# Signum Framework Tutorials - Part 3: Southwind Load

## About Signum Framework

Signum Framework is an application framework for making data-centric windows and web applications. It promotes a code-first workflow and is focused in composability, to share code between projects.

## About this series

In this series of tutorials we will work on a stable application: Southwind.

Southwind is the Signum version of Northwind, the well-known example database provided with Microsoft SQL Server.

In this series of tutorials we will create the whole application, including the entities, business logic, windows (WPF) and web (React) user interface, data loading and any other aspect worth to explain.

If you want to know more about the principles of Signum framework look at the previous tutorial:

* [Signum Framework Principles](http://www.codeproject.com/KB/linq/SignumFramework.aspx)
* Signum Framework Tutorials Part 1 – Southwind Entities
* Signum Framework Tutorials Part 2 – Southwind Logic

In this tutorial we will focus on moving the data from Northwind Database to the new Southwind.

### Introduction

One of the main reason applications get stuck in the past is because people is scared of changing the database, causing big amounts of money spent in maintaining old monsters that are not correctly normalized, constrained or has inconsistent naming.

Also, the fact that the application has been modified by too many people and the lack of validation rules centralized on the entities make it impossible to rely in any kind of invariant that the data should hold, making any modification way harder.

Signum Framework gives you the tools to make the new application right, but also helps you moving the legacy data in the new schema by using *Signum.Utilities*, LINQ, and CSV files.

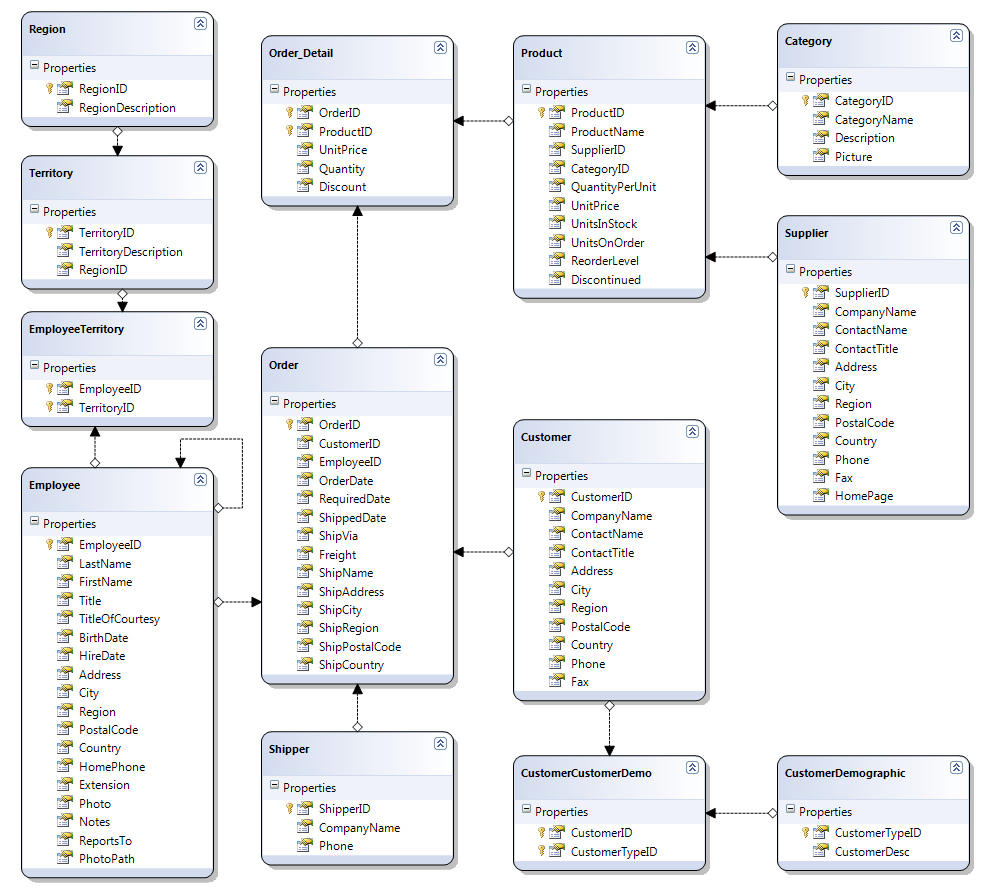
In order to read the legacy database (Northwind) we will use LINQ to SQL instead of LINQ to Signum. Signum Framework is a great tool to create data-centric applications on Greenfield projects, but is not the right tool if you just want to query legacy databases (it’s possible through, using IView and TableName attributes, but you will need to write the table classes by hand).

Northwind database is simplistic but is a good schema, reasonably normalized, the tables are property related and the data is quite homogeneous. This will make our work easier in this tutorial.

In a real world scenario, I will be glad if your legacy database looks like this. Most frequently, you will have to make more radical changes in your schema and more transformations of data in the loading application.

### Loading Data

Let’s go to Southwind.Load, Add new item, LINQ to SQL classes, connect to Northwind using Server Explorer, and drag all the tables in the designer surface. After arranging, the diagram should looks like this:



Let’s start loading some data!

### Loading Regions and Territories

In EmployeeLoader.cs , in order to load the Region table let´s add *LoadRegions* method.

Then, inside of the method create a NorthwindDataContext object and query the Regions property, creating a RegionEntity for each result like this:

public static void LoadRegions()

{

using (NorthwindDataContext db = new NorthwindDataContext())

{

db.Regions.Select(r => new RegionEntity

{

Description = r.RegionDescription.Trim(),

}).SaveList();

}

}

Even if this code works ok, it doesn’t preserve the Id from the Northwind database.

In data-loading scenarios is usually quite interesting to preserve the ids, if possible, to avoid polluting our entities with temporal Old\_Id fields or keeping separated mapping files.

In order to do that, we need to temporally disable the identity of the table, and set the Id property manually.

The Administrator static class is the Database counterpart for dangerous situations, and shouldn’t be used in production, but it’s quite useful for this kind of dirty hacks in the Load application.

The code will look like this:

public static void LoadRegions()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                Administrator.SaveListDisableIdentity(db.Regions.Select(r =>

                    new RegionEntity

                    {

                        Description = r.RegionDescription.Trim()

                    }.SetId(r.RegionID)));

            }

        }

If we run the Load application, choose load and pick the first method (0 – Load Regions), the code will be run and the new records will be in the database.

Let’s make a similar method for territories:

public static void LoadTerritories()

{

using (NorthwindDataContext db = new NorthwindDataContext())

{

var regionDic = Database.RetrieveAll<RegionEntity>().ToDictionary(a => a.Id);

Administrator.SaveListDisableIdentity(db.Territories.Select(r =>

Administrator.SetId(int.Parse(r.TerritoryID), new TerritoryEntity

{

Description = r.TerritoryDescription,

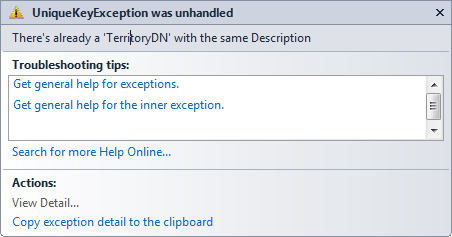
Region = regionDic[r.RegionID]

})));

}

}

Now we add the method to the console menu (ConsoleSwitch) and run it and… whoops:



Looks like NewYork is written twice in Northwind database, now it gets more interesting. In order to remove duplicates we need to group the territories by description like this:

public static void LoadTerritories()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                var regionDic = Database.RetrieveAll<RegionEntity>().ToDictionary(a => a.Id);

                var territories = (from t in db.Territories.ToList()

                                   group t by t.TerritoryDescription into g

                                   select new

                                   {

                                       Description = g.Key.Trim(),

                                       Id = g.Select(t => t.TerritoryID).OrderBy().First(),

                                       RegionID = g.Select(r => r.RegionID).Distinct().Single(),

                                   }).ToList();

                Administrator.SaveListDisableIdentity(territories.Select(t =>

                    new TerritoryEntity

                    {

                        Description = t.Description.Trim(),

                        Region = regionDic[t.RegionID]

                    }.SetId(int.Parse(t.Id))));

            }

        }

### Loading Employees

Employee is a bigger entity so it will get a little bit more complicated. We create *LoadEmployees* method with the same structure than the ones before and add it to the console menu.

The first problematic point is how to deal with the duplicated territories. What we need is a way to translate the duplicated territory id to the non-duplicated one. We can create a dictionary like this with this Linq to Object query:

var duplicateMapping = (from t in db.Territories.ToList()

                                   group int.Parse(t.TerritoryID) by t.TerritoryDescription

into g

                                   where g.Count() > 1

                                   let min = g.Min()

                                   from item in g.Except(new[] { min })

                                   select new

                                   {

                                          Min = min,

                                          Item = item

                                     }).ToDictionary(a => a.Item, a => a.Min);

What we do here is grouping the territory ids by territory description. For the groups with more than one element (duplicates) we pick the min Id and make a dictionary from everyone else, to the minimum. It’s not that complex!

If we try to run this code we get a summary of validation errors for each entity, basically there are many repetitions of the following errors:

The length of Region has to be greater than or equal to 3  
 The length of Country has to be greater than or equal to 3  
 Region is not set

In this case, our validation rules are too strict, let’s modify the StringLengthValidator over Country and Region property in AddressEmbedded to allow 2 character ones.

On Region property we also need to set AllowNulls to true and remove NotNullable from the field like this:

[SqlDbType(Size = 15)]

[StringLengthValidator(AllowNulls = true, Min = 2, Max = 15)]

public string Region { get; set; }

[NotNullable, SqlDbType(Size = 15)]

[StringLengthValidator(AllowNulls = false, Min = 2, Max = 15)]

public string Country { get; set; }

Since the last changes will modify the schema, we need to generate a new synchronization script before continuing:

Generating script...Already synchronized!

Whoops! We expected some changes but it says everything is ok. The reason is that EmbeddedEntities, in order to express a null reference, add an extra Boolean field HasValue, and force nullability on all the other fields.

In this case, however, AddressEmbedded is mandatory on EmployeeEntity, as well as CustomerEntity, OrderEntity or SupplierEntity, so we can add a NotNullValidator on every property of type AddressEmbedded and a NotNullable on the field like this:

[NotNullable]

[NotNullValidator]

public AddressEmbedded Address { get; set; }

Let’s try to synchronize again:

ALTER TABLE Company ALTER COLUMN Address\_Address NVARCHAR(60) NOT NULL;

ALTER TABLE Company ALTER COLUMN Address\_City NVARCHAR(15) NOT NULL;

ALTER TABLE Company ALTER COLUMN Address\_PostalCode NVARCHAR(10) NOT NULL;

ALTER TABLE Company ALTER COLUMN Address\_Country NVARCHAR(15) NOT NULL;

ALTER TABLE Company DROP COLUMN Address\_HasValue;

ALTER TABLE Employee ALTER COLUMN Address\_Address NVARCHAR(60) NOT NULL;

ALTER TABLE Employee ALTER COLUMN Address\_City NVARCHAR(15) NOT NULL;

ALTER TABLE Employee ALTER COLUMN Address\_PostalCode NVARCHAR(10) NOT NULL;

ALTER TABLE Employee ALTER COLUMN Address\_Country NVARCHAR(15) NOT NULL;

ALTER TABLE Employee DROP COLUMN Address\_HasValue;

ALTER TABLE Supplier ALTER COLUMN Address\_Address NVARCHAR(60) NOT NULL;

ALTER TABLE Supplier ALTER COLUMN Address\_City NVARCHAR(15) NOT NULL;

ALTER TABLE Supplier ALTER COLUMN Address\_PostalCode NVARCHAR(10) NOT NULL;

ALTER TABLE Supplier ALTER COLUMN Address\_Country NVARCHAR(15) NOT NULL;

ALTER TABLE Supplier DROP COLUMN Address\_HasValue;

ALTER TABLE Person ALTER COLUMN Address\_Address NVARCHAR(60) NOT NULL;

ALTER TABLE Person ALTER COLUMN Address\_City NVARCHAR(15) NOT NULL;

ALTER TABLE Person ALTER COLUMN Address\_PostalCode NVARCHAR(10) NOT NULL;

ALTER TABLE Person ALTER COLUMN Address\_Country NVARCHAR(15) NOT NULL;

ALTER TABLE Person DROP COLUMN Address\_HasValue;

ALTER TABLE Order ALTER COLUMN ShipAddress\_Address NVARCHAR(60) NOT NULL;

ALTER TABLE Order ALTER COLUMN ShipAddress\_City NVARCHAR(15) NOT NULL;

ALTER TABLE Order ALTER COLUMN ShipAddress\_PostalCode NVARCHAR(10) NOT NULL;

ALTER TABLE Order ALTER COLUMN ShipAddress\_Country NVARCHAR(15) NOT NULL;

ALTER TABLE Order DROP COLUMN ShipAddress\_HasValue;

Perfect, note how all the AddressEmbedded fields have become NOT NULL but Region. Let’s run the script to make the changes.

The last problem that we have to solve is recomposing the ReportsTo hierarchy. Since there are foreign keys to the table itself, it will be impossible to add references to managers that have not been loaded yet.

Instead, we will create a small loop that recomposes the hierarchy after all the employees have been loaded, like this:

var pairs = (from e in db.Employees

               where e.ReportsTo != null

               select new { e.EmployeeID, e.ReportsTo });

  foreach (var pair in pairs)

  {

     EmployeeEntity employee = Database.Retrieve<EmployeeEntity>(pair.EmployeeID);

      employee.ReportsTo = Lite.Create<EmployeeEntity>(pair.ReportsTo.Value);

      employee.Save();

  }

In this piece of code we query Northwind to get the pairs that represent the hierarchy, and then we use *Database.Retrieve* to retrieve the Employee, we create the *Lite<EmployeeEntity>* manually and then we use Save to update the Entity.

**Note:** Dealing with Ids manually in your business logic is more risky than using Lites, since Lites contain the Type information, helping to prevent bugs. In this case however we have no option since we are reading a LINQ to SQL database.

Ok, with this code we should be able to load Employees in our Southwind database.

Before moving forward to load other entities and become Program class a mess, let’s move the methods LoadRegions, LoadTerritories and LoadEmployees to a new EmployeeLoader static class. We will need to make the methods public and update the SwitchConsole menu on the Main method.

public static void LoadEmployees()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                var duplicateMapping = (from t in db.Territories.ToList()

                                        group int.Parse(t.TerritoryID) by t.TerritoryDescription

into g

                                        where g.Count() > 1

                                        let min = g.Min()

                                        from item in g.Except(new[] { min })

                                        select new

                                        {

                                            Min = min,

                                            Item = item

                                        }).ToDictionary(a => a.Item, a => a.Min);

                var territoriesDic =

Database.RetrieveAll<TerritoryEntity>().ToDictionary(a => a.Id);

                var exmployeeTerritories = (from e in db.Employees

                                            from t in e.EmployeeTerritories

                                            select new { e.EmployeeID, t.TerritoryID }).ToList()

                           .AgGroupToDictionary(a => a.EmployeeID, gr =>

                              gr.Select(a => int.Parse(a.TerritoryID))

                              .Select(id => duplicateMapping.TryGet(id, id))

                              .Select(id => territoriesDic[id])

                              .Distinct().ToMList());

                Administrator.SaveListDisableIdentity(

                    from e in db.Employees

                    select new EmployeeEntity

                    {

                        BirthDate = e.BirthDate,

                        FirstName = e.FirstName,

                        LastName = e.LastName,

                        TitleOfCourtesy = e.TitleOfCourtesy,

                        HomePhone = e.HomePhone,

                        Extension = e.Extension,

                        HireDate = e.HireDate,

                        Photo = new FileEntity {

FileName = e.PhotoPath.AfterLast('/'),

BinaryFile = RemoveOlePrefix(e.Photo.ToArray()) }.ToLiteFat(),

                        PhotoPath = e.PhotoPath,

                        Address = new AddressEmbedded

                        {

                            Address = e.Address,

                            City = e.City,

                            Country = e.Country,

                            Region = e.Region,

                            PostalCode = e.PostalCode,

                        },

                        Notes = e.Notes,

                        Territories = exmployeeTerritories[e.EmployeeID],

                    }.SetId(e.EmployeeID));

                var pairs = (from e in db.Employees

                             where e.ReportsTo != null

                             select new { e.EmployeeID, e.ReportsTo });

                foreach (var pair in pairs)

                {

                   EmployeeEntity employee = Database.Retrieve<EmployeeEntity>(pair.EmployeeID);

                   employee.ReportsTo = Lite.Create<EmployeeEntity>(pair.ReportsTo.Value);

                   employee.Save();

                }

            }

        }

### Loading Products

This time we will make things right in the first place, creating a ProductLoader static class. In there we will start creating *LoadSuppliers* method that will look like the previous ones:

public static void LoadSuppliers()

{

using (NorthwindDataContext db = new NorthwindDataContext())

{

Administrator.SaveListDisableIdentity(db.Suppliers.Select(s =>

Administrator.SetId(s.SupplierID, new SupplierDN

{

CompanyName = s.CompanyName,

ContactName = s.ContactName,

ContactTitle = s.ContactTitle,

Phone = s.Phone,

Fax = s.Fax,

HomePage = s.HomePage,

Address = new AddressDN

{

Address = s.Address,

City = s.City,

Region = s.Region,

PostalCode = s.PostalCode,

Country = s.Country

},

})));

}

If we add the method to the menu, and try to run it, we will get a set of validation errors, all of them repetitions like this:

Phone does not have a valid Telephone format  
 Fax does not have a valid Telephone format   
 Home Page is not set  
 Fax is not set

First of all, note how *SaveListDisableIdentity* is transactional -as any other method in Database or Administrator classes- so in the case of an exception no changes are made.

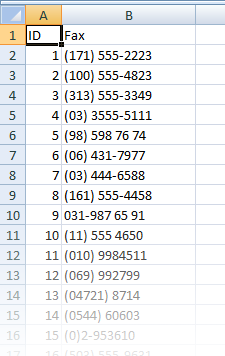
The first two errors are produced because Telephone validator does not allow dots (‘.’) on the telephone number, only numbers, space, hyphen and parenthesis. In this case let’s just replace the dots by space.

If we look at the Northwind data, we see that HomePage field contains very few and heterogeneous data that is not worth to load. Let’s change the field and property attributes to allow null values in our SupplierEntity entity and keep it blank.

### Reading CSV files

For the customers without faxes we will make it more complicated. Let’s pretend that it’s a business requirement for the new application to make the orders to the suppliers using fax, so we have to keep the field mandatory.

After an enormous chain of emails we finally get an Excel file with the missing fax numbers. Looks like this.



In excel, we save the file as a CSV file in our Southwind.Load directory (SupplierFaxes.csv).

Then in visual studio we include the file (Show all files icon in Solution Explorer 🡪 right click in the file 🡪 Include in Project) and on properties we set ‘Copy to Output Directory’ to ‘Copy if newer’.

Let’s take a look to the file. Depending of your **culture** the values will be separated by comma ‘,’ or semicolon ‘;’ and decimal numbers will use dot ‘.’ or comma ‘,’. In this case the file was generated in a Spanish computer.

It’s also necessary to look at the **encoding** of the file (File 🡪 Advanced Save Options). In this case Western European (Windows) – Codepage 1252.

Once we know the culture and the encoding, loading the file is easy. Let’s create a class with a public field for each column, in the same order.

public class SupplierFaxCSV

{

public int SupplierID;

public string Fax;

}

Then, in our *LoadSuppliers* method we use CSV.ReadCSV method to read the content of the file.

List<SupplierFaxCSV> faxes = CSV.ReadCVS<SupplierFaxCSV>("SupplierFaxes.csv",

Encoding.GetEncoding(1252), CultureInfo.GetCultureInfo("es"));

Note how we write all the parameters explicitly.

* We determine Encoding to Codepege 1252, otherwise it will be Unicode.
* CultureInfo to Spanish, otherwise it will be your current culture.
* And we explicitly tell that we want to skip reading the first line in the file (headers) even if it’s the default value.

Then let’s make a dictionary that we could use while loading out Customers:

var faxDic = faxes.ToDictionary(r => r.SupplierID, r => r.Fax);

And finally, let’s update our query for loading Suppliers:

public static void LoadSuppliers()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                List<SupplierFaxCSV> faxes =

Csv.ReadFile<SupplierFaxCSV>("SupplierFaxes.csv", culture: CultureInfo.GetCultureInfo("es"));

                var faxDic = faxes.ToDictionary(r => r.SupplierID, r => r.Fax);

                Administrator.SaveListDisableIdentity(db.Suppliers.Select(s =>

                    new SupplierEntity

                    {

                        CompanyName = s.CompanyName,

                        ContactName = s.ContactName,

                        ContactTitle = s.ContactTitle,

                        Phone = s.Phone.Replace(".", " "),

                        Fax = faxDic[s.SupplierID].Replace(".", " "),

                        Address = new AddressEmbedded

                        {

                            Address = s.Address,

                            City = s.City,

                            Region = s.Region,

                            PostalCode = s.PostalCode,

                            Country = s.Country

                        },

                    }.SetId(s.SupplierID)));

            }

        }

Let’s compile and generate a new sync script to update the field HomePage nullability.

ALTER TABLE Supplier ALTER COLUMN HomePage NVARCHAR(MAX) NULL;

And then the method should load the Suppliers as expected.

### Load Categories and Products

Loading categories shouldn’t have any difficulty, let’s just create a method like this, add it to the menu and run it:

public static void LoadCategories()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                Administrator.SaveListDisableIdentity(db.Categories.Select(s =>

                    new CategoryEntity

                    {

                        CategoryName = s.CategoryName,

                        Description = s.Description,

                        Picture = new FileEmbedded {

FileName = s.CategoryName + ".jpg",

BinaryFile = EmployeeLoader.RemoveOlePrefix(s.Picture.ToArray()) },

                    }.SetId(s.CategoryID)));

            }

        }

Loading products it’s just a little bit more complicated. Let’s create a similar method:

public static void LoadProducts()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                Administrator.SaveListDisableIdentity(db.Products.Select(s =>

                    new ProductEntity

                    {

                        ProductName = s.ProductName,

                        Supplier =  Lite.Create<SupplierEntity>(s.SupplierID.Value),

                        Category = Lite.Create<CategoryEntity>(s.CategoryID.Value),

                        QuantityPerUnit = s.QuantityPerUnit,

                        UnitPrice = s.UnitPrice.Value,

                        UnitsInStock = s.UnitsInStock.Value,

                        ReorderLevel = s.ReorderLevel.Value,

                        Discontinued = s.Discontinued,

                    }.SetId(s.ProductID)));

            }

        }

Note that the Northwind database allows null values in some fields that should not, but since the data does not contain any null we can safely use Value property.

Also, in the case of Suppliers and Category we create again the Lites manually, and use the fact that we are loading entities with the legacy Ids.

If we try to run this code we will get some validation errors, all of them like this:

Units In Stock has to be greater than 0

There’s a tiny bug in our UnitsInStock validator, we should allow 0 to be valid. Let’s change ComparisonType.GreaterThan for ComparisonType.GreaterThanOrEqual.

With this fix we should be able to load Products (no changes in the schema needed).

### Load Customers

In the previous tutorial we take the decision to split customers in two different classes, PersonEntity and CompanyEntity. You can invent whatever business reason for that, but we did it just to explain how inheritance works in Signum Framework.

Now, in the load application, we have to split the data using some criteria. We have chosen to make all the customers witch ContactTitle is “Owner” to be PersonsEntity, otherwise CompanyEntity.

Now let’s start by loading the Companies. As usual, we create a CustomerLoader static class and we add a method like this:

public static void LoadCompanies()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                db.Customers.Where(c => !c.ContactTitle.Contains("Owner")).Select(c =>

                    new CompanyEntity

                    {

                        CompanyName = c.CompanyName,

                        ContactName = c.ContactName,

                        ContactTitle = c.ContactTitle,

                        Address = new AddressEmbedded

                        {

                            Address = c.Address,

                            City = c.City,

                            Region = c.Region,

                            PostalCode = c.PostalCode,

                            Country = c.Country,

                        },

                        Phone = c.Phone.Replace(".", " "),

                        Fax = c.Fax.Replace(".", " "),

                    }).SaveList();

            }

        }

Note that this time we are not trying to keep the old ID because it’s a sequence of letters for customers, instead we use *Database.SaveList*, also an extension method.

Note how we use our experience to fix the problems that we had with dots on phone and fax numbers.

Also we should allow Fax numbers to be optional this time (both in the field and in the property) and if we take a look at the data ContactTitle is little bit longer than what we expected when we created the entities (10 characters) so let’s make it 30 instead.

These changes will affect the schema, so let’s create a sync script before.

ALTER TABLE Company ALTER COLUMN ContactTitle NVARCHAR(30) NOT NULL;

ALTER TABLE Company ALTER COLUMN Fax NVARCHAR(24) NULL;

ALTER TABLE Person ALTER COLUMN Fax NVARCHAR(24) NULL;

If we try to load we will get an error:

Postal Code is not set

There’s a Company in Ireland that has no postal code, a little bit of research and you will notice that Irish people are not using postal codes (Live is so complex!) so let’s improve our AddressEmbedded to know about that.

We don’t want to make PostalCode optional just because of Ireland, what we will do instead is to make it optional if the country is called “Ireland”. In order to do that we will have to set AllowNulls = true in the validator and remove NotNullable in the field like this:

[SqlDbType(Size = 10)]

        [StringLengthValidator(AllowNulls = true, Min = 3, Max = 10)]

        public string PostalCode { get; set; }

But then we can override PropertyValidation in our Address class, as we did in the last tutorial, like this:

protected override string PropertyValidation(PropertyInfo pi)

    {

        if (pi.Name == nameof(PostalCode))

        {

            if (string.IsNullOrEmpty(PostalCode) && Country != "Ireland")

                return Signum.Entities.ValidationMessage.\_0IsNotSet.NiceToString(pi.NiceName());

            }

            return null;

        }

}

See how we used the resources in *Signum.Entities* to create the error message so we take advantage of all the localized error messages.

We also used the extension method *NiceName* over PropertyInfo. Signum framework provides an infrastructure to localize Type and Property names and enum values by using attributes, resource files, and the *NiceName/NiceToString* methods. These localizations will be used in the user interface, dynamic queries, error messages and auto generated help.

Some of these changes will modify our schema, so let’s sync:

ALTER TABLE Company ALTER COLUMN Address\_PostalCode NVARCHAR(10) NULL;

ALTER TABLE Employee ALTER COLUMN Address\_PostalCode NVARCHAR(10) NULL;

ALTER TABLE Supplier ALTER COLUMN Address\_PostalCode NVARCHAR(10) NULL;

ALTER TABLE Person ALTER COLUMN Address\_PostalCode NVARCHAR(10) NULL;

ALTER TABLE Order ALTER COLUMN ShipAddress\_PostalCode NVARCHAR(10) NULL;

And if we try again, now the companies should load without problems.

### CorrupEntity

Let’s focus on PersonEntity now. A similar method like the one before should make the work, let’s try:

public static void LoadPersons()

        {

            using (NorthwindDataContext db = new NorthwindDataContext())

            {

                db.Customers.Where(c => c.ContactTitle.Contains("Owner")).Select(c =>

                     new PersonEntity

                     {

                         FirstName = c.ContactName.Substring(0, c.ContactName.LastIndexOf(' ')),

                         LastName = c.ContactName.Substring(c.ContactName.LastIndexOf(' ') + 1),

                         DateOfBirth = null,

                         Title = null,

                         Address = new AddressEmbedded

                         {

                             Address = c.Address,

                             City = c.City,

                             Region = c.Region,

                             PostalCode = c.PostalCode,

                             Country = c.Country,

                         },

                         Phone = c.Phone.Replace(".", " "),

                         Fax = c.Fax.Replace(".", " "),

                         Corrupt = true,

                     }).SaveList();

            }

        }

Note how we take now the Customers witch name is Owner, and how we split the ContactName in first and last name.

We have a problem however: we have no DateOfBirth or Title.

Let’s pretend there’s an important business requirement to send birthday letters automatically to our Person customers, and for that we need those fields. In this case, however, asking them the birth date is not an option.

What we would like is to be able to load the customers, but next time they make an Order the fields are shown as an error so they have to fix it.

Signum Framework allows this behavior by using the Corruption class and DisableOnCorrupt property on our validation attributes.

The first thing we have to do is to allow title to have nulls on the database by removing NotNullable attribute on the field, and on the StringLengthValidator set DisableOnCorrupt=true.

Should look like this:

[SqlDbType(Size = 10)]

string title;

[StringLengthValidator(AllowNulls = true, Min = 3, Max = 10, DisableOnCorrupt=true)]

public string Title

{

get { return title; }

set { Set(ref title, value, () => Title); }

}

Also, we have to make DateOfBirth nullable (on the property and on the field also). And add a NotNullValidator on the property that is also DisableOnCorrupt.

This will be enough if we could inheriting from CorruptEntity, but in this case is not an option since PersonEntity is already inheriting CustomerEntity, so we will have to program the pattern manually.

To do that, we have to create a new field of type bool in PersonEntity with name ‘Corrupt’. Should look like this:

public bool Corrupt { get; set; }

This is a normal entity field that will be also useful to know which PerdonDN contains invalid data.

Then we have to override the way validation is done globally on the entity:

public override string IdentifiableIntegrityCheck()

{

using (this.Corrupt ? Corruption.Allow() : null)

{

return base.IdentifiableIntegrityCheck();

}

}

This way, if the entity is set to corrupt by the user, then the validation rules will be executed in a context where Corruption is Allowed, this will affect the validation attributes that have DisableOnCorrupt=true, but you can also use this context information in you custom validation logic using Corruption.Strict property.

Finally, we have to find a way to set Corrupt = false automatically if the entity passes validations in a Strict mode. We can do that by overriding *PreSaving* method like this:

protected internal override void PreSaving(Action graphModified)

{

base.PreSaving(ref graphModified);

if (this.Corrupt && string.IsNullOrEmpty(base.IdentifiableIntegrityCheck()))

{

this.Corrupt = false;

}

}

Doing this should be enough, let’s sync the schema:

ALTER TABLE Person ALTER COLUMN Title NVARCHAR(10) NULL;

ALTER TABLE Person ALTER COLUMN DateOfBirth DATETIME NULL;

ALTER TABLE Person ADD Corrupt BIT NOT NULL; -- DEFAULT( );

Since we don’t have any person yet we don’t need to write a DEFAULT value for Corrupt.

Finally, before we try to load persons we have to set Corrupt = true to allow corruption for every entity. If the entity is not corrupted it will be turn off automatically.

With this code PersonEntity customers should be able to get loaded.

### Hooking in the Engine

In order to give more control and expansion points to the programmer, the engine exposes to ways to hook user code before or after actions of the engine:

* Overriding methods on the entities: convenient when the code to run has no dependencies to the database or any other resource only available in the server.
* Handling events exposed in *EntityEvents<T>* class: convenient if those dependencies exist.

Note: There’s a per-class set of *EntityEvents* accessible using *EntityEvent<T>()* method and a global one using *EntityEventsGlobal*, both on the Schema object,

**PreSaving:**This convenient method will be called for every entity in the graph before saving. The method get’s called also before validation.

In the case that the method sets properties of type entity or collection (not just values) and the graph gets modified you should set graphModified parameter to true (don’t set it to false!) in order to validate and save the changes.

*EntityEvents’* also has an event that allows a server-side version of *PreSaving*, as well as a Saving event that is thrown afterwards. The exact order of events is like this:

Save sequence:

1.- Graph created

2.- Entitie’s *PreSaving* virtual method. If graphModified = true, graph is re-created

3.- EntityEvent’s *PreSaving* event. If graphModified = true, graph is re-created

4.- Entity graph gets validated.

5.- EntityEvent’s *Saving* event.

6.- INSERT / UPDATE in database.

**PostRetriving**   
A symmetric virtual method, *PostRetrieving*, is available whenever you want to execute code after the entity is retrieved from the database. On EntityEvents, there’s also another pair of methds, they get executed in this order:

1.- EntityEvents’ *Retriving* event.   
2.- Retrieve from database.  
3.- Entitie’s *PostRetrieving* method.  
4.- EntityEvents’ *Retrived* event.

**Deleting**

Delete event on EntityEvents gets fired before deleting a set of entities.

**FilterQuery**

Finally, FilterQuery event allows to filter globally all the queries by adding a hidden Where every time a *Database.Query<T>* gets invoked. This filter is only available per-class, not globally.

### Loading Shippers

Loading shippers should not have any trouble. Let’s create a new OrderLoader class and add a method like this one:

public static void LoadShippers()

       {

           using (NorthwindDataContext db = new NorthwindDataContext())

           {

               Administrator.SaveListDisableIdentity(db.Shippers.Select(s =>

                   new ShipperEntity

                   {

                       CompanyName = s.CompanyName,

                       Phone = s.Phone,

                   }.SetId(s.ShipperID)));

           }

       }

We add it to the menu and should work.

### Loading Orders

Just as we did when we created the entities, in order of dependencies, we will finish loading our Orders.

Loading CustomerEntity has some minor difficulties:

* We forget to create the Freight property, of type decimal, in the last tutorials, let’s do it now.
* We will need an embedded sub-query in order to load the OrderDetail collection.

A code like this should do the work:

public static void LoadOrders()

  {

      using (NorthwindDataContext db = new NorthwindDataContext())

      {

         var northwind = db.Customers.Select(a => new { a.CustomerID, a.ContactName }).ToList();

          var companies = Database.Query<CompanyEntity>().Select(c => new

          {

               Entity = (CustomerEntity)c,

               c.ContactName

          }).ToList();

          var persons = Database.Query<PersonEntity>().Select(p => new

          {

               Entity = (CustomerEntity)p,

               ContactName = p.FirstName + " " + p.LastName

          }).ToList();

          Dictionary<string, CustomerEntity> customerMapping =

                    (from n in northwind

                     join s in companies.Concat(persons) on n.ContactName equals s.ContactName

                     select KVP.Create(n.CustomerID, s.Entity)).ToDictionary();

          var orders = db.Orders.ToList().Select(o => new OrderEntity

          {

                    Employee = Lite.Create<EmployeeEntity>(o.EmployeeID.Value),

                    OrderDate = o.OrderDate.Value,

                    RequiredDate = o.RequiredDate.Value,

                    ShippedDate = o.ShippedDate,

                    State = o.ShippedDate.HasValue ? OrderState.Shipped : OrderState.Ordered,

                    ShipVia = Lite.Create<ShipperEntity>(o.ShipVia.Value),

                    ShipName = o.ShipName,

                    ShipAddress = new AddressEmbedded

                    {

                        Address = o.ShipAddress,

                        City = o.ShipCity,

                        Region = o.ShipRegion,

                        PostalCode = o.ShipPostalCode,

                        Country = o.ShipCountry,

                    },

                    Freight = o.Freight.Value,

                    Details = o.Order\_Details.Select(od => new OrderDetailEmbedded

                    {

                        Discount = (decimal)od.Discount,

                        Product = Lite.Create<ProductEntity>(od.ProductID),

                        Quantity = od.Quantity,

                        UnitPrice = od.UnitPrice,

                    }).ToMList(),

                    Customer = customerMapping.GetOrThrow(o.CustomerID),

                    IsLegacy = true,

                }.SetId(o.OrderID));

                orders.BulkInsert(disableIdentity: true, validateFirst: true, message: "auto");

        }

   }

There’s however a bigger problem we will have to face: loading Customer property.

Loading customers is harder for three reasons:

* It’s a full entity, not a Lite, so we will need to retrieve it first.
* Its’ a polymorphic relationship. Sometimes the customer will be a PersonEntity, sometimes a CompanyEntity.
* The IDs do not match this time.

Since Signum Framework forces the entities to have a primary key of type int and name Id, we were unable to keep the old Id’s for Customers. Now we have to face the problem of translating old Ids like ‘ALFKI’, ‘ANATR’ to pairs like [PersonEntity, 1], [CustomerEntity, 3], maybe using a dictionary like this:

Dictionary<string, Tuple<Type, int>> customerMapping;

Signum Framework however already provides an elegant way of using this Type-Id pairs: Lites

In fact, a Lites variable has an static type (the *T* of a *Lite<T>*) and run-time type (EntityType). This is possible because Lite<T> is a covariante interface.

This way we can represent a *Lite<CustomerEntity>* that refers t to the PersonEntity with Id=4, or the CompanyEntity with Id=1. So instead we will use this dictionary:

Dictionary<string, CustomerEntity> customerMapping;

In order to fill such dictionary we need to make a query that get’s old ID (Northwind) and Lites (Southwind). Unfortunately none of the Linq providers is able to reach the other database, so we will have to join by ContactName in-memory. Let’s start by querying Northwind:

var northwind = db.Customers.Select(a => new { a.CustomerID, a.ContactName }).ToList();

So for companies it will be just:

var companies = Database.Query<CompanyEntity>().Select(c => new

                {

                    Entity = (CustomerEntity)c,

                    c.ContactName

                }).ToList();

And for PersonsEntity we will have to re-compose the ContactName:

var persons = Database.Query<PersonEntity>().Select(p => new

                {

                    Entity = (CustomerEntity)p,

                    ContactName = p.FirstName + " " + p.LastName

                }).ToList();

Finally, in order to join the lists in-memory and create a dictionary we will need a query like this:

Dictionary<string, CustomerEntity> customerMapping =

                    (from n in northwind

                     join s in companies.Concat(persons) on n.ContactName equals s.ContactName

                     select KVP.Create(n.CustomerID, s.Entity)).ToDictionary();

Dirty Hack # 1  
I’ve to recognize now that joining the data by ContactName is, at least, a dangerous hack.

A safer possibility would have been to include and OldCustomerID property of type string on our CustomerEntity (so it will create fields on Person and Company tables). After loading the entities, we could remove the property and sync, or just keep it for future references.

In this case, however, I choose to do it this way to show how to join data in-memory from different databases.

Before loading we will have to synchronize to create our new Freight property.

ALTER TABLE Order ADD Freight DECIMAL(18,2) NOT NULL -- DEFAULT( );

And then, when we try to load our Orders, we get some nasty errors, all like this:

Discount should be multiple of 5%

It looks like there are some OrderDetails that do not follow the rules:

* 1 with 6%
* 1 with 4%
* 3 with a 3% discount
* 1 with 1% and

We cannot use a Corrupt entity this time, because these problems cannot be fixed (since this will affect the price of the off the Order, and this is not an option).

Removing the 5% multiple validation rules is also not allowed since we don’t want this kind of discounts in the future.

What we do then?

The only solution is to create a new IsLegacy property in the orders, and only allow this kind of discounts on legacy orders. This looks easier than it is.

In order to do that we have to modify the PropertyValidation we did on OrderDetailEmbedded, disabling if when the Order is legacy…. But there’s no reference from OrderDetailEmbedded to its parent!

There are many different solutions to our crisis:

We could create a reference from OrderDetailEmbedded to OrderEntity, but it will create redundancy in our data model. Even if we avoid the reference to be represented in the database (by adding an IgnoreAtrtribute over the field) we still have to manage the complexity of keeping this redundancy.

Other option will be to create a ValidateDiscount event on OrderDetailEmbedded that is captured by OrderEntity, but attaching and detaching the event will have the same complexity.

Fortunately, *Signum.Entities* has a declarative way to do this. Just by adding the attribute [NotifyChildProperty] over the ‘Details’ field on OrderEntity, and override ChildPropertyValidation, we can control the validation messages of child entities properties.

The base entity will manage all this complexity to keep the event attached.

In this case, let’s move the code from OrderDetailEmbedded.PropertyValidation to OrderEntity.ChildPropertyValidation and let’s make some changes like this:

protected override string ChildPropertyValidation(ModifiableEntity sender, PropertyInfo pi)

     {

    if (sender is OrderDetailEmbedded details && !IsLegacy && pi.Name == nameof(details.Discount))

            {

                if ((details.Discount \* 100.0m) % 5.0m != 0)

                    return OrderMessage.DiscountShouldBeMultpleOf5.NiceToString();

            }

            return base.ChildPropertyValidation(sender, pi);

    }

The last thing will be to set IsLegacy to true in our LoadOrders’ query.

Finally, let’s just sync the database again to add the latest changes.

ALTER TABLE Order ADD Freight DECIMAL(18,2) NOT NULL -- DEFAULT( );

ALTER TABLE Order ADD IsLegacy BIT NOT NULL -- DEFAULT( );

And with this code the orders should be able to get into your database!

## Conclusion

In exchange for centralized validation, code-first approach, synchronization of the schema and all the productivity gains on the user interface, Signum Framework forces you to create a new schema and load data into it.

There are not a lot of tutorials out there to show you how to load legacy data, and since it’s a mandatory step in Signum Framework we thought was fair to guide you through this process.

However, since Southwind schema was designed as a mimic of Northwind there were not really big transformations of data, but we saw some interesting tricks though:

* Keeping legacy Ids
* Using CSV files to complete our data
* Merging information from different databases and dealing with legacy non-numeric IDs
* Dealing with Lites and inheritance.

Also, we learn more things about what the validation system is capable of in order to let out data get in:

* Using Attributes and PropertyValidation (again)
* Using DisableOnCorrupt to disable some validation rules for some entities, so we can delay fixing some legacy data after going to production.
* Using ChildPropertyValidation to make a parent entity add validation rules over child entities.

### NotifyCollectionChangedAttribute

When placed over a MList field, keeps the protected **method** **ChildCollectionChanged** of the current entity subscribed to the MList’s **CollectionChanged** **event**.

### NotifyChildPropertyAttribute

When placed over a ModifiableEntity field, or a MList of ModifiableEntities, keeps the protected **method** **ChildPropertyChanged** subscribed to the entitie’s **PropertyChanged event** .

In order to keep this event’s attached, the entities does the following:

* Attach the event every time the entity is set in the property, added to the collection, or the whole collection is set.
* Detach the event if the entity is cleared from the property, remove from the collection, or the whole collection is cleared.
* Avoid the events field to be serialized. ([field:NonSerialized])
* Re-attach after deserialization.
* Avoid the event field to be stored in the database ([field:IgnoreAttribute])
* Re-attach in *PostRetrieving*.

In the next tutorials we will focus on the user interface. We will a web application using Typescript, React and Web.API, or a WPF application using Signum.Windows.