD=

</ConnectionString>

<Driver>Evaluant.Uss.SqlMapper.MsSqlDriver</Driver>

<Dialect>Evaluant.Uss.SqlMapper.MsSqlDialect</Dialect>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <ConnectionString> | required | The connection string representing the database to use. The syntax depends on the Driver. |
| <Driver> | required | The .NET driver used to communicate with the database. Allowed values:  Evaluant.Uss.SqlMapper.MsSqlDriver  Evaluant.Uss.SqlMapper.MsOracleDriver  Evaluant.Uss.SqlMapper.OracleDPDriver  Evaluant.Uss.SqlMapper.Db2Driver  Evaluant.Uss.SqlMapper.MySqlDriver  Evaluant.Uss.SqlMapper.OdbcDriver  Evaluant.Uss.SqlMapper.OleDbDriver  Evaluant.Uss.SqlMapper.SQLiteDriver |
| <Dialect> | required | Represents the logic used to create the SQL which will be sent to the database. Allowed values:  Evaluant.Uss.SqlMapper.MsSqlDialect  Evaluant.Uss.SqlMapper.OracleDialect  Evaluant.Uss.SqlMapper.Db2Dialect  Evaluant.Uss.SqlMapper.AccessDialect  Evaluant.Uss.SqlMapper.MySqlDialect  Evaluant.Uss.SqlMapper.SQLiteDialect |

## Memory Provider (Evaluant.Uss.Memory.MemoryProvider)

This provider is able to use the local memory to store data. This technique is commonly named IMDB (In Memory Database) or *prevalence*. The objective is to provide a very fast data store.

The MemoryProvider is also capable of using a delegate persistence engine to act as a persistent storage. It will then proxy this delegate persistence engine and respond to load queries in its place. During serialization both engines will register the modification applied on the objects.

#### Configuration

Here is a sample configuration referencing the already existing persistence engine configuration section name Xml:

<PersistenceEngine Name="Memory" Factory="Evaluant.Uss.Memory.MemoryProvider">

<Delegator>

<PersistenceEngine Name="Xml" />

</Delegator>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

You can also declare a new engine directly inside the Delegator tag like this:

<PersistenceEngine Name="Memory" Factory="Evaluant.Uss.Memory.MemoryProvider">

<Delegator>

<PersistenceEngine Factory="Evaluant.Uss.Xml.XmlProvider">

<FileName>euss.xml</FileName>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

</Delegator>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

#### XML tag definitions

The available XML tags are described below.

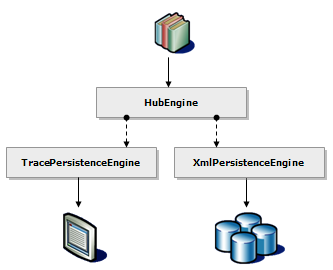
| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Delegator> | optional | Declares a new PersistenceEngine section or references an existing one. If it is omitted it will work as an autonomous persistence engine. |

# Infrastructure Providers

Those providers behave like any other one but are designed to provide infrastructure added value.

## Hub Provider (Evaluant.Uss.Hub.HubProvider)

This provider is capable of duplicating any message to several other providers. With this technique you can persist data in multiple repositories at the same time.



To set the data loading strategy you can define which provider will handle it by default.

#### Configuration

Here is a sample configuration referencing the already existing persistence engine configuration sections named Xml and Sql:

<PersistenceEngine Name="Hub" Factory="Evaluant.Uss.Hub.HubProvider">

<Delegators>

<PersistenceEngine Name="Xml" />

<PersistenceEngine Name="Sql" />

</Delegators>

<DefaultEngineIndex>0</DefaultEngineIndex>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

In this example the engine named Xml will be used for reading data.

#### XML tag definitions

The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Delegators> | required | Declares a the PersistenceEngine sections or references existing ones. |
| <DefaultEngineIndex> | optional | Defines the index of the PersistenceEngine declaration to use a the reader. |

## Cache Provider (Evaluant.Uss.Cache.CacheProvider)

This provider acts as a *second level cache* provider.

Instead of storing all the data into the memory like with the MemoryProvider, this engine stores only the most used data. Thus you keep an efficient data storage system which needs less memory than with the MemoryProvider. The drawback is that all the data isn't available from the memory and the queries can’t be evaluated in memory but have to be sent to an underlying persistence engine.

This engine shouldn’t be used in a client-server infrastructure on the client side. As any cache server, the cached information should be served on a central node to ensure data synchronization.

#### Configuration

<PersistenceEngine Name="Cache" Factory="Evaluant.Uss.Cache.CacheProvider">

<Delegator>

<PersistenceEngine Name="Xml" />

</Delegator>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

This example is wrapping the persistence engine named Xml to enhance its performances.

#### XML tag definitions

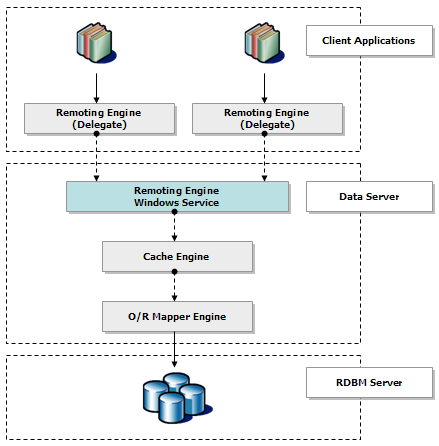
The available XML tags are described below.

| Tag/Attribute |  | Description |
| --- | --- | --- |
| <Delegator> | required | Declares a new PersistenceEngine section or references an existing one. |

## Remoting Provider (Evaluant.Uss.Remoting.RemotingProvider)

With a RemotingProvider configuration, all clients can share the same persitence engine on the server side. For instance, the server application can be configured with a cache engine which will be shared among all clients. This provides a server centric behavior to any repository. Euss is shipped with a host Console application and a Windows Service making use of this engine.

On the client side, the configuration file needs only the location of this service. The real repository is only described on the server side.



This schema describes a typical multi-tier architecture with a shared cache engine in front of a mapped relational database.

#### Client Configuration

<PersistenceEngine Name="Remoting" Factory="Evaluant.Uss.Remoting.RemoteProvider">

<Host>localhost</Host>

<Port>8085</Port>

<Metadata Type="assembly">BasicApplication</Metadata>

</PersistenceEngine>

This example configures the client application to use a remote persistence engine which services runs on he machine localhost on the port 8085.

#### Server Configuration

The serever can be hosted in a console application or in a windows service. Those executables can be found in the installation directory of Euss.

The server will search the file named engines.config and use the default provider.

# OPath

OPath is a repository neutral filtering language for querying objects in Euss. As its name states it is widely inspired by the *XPath* syntax.

An OPath query is simply a starting type definition followed by constraints and relationships.

## Case Sensitivity

Everything in an OPath query is case sensitive, from keywords to domain elements.

## Overview

### Select all clause

The OPath to select all the entities of the class Domain.Person is the following one:

Domain:Person

This OPath query is generated inside an ObjectContext by using this code :

oc.Load<Customer>(typeof(Customer));

### Selecting references

To select a set of referenced entities, you shall use the dot (.) between the first class name and the name of its reference.

Domain:Person.Address

This will load all the Address entities attached to any Person. Using an ObjectContext is programmed like this:

oc.Load<Address>(typeof(Person), ".Address");

### Constraints

Adding constraints to a path is done using brackets.

oc.Load<Person>(typeof(Person), "[Name = 'Erik']");

This will load all Person entities with a Name property equal to 'Erik'.

Those constraints can be applied everywhere in the query path.

oc.Load<Address>(typeof(Person), "[Name = 'Erik'].Address[City = 'Paris']");

Which loads all Address entities with a City equal to 'Paris' and belonging to all Person with a Name equal to 'Erik'.

### Referring to identifier property

To access the identifier in OPath queries the function id() must be used.

oc.Load<Address>(typeof(Person), "[id('123')].Address");

which is equivalent to

Domain:Person[id('123’)].Address

The result is all the Address entities belonging to the Person with an id of '123'.

### Filtering on the existence of a reference

The function exists() allows a query to filter elements depending whether they have some relationships or not.

oc.Load<Person>(typeof(Person), "[ exists( Address[City = 'Paris'] ) ]");

This wil return all Person entities having at least one Address with City equal to 'Paris'.

## Logical Expressions

### Operators

You can apply some operators into a *logical expression*. They are the following:

|  |  |
| --- | --- |
| Unary operators | Description |
| . | Used to separate several elements (items or attributes)  Example: Person.Partners. |
| [] | Group several logical expressions to constraint an element  [Name = 'Moos' or Name='Litamer'] |

|  |  |
| --- | --- |
| Unary operators | Description |
| not | Not  [not (Active)] |

|  |  |
| --- | --- |
| Binary operators | Description |
| ^ | Begins With  [Firstname ^ 'n'] |
| ~ | Contains  [Firstname ~ 'e'] |
| $ | Ends With  [Firstname $ 's'] |

|  |  |
| --- | --- |
| Comparison operators | Description |
| = | Equal  [Name = 'Moreno'] |
| != | Not equal  [Name != 'Leary'] |
| <= | Lesser or equal  [Value <= 10] |
| >= | Greater or equal  [Value >= -2] |
| < | Lesser  [Value < 101.5] |
| > | Greater  [Value > 5] |

|  |  |
| --- | --- |
| Additive operators | Description |
| - | Minus  [Value - 10 = 0] |
| + | Plus  [Value + 52.7 = 62] |
| % | Modulo  [Value % 2 = 1] |
| \* | Times  [value \* 2 = 10] |
| / | Div  [Value / 2 = 17.5] |

|  |  |
| --- | --- |
| Conditional operators | Description |
| or | [Name = 'King' or Name = 'Greaves] |
| and | [Value = 10 and Name = 'King'] |

### Functions

Several commands are available to make logical expressions.

|  |  |
| --- | --- |
| Command | Description |
| id(*list*) | *True* if the object’s identifier is among the *list*. The list is made of comma separated string values.  [id('xyz', '12')] |
| max(*path*) | The maximum value of any element represented by the *path.* The *path* must end with an attribute.  [max(Partners.Age) = 10] |
| min(*path)* | The minimum value of any element represented by the *path* . The *path* must end with an attribute.  [Min(Marks.Value) < 2] |
| count(*path)* | The number of elements represented by the *path*  [count(Partners) = 50] |
| avg(*path)* | The average value of all the elements represented by the *path.* The *path* must end with an attribute.  [avg(Partners.Age) > 12] |
| sum(*path)* | The sum of all the elements represented by the *path.* The *path* must end with an attribute.  [sum(Marks.Note) = 120] |
| exists(*path)* | *True* if the *path* exists  [exists(Partners)] |
| isNull(*path)* | *True* if the *path* doesn’t exist. The *path* must end with an attribute.  [isNull(Child)] |

Remark: As everything on OPath those commands are case sensitive.

In commands, *path* defines any OPath expression made by *identifiers* and *queries*. It is then allowed to include sub-queries into queries.

For example, the *path* could be a relationship from a *type* (the *Partners* of a *Person* type). Have a look to the following *OPath* request:

Person[ count(Partners) = 1 ]

This OPath request loads all entities of type Personwhich own one Partner.

### Literals

The values are separated in several data types:

#### Strings

The string values are given between simple quote.

*A simple quote in a string value can be escaped by including "\\" before it.*

* 'p1'
* 'John'
* 'O\\'Bryan'

#### Integers

* 42
* 2005
* -16

#### Reals

* 12.56
* -3.03

The decimal separator is always the dot (.), whatever the culture.

#### Date/Times

* #2005-25-12# (*en-us* culture)
* #25/12/2005# (*fr* culture)
* #2005-06-12 18:32:00#

The date format in OPath queries depends on the local machine system. All the more, you can write culture independent queries by using the *sortable* date format string like this:

DateTime.Now.ToString("s");

#### Booleans

* *true*
* *false*

## Referring to the Identifier

The identifier of any object can be constrained by using the id() function. Some special cases can make the Id property to work also but this is not a supported behavior and thus should not be used.

## OPath Examples

| Example | Description |
| --- | --- |
| Person | All entities of type *Person* |
| Person[Name = 'p3'] | All entities of type *Person*, where *Name* attribute is *p3* |
| Person[Name = 'p1' or Name = 'p2'] | All entities of type Person where *Name* attribute is *p1* *or* *p2* |
| Person[].Partners  Person.Partners | All entities of type *Partners* from an Entity of type *Person* |
| Person[Name = 'p3'].Partners | All entities of type *Partners* from an Entity of type *Person* where *Name* attribute is *p3* |
| Person[count(Partners) = 2] | All entities of type *Person* which own 2 *Partners* relationships |
| Person[exists(Partners)] | All entities of type *Person* which own one or more *Partners* relationships |
| Person[not exists(Partners)] | All entities of type *Person* which don't have any *Partners* relationship |
| Person[exists(Partners[Name = 'p2'])] | All entities of type *Person* which own a *Partner* relationship where *Name* attribute is *p2* |
| Person[ exists(Partners[ exists(Partners[Name = 'p3'])])] | All entities of type *Person* which own a *Partner* relationship which owns itself a *Partner* relationship where *Name* attribute is *p3* |
| Person[Name='p1' or exists(Partners[Name = 'p3'])] | All entities of type *Person* where *Name* attribute is *p1* or all entities of type *Person* which own a *Partners* relationship where *Name* attribute is *p3* |

## Scalar Opath

Euss provides a LoadScalar() method which returns a single value computed from an expression. This expression can include one or more aggregation functions (sum, avg, min, max, count) link by additive operators (+, -, \*, /, %).

Note: Not all repositories support aggregation expressions with operators, that’s why it is recommended to use one arithmetic function (in order to be cross repository).

### Aggregate functions

|  |  |
| --- | --- |
| Example | Description |
| count(Person) | Number of *Person* |
| sum(Person.Age) | Sum of ages of all *Person* |
| avg(Person.Partners.Age) | Age average of all Partners of all *Person* |
| min(Person.Age) | Age of the younger *Person* |
| max(Person.Age) | Age of the older *Person* |
| count(Person[count(Partners) = 2]) | Number of *Person* which own 2 *Partners* relationships |
| avg(Person[exists(Partners)].Age) | Average age of *Person* which own one or more *Partners* relationships |

### Arithmetic expressions

|  |  |
| --- | --- |
| Example | Description |
| count(Person) % 2 | Number of Person modulo two |
| sum(Person.Age) / count(Person) | Age average of all *Person* |

# Tools and Utilities

## Command Line Tools

### Location

All the command line utilities can be found under the installation folder used for Euss. By default they are installed under C:\Program Files\Evaluant\Eussfor .Net X.X, where *X.X* stands for the Euss version you installed.

### Conventions

#### Filenames

When setting filenames as command line arguments you can specify long filenames by using quotes (") thus enabling you to insert spaces in arguments.

#### Multi-valued arguments

If an argument accepts multiple values, those ones must be separated by a semi-colon (;). They are marked using a star (\*).

## mapping.exe

Generates a mapping file from several metadata sources

#### Syntax

mapping.exe

[/model:model\_filenames\*]

[/assembly:assembly\_filenames\*]

[/metadata:metadata\_filenames\*]

[/constraints]

[/native]

[/truncate:length]

[/strategy:strategy]

/out:mapping\_filename

#### Arguments

/model:model\_filenames

Optional. The list of model filenames the metadata should be loaded from.

/assembly:assembly\_filenames

Optional. The list of assembly filenames the metadata should be loaded from.

/metadata:metadata\_filenames

Optional. The list of metadata filenames the metadata should be loaded from.

/native

Optional. If specified, uses native identifiers instead of guids.

/truncate:length

Optional. The maximum length of name. For instance Oracle database are limited to 30.

/string:size

Optional. The default size applied to string fields.

/strategy:strategy

Optional. The name of the inheritance strategy to use. Allowed values: TablePerHierarchy, TablePerConcreteClass, TablePerSubClass.

/constraints

Optional. If set, the constraints between foreign keys will be used in the mapping file.

/out:mapping\_filename

Mandatory. The name of the mapping file to generate.

#### Notes

This command line utility generates a sample mapping file which can be changed afterwards to comply with your specific needs. For instance all the textual fields will be configured to use fifty (50) characters by default, and each class hierarchy mapped to a single database table.

#### Example

This example generates a mapping file from an XML model file.

mapping.exe /model:"c:\models\shop.xml" /out:"c:\models\shop.eum.xml"

This example generates a mapping file from two model files and an assembly.

mapping.exe /model:"c:\models\shop1.xml;c:\models\shop2.xml"

/assembly:"c:\domain.dll" /out:"c:\models\shop.eum.xml"

## domain.exe

Generates the persistence source code layer corresponding to a domain model.

#### Syntax

domain.exe

/model:model\_filename

[/lang:programming\_language]

#### Arguments

/model:model\_filename

Mandatory. The filename of the model. It can be an Euss XML model file or an XMI

/lang:programming\_language

Optional. The programming language used to generate the persistence layer (C# by default).

Allowed values: csharp or vbnet.

#### Example

This example generates the source code from a model file:

domain.exe /model:crm.domain.xml

#### Notes

The source code is generated in the same folder as the model.

The XMI files can be generated using these tools:

* Objecteering
* PowerDesigner
* PowerAMC
* Rose
* Visio
* Together
* VisualParadigm
* Poseidon
* StarUML
* Enterprise Architect

## initialize.exe

Initializes a repository.

#### Syntax

initialize.exe

/config:config\_filename

[/engine: engine\_name]

#### Arguments

/config:config\_filename

Mandatory. The Euss Configuration File containing the persistence engine’s repository to initialize.

/engine:engine\_name

Optional. The engine configuration name to use. If not provided the default engine is used.

#### Notes

This command line tool is equivalent to calling the method InitializeRepository() on the specified configuration.

## migrate.exe

Copies all the data from a repository to another one. This is generally used when migrating from a generic repository (XmlProvider, SqlProvider) to a specific one (SqlMapperProvider).

#### Syntax

migrate.exe

/config:config\_filename

/source:source\_config\_name

/target:target\_config\_name

[/init]

[/bulk]

[/silent]

#### Arguments

/config:config\_filename

Mandatory. The Euss Configuration filename containing repositories information.

/source:source\_config\_name

Mandatory. The name of the configuration section to use as the source repository.

/target:target\_config\_name

Mandatory. The name of the configuration section to use as the target repository.

/init

Optional. Whether the target repository should be initialized before the migration.

/bulk

Optional. Whether a single transaction should be used or not. This can improve the performances for some providers.

/silent

Optional. Whether the trace should not be displayed.

#### Notes

Both models must be compatible.

#### Example

This example transfers all the data from an XML file repository to a relational database based. The target repository is initialized before the transfer to occur.

migrate.exe /config:engines.config /source:"data (xml)" /target:"data (sql)" /init

## xmi2domain.exe

Converts an XMI file to the Euss xml domain model format.

#### Syntax

xmi2domain.exe

/xmi:xmi\_filename

[/model:model\_filename]

#### Arguments

/xmi:xmi\_filename

Mandatory. The XMI file describing the domain model in XMI format.

/model:model\_filename

Optional. The file containing the domain model in XML format. If not defined the model file is generated close to the XMI file ('model.xml')

## scriptddl.exe

Generates the DDL script from a mapping file.

#### Syntax

scriptddl.exe

/mapping:mapping\_filename

/dialect:dialect\_type

[/script:script\_filename]

#### Arguments

/mapping:mapping\_filename

Mandatory. The file containing the mapping.

/dialect:dialect\_type

Mandatory. The name of the dialect to use.

/script:script\_filename

Optional. The file containing the generated script. If not defined the script is generated on the basis of the mapping filename

#### Example

This example generates an Oracle DDL script in the file customers.sql.

scriptddl.exe /mapping:"customers.eum.xml"   
 /dialect:"Evaluant.Uss.SqlMapper.OracleDialect"   
 /script:"customers.sql"

# License

Copyright (c) 2006 Evaluant RC S.A.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1. .NET types in italic mean that this is a suggested .NET type if you want to set the database type by hand [↑](#footnote-ref-1)
2. Set size to zero (0) for unlimited column types (e.g. CLOB, TEXT) [↑](#footnote-ref-2)