



EntityORM 1.2

Documentation

Executed by
Hugo Malha Ferreira

Release Date
16-04-09

1. Table of Contents

1. TABLE OF CONTENTS	2
2. CONFIGURATION	4
2.1. CONNECTING TO THE DATABASE	4
2.2. ACCEPT NULLS	4
2.3. NULL AS DEFAULT NUMBER	4
2.4. AUTOMATIC MARK CHANGED	5
2.5. AUTOMATIC LAZY LOADING	5
3. WALKTHROUGH.....	6
3.1. ENTITIES	6
3.2. GENERIC ENTITY CLASS.....	8
3.3. GENERIC BASEENTITY CLASS	8
3.4. ENTITYMANAGER CLASS	8
3.5. TYPES OF SUPPORTED PROPERTIES	9
3.6. CONDITIONS	9
3.7. JOIN CONDITIONS	10
3.8. PERSISTRESULT	10
3.9. GENERIC ENTITYLIST	10
3.10. ENTITYLISTMANAGER CLASS.....	11
4. RULES VALIDATION.....	12
4.1. WHAT IS RULES VALIDATION ?	12
4.2. CREATE A RULE VALIDATION CLASS	12
5. VIEW.....	13
5.1. ENTITYVIEW.....	13
5.2. ENTITYVIEWMANAGER CLASS AND IENTITYVIEW INTERFACE	14
5.3. ENTITYVIEWLIST	14
5.4. ENTITYVIEWLISTMANAGER CLASS	14
6. ATTRIBUTES	15
7. VIEW ATTRIBUTES	16
8. EXCEPTIONS	17
9. GOOD PRACTICES	18
10. USE CASES	19

10.1.	MORE COMPLETE EXAMPLE	19
10.2.	MOST COMMON EXAMPLE.....	21
10.3.	EXAMPLES WITH VIEWS.....	23

2. Configuration

2.1. Connecting to the database

You only have to specify your database connection once in your application.

```
C#: Config.DataProvider = new EntityORM.SqlServer(@"Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa");
```

```
VB.NET: Config.DataProvider = New EntityORM.SqlServer("Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa")
```

2.2. Accept Nulls

If you want that your tables never store null values than you can specify once in your application (the default value is true).

```
C#: Config.AcceptNulls = false;
```

```
VB.NET: Config.AcceptNulls = False
```

2.3. Null As Default Number

If you want, the number properties (integers and decimals) can be persisted an null if the value is zero (the default value if false).

```
C#: Config.NullAsDefaultNumber = true;
```

```
VB.NET: Config.NullAsDefaultNumber = True
```

2.4. Automatic mark changed

The framework can automatic mark the changed properties (for changed entities), persisting only the changed properties in the database (the default value is true).

This mechanism has three requirements:

1. The entity can't be sealed (NotInheritable);
2. The entity class could not be created as nested class (class inside another class);
3. The properties must be virtual (Overridable).

C#: *Config*.AutomaticMarkChanged = *false*;

VB.NET: *Config*.AutomaticMarkChanged = *False*

2.5. Automatic lazy loading

The framework can also be in automatic lazy loading mode (the default value is true). Automatic lazy loading is a mechanism for load the data only and once when is required.

C#: *Config*.AutomaticLazyLoading = *false*;

VB.NET: *Config*.AutomaticLazyLoading = *False*

3. Walkthrough

3.1. Entities

The entities represents the records in the database in the form of objects, so the entities structures (the class that represents an entity project) can be decorated with attributes (provided by the framework) for define how the data is persisted in the database (see the samples above).

The classes should extend from generic Entity (provided by the framework) of entity type defined or at least extends from generic BaseEntity (explained later). These typed entities avoid castings later in the code.

The properties can be decorated with attributes to tell the framework how the data is persisted in the database or how the entities is linked each other.

The tables with numeric fields with zero decimal length instead of integer from old databases or database engines that not support integer or else for backward compatibility reason can be unboxed as integer to corresponding integer properties in the entities.

In the above sample you can see a simple Client class with several properties, one is the primary key, other is a relation one-to-one to the Country entity and the other is a list of other entities related with this Client entity.

In this sample the Account class has several properties and one is the key for the relation with the Client entity and the other is a relation one-to-one to the Article entity.

C# sample with 2 entity classes:

```
public class Client : Entity<Client>
{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    public virtual string Name { get; set; }
    public virtual int CountryID { get; set; }
    public Country Country { get; set; }
    public virtual string EMail { get; set; }
    public virtual byte[] Picture { get; set; }
    public EntityList<Account> Accounts { get; set; }
}

public class Country : Entity<Country>
{
    [PrimaryKey(), AutoNumber(), Relation(typeof(Client), "CountryID")]
    public int ID { get; set; }
    public virtual string Code { get; set; }
    public virtual string Name { get; set; }
}
```

```

public class Account : Entity<Account>
{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    [Relation(typeof(Client), "ID")]
    public virtual int ClientID { get; set; }
    public virtual int ArticleID { get; set; }
    public Article Article { get; set; }
    public virtual decimal Debit { get; set; }
    public virtual decimal Credit { get; set; }
}

public class Article : Entity<Article>
{
    [PrimaryKey(), AutoNumber(), Relation(typeof(Account), "ArticleID")]
    public int ID { get; set; }
    public virtual string Reference { get; set; }
    public virtual string Description { get; set; }
}

```

VB.NET sample with 2 entity classes (the properties was resumed to shrink the sample):

```

Public Class Client
    Inherits Entity(Of Client)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    Public Overridable Property Name() As String
    Public Overridable Property CountryID() As Integer
    Public Overridable Property Country() As Country
    Public Property EMail() As String
    Public Property Picture() As Byte()
    Public Property Accounts() As EntityList(Of Account)
End Class

Public Class Country
    Inherits Entity(Of Country)

    <PrimaryKey(), AutoNumber(), Relation(GetType(Client), "CountryID")> _
    Public Property ID() As Integer
    Public Overridable Property Code() As String
    Public Overridable Property Name() As String
End Class

Public Class Account
    Inherits Entity(Of Account)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    <Relation(GetType(Client), "ID")> _
    Public Overridable Property ClientID() As Integer
    Public Overridable Property ArticleID() As Integer
    Public Property Article() As Article
    Public Property Debit() As Decimal
    Public Property Credit() As Decimal
End Class

Public Class Article
    Inherits Entity(Of Article)

    <PrimaryKey(), AutoNumber(), Relation(GetType(Account), "ArticleID")> _
    Public Property ID() As Integer
    Public Overridable Property Reference() As String
    Public Overridable Property Description() As String
End Class

```

3.2. Generic Entity class

The generic Entity class should be extended by the entity classes and provides all the features needed like events before and after loading data, events before and after saving data, load method with condition, save method, delete method and reject changes method and also provides all the features from the BaseEntity.

3.3. Generic BaseEntity class

The generic BaseEntity class must be extended by the entity classes if not extended by the generic Entity class.

The BaseEntity class provides the state of the entity (Detached; Unchanged; Added; Modified; Deleted), the changed properties list for the changed properties of modified entities, pending lazy loading data list that provides a list of the properties pending for loading data, new method to return a new entity ready, on lazy loading method to load the pending data and on property changed method to mark a property as changed.

3.4. EntityManager class

As already told before you should extend from Entity class with all the work done but this method is not mandatory, in fact you can build a cover from the framework using BaseEntity and EntityManager class. In fact the Entity class extends from BaseEntity and uses EntityManager to complete the work and add delete and reject changes methods (this two are the only that is particularly from the Entity class).

The EntityManager class provides the following:

1. Events: BeforeReading; AfterRead; BeforeSaving; AfterSave;
2. Default method that return a new entity with the default values for all the property types;
3. Load that loads the data into the entity with a condition;
4. Save that persists the data in the database and returns a PersistResult object.

3.5. Types of supported properties

The following types of properties are supported by the framework:

1. String;
2. Integer;
3. Long;
4. Decimal;
5. DateTime (for date fields);
6. Boolean;
7. Byte Array (for binary fields).

3.6. Conditions

The conditions are useful to load filter data from the database into the new entity object and the following operations are available:

- And (&)
- Or (|)
- Equal (==)
- Not Equal (!=)
- Larger (>)
- LargerOrEqual (>=)
- Smaller (<)
- SmallerOrEqual (<=)
- StartWith
- Contains
- EndsWith
- Between
- NotBetween

3.7. Join Conditions

The join conditions are useful to join data from the database into one single flat entity object and the following operations are available:

- And (full join)
- Bigger (left join)
- Lower (right join)
- And (add more joins)

The join conditions can also receive a second parameter with the join cast type (text or number) for casting the member.

3.8. PersistResult

The persist result have a code from the persisting result and could have a message string with other information. The persist code could be one of the following:

- OperationComplete
- OperationCanceled
- NothingToDo
- CreateError
- UpdateError
- DeleteError
- RuleValidationFailed

3.9. Generic EntityList

You can use simple lists to store several entities from the same type but the framework provides a EntityList (in fact extends from generic List) and provides events before and after loading data and before and after saving data, a list of the deleted entities that automatic are moved to this sub-list when deleted, add method to automatic add a new entity typed object to

the list, list method to load all the data from the database, load with condition, load with condition and order, save method to persist all the entities and reject changes method to reject the changes from all entity objects.

3.10.EntityListManager class

You should use EntityList for a list of entities of the same type but if you want you can create your entity list using EntityListManager class to achieve this end, in fact EntityList uses EntityListManager adding also reject changes method and internal manage the deleted entities.

The EntityListManager class provides the following:

1. Events: BeforeReading; AfterRead; BeforeSaving; AfterSave;
2. New method to create a new type list;
3. List with condition to load data with filter;
4. List with condition and order to load data with filter and sort the data;
5. List with query (the Query is a class that provides the condition, order and limit of data);
6. Save to persist all the data.

4. Rules Validation

4.1. What is rules validation ?

Rules validation is specific property entities restrictions to be validated before the entity is persisted, so when the first rule validation failed in the entity/entity list save then a rule validation failed persist result code will be returned with the property name in the message data.

4.2. Create a rule validation class

The rules validation classes must implement the *IRuleValidation* interface provided by the framework. This interface required that the class that implements it have an *IsValid* method with an object parameter.

C# sample:

```
public class Client : Entity<Client>
{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    public virtual string Name { get; set; }
    [RuleValidation(typeof(MailValidation))]
    public virtual string EMail { get; set; }
}

public class MailValidation : IRuleValidation
{
    public bool IsValid(object value)
    {
        return value != null && new Regex(@"^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,4}$").IsMatch(value.ToString());
    }
}
```

VB.NET sample (the properties was resumed to shrink the sample):

```
Public Class Client
    Inherits Entity(Of Client)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    Public Overridable Property Name() As String
    <RuleValidation(GetType(MailValidation))> _
    Public Property EMail() As String
End Class

Public Class MailValidation
    Implements IRuleValidation

    Public Function IsValid(ByVal value As Object) As Booleana
        Return value IsNot Nothing AndAlso New Regex(@"^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,4}$").IsMatch(value.ToString())
    End Function
End Class
```

5. View

5.1. EntityView

The entities are great for mapping the database tables into entities logical related each other without the needed for defining joins and are enough for load, change, create and delete but sometimes we could need map several tables into a single flat entity. This mechanism doesn't allow persisting the entities changes but is must faster loading a pile of data and is useful for mapping the entity properties in reporting tools.

The EntityView provides events for before loading and after loaded entities.

The EntityView supports conditions exactly like the Entity but also could be load the entities by join conditions or both.

The following two examples show two entity views structures.

C# sample with 2 entity classes:

```
[MapTo("Client")]
public class SellDetailReport : EntityView<SellDetailReport>
{
    public string Name { get; set; }
    [FromTable("Country"), MapTo("Name")]
    public string CountryName { get; set; }
    public string EMail { get; set; }
    [FromTable("Article"), MapTo("Reference")]
    public string ArticleReference { get; set; }
    [FromTable("Article")]
    public string Description { get; set; }
    [FromTable("Sell"), MapTo("Qty")]
    public int Quantity { get; set; }
}

[MapTo("Client")]
public class TotalSellClientReport : EntityView<TotalSellClientReport>
{
    public string Name { get; set; }
    [FromTable("Sell"), Aggregate(Aggregate.Count)]
    public int TotalSells { get; set; }
    [FromTable("Sell"), MapTo("Qty"), Aggregate(Aggregate.Sum)]
    public int TotalQuantity { get; set; }
}
```

VB.NET sample with 2 entity view classes (the properties was resumed to shrink the sample):

```
<MapTo("Client")> _
Public Class SellDetailReport
    Inherits EntityView(Of SellDetailReport)

    <FromTable("Country"), MapTo("Name")> _
    Public Property CountryName() As String
```

```

Public Property EMail() As String
<FromTable("Article"), MapTo("Reference")> _
Public Property ArticleReference() As String
<FromTable("Article")> _
Public Property Description() As String
<FromTable("Sell"), MapTo("Qty")> _
Public Property Quantity() As Integer
End Class

<MapTo("Client")> _
Public Class TotalSellClientReport
Inherits EntityView(Of TotalSellClientReport)

Public Property Name() As String
<FromTable("Sell"), Aggregate (Aggregate.Count)> _
Public Property TotalSells() As Integer
<FromTable("Sell"), MapTo("Qty"), Aggregate (Aggregate.Sum) > _
Public Property TotalQuantity() As Integer
End Class

```

5.2. EntityViewManager class and IEntityView interface

You can use EntityView as also Entity but like Entity you can made you own entity view class using EntityViewManager that provides all the need functions as long your class implements the IEntityView interface provided by the framework.

5.3. EntityViewList

Like the EntityList for standard entities, there is an EntityViewList for entity views with events before and after loading entities, conditions, join conditions, orders or any mixed of this features.

5.4. EntityViewListManager class

You can use EntityViewList as also EntityList but like EntityList you can made you own entity view list class using EntityViewListManager that provides all the need functions.

6. Attributes

1. `MapTo` (also available for the classes) – defines the real name in the database (for default the name is the same of the class/property).
2. `LazyLoading` – when defined, the data is now immediately loaded and will be loaded when it is needed.
3. `PrimaryKey` – the corresponding primary key in the table.
4. `AutoNumber` – sequential number managed by the database (when the property also has the primary key attribute then number is automatically returned from the database for the created entities).
5. `Guid` – unique identifier managed by the framework.
6. `Size` – the reserved size for the field in the table (exists but is ignored in this version).
7. `Decimal` – the reserved decimal for the field in the table (exists but is ignored in this version).
8. `Relation` – tells the framework that the property should be linked with another entity type.
9. `NotExists` – tells the framework that the property should be ignored for loading and persisting data.
10. `NullAsZero` – overrides the *NullAsDefaultNumber* configuration and tells the framework to persist the property zero value as null.
11. `DefaultValue` – defines the property default value for new created entity.
12. `RuleValidation` – defines the type of specific rule class to be validated in the entity property on the save method (see the rules validation chapter).

7. View Attributes

1. `MapTo` (also available for the classes) – defines the real name in the database (for default the name is the same of the class/property).
2. `FromTable` – defines the table name that belongs to the property.
3. `Distinct` – defines in the entity class to distinct the entities.
4. `NotExists` – tells the framework that the property should be ignored for loading data.
5. `Group` – the property will be grouped (is automatically assumed when other property have an aggregate attribute).
6. `Aggregate` – the property is *virtual* and the framework will store the returned value from the aggregated function (count, sum, largest, average or smallest).

8. Exceptions

The following exceptions may occur to help the developer to solve problems:

- Property type "XX" not supported;
- Property XX in the entity type YY have a relation one-to-one to entity type ZZ but the Relation attribute is not found in the ZZ entity type !;
- Property XX in the entity type YY have a relation one-to-many to entity type ZZ but the Relation attribute is not found in the ZZ entity type !;
- Join condition not supported !

9. Good Practices

There are some good practices to have in mind:

- The class and properties definition should be camel case
- The variables should be private and start with lower case or underscore
- The tables names and fields in the database should be in the same case as the entities and properties because there could be same database engine that the objects are case sensitive (ex. *PostgreSQL*)
- The *MapTo* attribute, conditions and join conditions should have same case as the case of the objects in the database because there could be same database engine that the objects are case sensitive (ex. *PostgreSQL*)

10. Use Cases

10.1. More complete example

The following use case demonstrates the use of the EntityORM framework using the two entities from the chapter entities and the mandatory configuration.

In this sample you can see the use of AfterSave event handler.

As you can see that there is now need for casting types all the time that makes the code more “cleaner” and efficient.

C#:

```
using System;
using EntityORM.Core;

namespace Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            //mandatory configuration
            Config.DataProvider = new EntityORM.SqlServer(@"Data Source=.sqlexpress;Initial Catalog=demo;User Id=sa");

            /*
             * single entity
             */

            //event handler after saves a client
            Client.AfterSave += new EntityEventHandler<Client>(Client_AfterSave);

            //load a single client
            Client myClient = Client.Load(Condition.Member("Name") == "Hugo");
            //changes the email
            myClient.EMail = "hferreira.80@gmail.com";
            //persiste the entity (only the email field will be sent the database)
            myClient.Save();

            /*
             * list of entities
             */

            //load all my clients
            EntityList<Client> myClients = EntityList<Client>.List();
            //delete the last client
            myClients[myClients.Count - 1].Delete();
            //persiste the data (only the last client will the dropped)
            myClients.Save();
        }

        //event after saves a client
        static void Client_AfterSave(Client sender, System.EventArgs e)
        {
            Console.WriteLine(string.Format("Client {0} sucessfull saved.", sender.Name));
        }
    }
}
```

```
}
```

VB.NET:

Imports System

Imports EntityORM.Core

Namespace Demo

Public Class Program

Private Shared Sub Main(ByVal args As String())

'mandatory configuration

Config.DataProvider = New EntityORM.SqlServer("Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa)

,

' single entity

,

'event handler after saves a client

AddHandler Client.AfterSave, AddressOf Client_AfterSave

'load a single client

Dim myClient As Client = Client.Load(Condition.Member("Name") = "Hugo")

'changes the age

myClient. EMail = "hferreira.80@gmail.com"

'persiste the entity (only the email field will be sent the database)

myClient.Save()

,

' list of entities

,

'load all my clients

Dim myClients As EntityList(Of Client) = EntityList(Of Client).List()

'delete the last client

myClients(myClients.Count - 1).Delete()

'persiste the data (only the last client will the dropped)

myClients.Save()

End Sub

'event after saves a client

Private Shared Sub Client_AfterSave(ByVal sender As Client, ByVal e As System.EventArgs)

Console.WriteLine(String.Format("Client {0} sucessfull saved.", sender.Name))

End Sub

End Class

End Namespace

10.2. Most common example

The following use case demonstrates the use of the EntityORM framework using a most common example where the client have extended properties in a relation *one-to-one* in other entity and one of the properties is marked as *LazyLoading*, because could be an heavy data than is need when the user interacted like a page call.

The sells made to the user are a list of sells in a relation *one-to-many*.

The other entities could be used like in *comboboxs*.

The property notes have a default value.

C#:

```
using System;
using EntityORM.Core;

namespace Demo
{
    public class Program
    {
        static void Main(string[] args)
        {
            //mandatory configuration
            Config.DataProvider = new EntityORM.SqlServer(@"Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa");

            //load all my clients without condition filter
            EntityList<Client> myClients = EntityList<Client>.List();
        }
    }

    public class Client : Entity<Client>
    {
        [PrimaryKey(), AutoNumber()]
        public int ID { get; set; }
        public virtual string Name { get; set; }
        public virtual int CountryID { get; set; }
        public virtual string EMail { get; set; }
        public virtual byte[] Picture { get; set; }
        public ClientExtended ClientExtended { get; set; }
        public EntityList<Sell> Sells { get; set; }
    }

    [MapTo("ClientOther")]
    public class ClientExtended : Entity<ClientExtended>
    {
        [PrimaryKey(), AutoNumber()]
        public int ID { get; set; }
        [Relation(typeof(Client), "ID")]
        public int ClientID { get; set; }
        [DefaultValue("There is no notes")]
        public virtual string Notes { get; set; }
        [LazyLoading()]
        public virtual byte[] HousePlan { get; set; }
    }

    public class Sell : Entity<Sell>
```

```

{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    [Relation(typeof(Client), "ID")]
    public virtual int ClientID { get; set; }
    public virtual int ArticleID { get; set; }
    [MapTo("Qty")]
    public virtual int Quantity { get; set; }
}

public class Country : Entity<Country>
{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    public virtual string Code { get; set; }
    public virtual string Name { get; set; }
}

public class Article : Entity<Article>
{
    [PrimaryKey(), AutoNumber()]
    public int ID { get; set; }
    public virtual string Reference { get; set; }
    public virtual string Description { get; set; }
    public decimal Price { get; set; }
}
}

```

VB.NET (the properties was resumed to shrink the sample):

Imports System

Imports EntityORM.Core

Namespace Demo

Public Class Program

Private Shared Sub Main(ByVal args As String())

'mandatory configuration

Config.DataProvider = New EntityORM.SqlServer("Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa)

'load all my clients without condition filter

Dim myClients As EntityList(Of Client) = EntityList(Of Client).List()

End Sub

End Class

Public Class Client

Inherits Entity(Of Client)

<PrimaryKey(), AutoNumber()> _

Public Property ID() As Integer

Public Overridable Property Name() As String

Public Overridable Property CountryID() As Integer

Public Property EMail() As String

Public Property Picture() As Byte()

Public Property ClientExtended() As EntityList(Of ClientExtended)

Public Property Sells() As EntityList(Of Sell)

End Class

<MapTo("ClientOther")> _

Public Class ClientExtended

Inherits Entity(Of ClientExtended)

<PrimaryKey(), AutoNumber()> _

Public Property ID() As Integer

<Relation(GetType(Client), "ID")> _

Public Overridable Property ClientID() As Integer

<DefaultValue("There is no notes")> _

Public Overridable Property Notes() As String

```

    <LazyLoading()> _
    Public Property HousePlan() As Byte()
End Class

Public Class Sell
    Inherits Entity(Of Sell)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    <Relation(GetType(Client), "ID")> _
    Public Overridable Property ClientID() As Integer
    Public Overridable Property ArticleID() As Integer
    <MapTo("Qt")> _
    Public Property Quantity() As Decimal
    Public Property Debit() As Decimal
    Public Property Credit() As Decimal
End Class

Public Class Country
    Inherits Entity(Of Country)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    Public Overridable Property Code() As String
    Public Overridable Property Name() As String
End Class

Public Class Article
    Inherits Entity(Of Article)

    <PrimaryKey(), AutoNumber()> _
    Public Property ID() As Integer
    Public Overridable Property Reference() As String
    Public Overridable Property Description() As String
    Public Overridable Property Price() As Decimal
End Class

End Namespace

```

10.3.Examples with Views

The following use case demonstrates the use of the EntityORM framework using the two entities from the chapter view and the mandatory configuration.

In this sample you can see the use of JoinCondition.

The join condition member must be the *tablename.fieldname* instead of (*entityname.propertyname*) as also de Order (the order also exists in standard EntityList and in this case we use only the property name of the entity).

C#:

```

using System;
using EntityORM.Core;

namespace Demo
{
    public class Program

```

```

{
    static void Main(string[] args)
    {
        //mandatory configuration
        Config.DataProvider = new EntityORM.SqlServer(@"Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa");

        //sells details
        EntityViewList<SellDetailReport> sellDetailReport = EntityViewList<SellDetailReport>.List(
            JoinCondition.Member("Client.CountryID") == JoinCondition.Member("Country.ID") &
            JoinCondition.Member("Client.ID") == JoinCondition.Member("Sell.ClientID") &
            JoinCondition.Member("Sell.ArticleID") == JoinCondition.Member("Article.ID"));

        //total sells per client and sum the quantities
        EntityViewList<TotalSellClientReport> totalSellClientReport = EntityViewList<TotalSellClientReport>.List(
            JoinCondition.Member("Client.ID") == JoinCondition.Member("Sell.ClientID"), Order.Member("Client.Name"));
    }
}

```

VB.NET:

```

Imports System
Imports EntityORM.Core

Namespace Demo
    Public Class Program
        Private Shared Sub Main(ByVal args As String())
            'mandatory configuration
            Config.DataProvider = New EntityORM.SqlServer("Data Source=.\sqlexpress;Initial Catalog=demo;User Id=sa")

            'sells details
            Dim sellDetailReport As EntityViewList(Of SellDetailReport) = EntityViewList(Of SellDetailReport).List( _
                JoinCondition.Member("Client.CountryID") = JoinCondition.Member("Country.ID") And _
                JoinCondition.Member("Client.ID ") = JoinCondition.Member("Sell.ClientID") And _
                JoinCondition.Member("Sell.ArticleID ") = JoinCondition.Member("Article.ID"))

            ' total sells per client and sum the quantities
            Dim totalSellClientReport As EntityViewList(Of TotalSellClientReport) = EntityViewList(Of TotalSellClientReport).List( _
                JoinCondition.Member("Client.ID") = JoinCondition.Member("Sell.ClientID"), Order.Member("Client.Name"))
        End Sub
    End Class
End Namespace

```