# Signum Framework Tutorials - Part 3: Southwind Terminal

## 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 fears 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 on any kind of invariant that the data should hold, making any modification way harder.

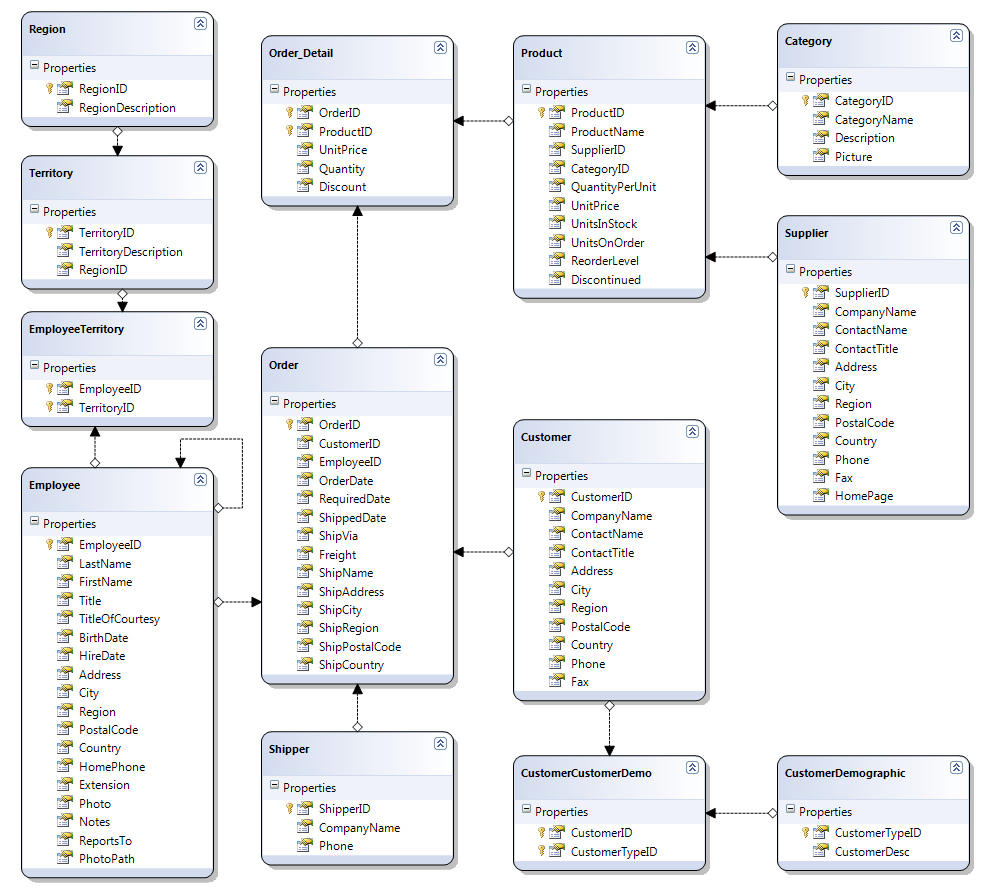
Signum Framework gives you the tools to make the new application right, but also helps you move the legacy data in the new schema by using *Signum.Utilities*, LINQ, and CSV files.

Northwind database is simplistic but is a good schema, reasonably normalized, the tables are properly 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.Terminal, 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:



To read the legacy database (Northwind) we will use IView interface. This interface allows to make queries to DB tables or views using Database.View<T>, without all unnecessary features of tables owned by entities (synchronization, validation, concurrency changes, etc.)

First, we need to create a secondary Connector for the Northwind database:

public static class Northwind

{

static Connector? connector;

public static Connector Connector

{

get

{

if (connector != null)

return connector;

var northwindConnectionString = Program.Configuration.GetConnectionString("NorthwindConnectionString")!;

return connector = new SqlServerConnector(northwindConnectionString, new SchemaBuilder().Schema, SqlServerVersion.SqlServer2012);

}

}

}

Then we can automatically generate all the IViews with the following code:

public static void GenerateViewCode()

{

using (Connector.Override(Northwind.Connector))

{

var myCode = Administrator.GenerateViewCodes(

"Categories",

"Suppliers",

"Customers",

"Employees",

"EmployeeTerritories",

"Orders",

"Order Details",

"Products",

"Region",

"Shippers",

"Territories"

);

}

}

Just placing a breackpoint in myCode and copy-paste the generate string below the northwind class:

[TableName("dbo.Categories")]

public class Categories : IView

{

[ViewPrimaryKey]

public int CategoryID;

public string CategoryName;

public string? Description;

public byte[]? Picture;

}

…

[TableName("dbo.Territories")]

public class Territories : IView

{

[ViewPrimaryKey]

public string TerritoryID;

public string TerritoryDescription;

public int RegionID;

}

Let’s start loading some data!

### Loading Regions

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

Then, inside of the method we override the connecto and query the Region table, creating a new RegionEntity for each result like this:

public static void LoadRegions()

{

var regions = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Region>().ToList());

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.

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

The Administrator static class has many useful (but sometimes dangerous) methods for manipulating the database schema, and typically should not be used in production code, but it’s quite useful for this kind of dirty hacks in the Terminal application.

The code will look like this:

public static void LoadRegions()

{

var regions = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Region>().ToList());

Administrator.SaveDisableIdentity(regions.Select(r => new RegionEntity

{

Description = r.RegionDescription.Trim()

}.SetId(r.RegionID)));

}

We could improve the code even more using BulkInsert. For simple saves of many entities that are only going to affect one entity (one table and his MList tables) **BulkInsert is often more than 10x faster.** And is very easy to use!

public static void LoadRegions()

{

var regions = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Region>().ToList());

regions.Select(r => new RegionEntity

{

Description = r.RegionDescription.Trim()

}.SetId(r.RegionID))

.BulkInsert(disableIdentity: true);

}

If we run the Terminal application, choose load and pick the first method (0 – Load Regions), or add it to our C# Migrations, the code will be run, and the new records will be in the database.

### Loading Territories

Let’s make a similar method for territories:

public static void LoadTerritories()

{

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

var territories = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Territories>().ToList());

var entities = territories.Select(t => new TerritoryEntity

{

Region = regionDic.GetOrThrow(t.RegionID),

Description = t.TerritoryDescription.Trim(),

}.SetId(new PrimaryKey(t.TerritoryID)))

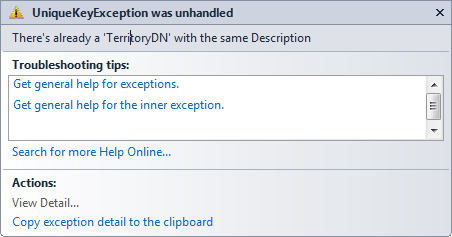
.ToList();

entities.BulkInsert(disableIdentity: true);

}

Here we use a dictionary from Id to Region. This way we only need to query Region table once instead of once per Territory. **Reducing queries and using dictionaries effectively are the best way to improve performance in Signum Framework applications.**

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



Looks like NewYork is written twice in Northwind database. To remove duplicates, we can just use:

entities.Duplicates(a => a.Description).ForEach(t => t.Description += " (Dup)");

### 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.

public static void LoadEmployees()

{

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

var employees = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Employees>()

.Select(e => new

{

e.ReportsTo,

employee = new EmployeeEntity

{

BirthDate = e.BirthDate.ToDateOnly(),

FirstName = e.FirstName,

LastName = e.LastName,

TitleOfCourtesy = e.TitleOfCourtesy,

HomePhone = e.HomePhone,

Extension = e.Extension,

HireDate = e.HireDate.ToDateOnly(),

Photo = new FileEntity { FileName = e.PhotoPath!.AfterLast('/'), BinaryFile = RemoveOlePrefix(e.Photo!.ToArray()) }.ToLiteFat(),

Address = new AddressEmbedded

{

Address = e.Address!,

City = e.City!,

Country = e.Country!,

Region = e.Region,

PostalCode = e.PostalCode,

},

Notes = e.Notes,

Territories = Database.View<NW.EmployeeTerritories>()

.Where(et => et.EmployeeID == e.EmployeeID)

.Select(a => territoriesDic.GetOrThrow(int.Parse(a.TerritoryID)))

.ToMList(),

}.SetId(e.EmployeeID)

})

.ToList());

Administrator.SaveListDisableIdentity(employees.Select(a=>a.employee));

var dictionary = employees.Select(a => a.employee).ToDictionary(a => a.Id);

foreach (var pair in employees)

{

pair.employee!.ReportsTo = pair.ReportsTo == null ? null : dictionary.GetOrThrow(pair.ReportsTo!.Value).ToLite();

}

dictionary.Values.SaveList();

}

Some remarks about this method:

* It creates a dictionary of TerritoryEntity to avoid multiple queries, great!.
* It needs to convert many old DateTime without time part into the new DateOnly using the ToDateOnly extension method.
* The Photo creates a new FileEntity, defined in Signum.Files. This means that for each row we need to save two entities (EmployeeEntity and FileEntity) so using BulkInsert will be complicated. Also a helper method is used to remove the OleObject header from Access.
* Many columns require the ! operator to cast to not-nullable because Northwind schema is too liberal on nullability, but the data is OK.
* We use a nested query for the Database.View<NW.EmployeeTerritories>() they are implemented efficiently in the LINQ provider so only two (instead of N+1) queries will be made.

The last problem 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.

This is how we do it:

* When querying the Employee table we directly construct the EmployeeEntity but we keep the ReportsTo property aside in an anonymous object.
* Save all the new employees.
* Create a dictionary from id to employee, and use it to set the ReportsTo dictionary, using ToLite to convert an EmployeeEntity to a Lite<EmployeeEntity>

### 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()

{

var suppliers = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Suppliers>().ToList());

suppliers.Select(s => new SupplierEntity

{

CompanyName = s.CompanyName,

ContactName = s.ContactName,

ContactTitle = s.ContactTitle,

Phone = s.Phone!,

Fax = s.Fax!,

Address = new AddressEmbedded

{

Address = s.Address!,

City = s.City!,

Region = s.Region,

PostalCode = s.PostalCode,

Country = s.Country!

},

}.SetId(s.SupplierID))

.BulkInsert(disableIdentity: true);

}

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

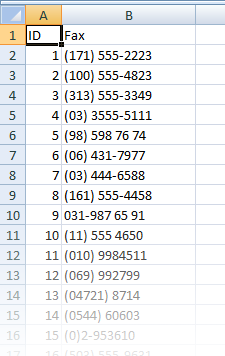
The first two errors are produced because Telephone validator does not allow dots (‘.’) on the telephone number, only numbers, space, hyphen, and parenthesis. In this case let’s just replace the dots with space.

If we look at the Northwind data, we see that HomePage field contains very few heterogeneous data that is not worth loading. 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 must 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.Terminal 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 look to the file. Depending on 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 is UTF-8.

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

Note how we write all the parameters explicitly.

* We let the encoding empty, defaults to UTF-8.
* 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()

{

var suppliers = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Suppliers>().ToList());

List<SupplierFaxCSV> faxes = Csv.ReadFile<SupplierFaxCSV>("SupplierFaxes.csv", culture: CultureInfo.GetCultureInfo("es-ES"));

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

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))

.BulkInsert(disableIdentity: true);

}

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()

{

var category = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Categories>().ToList());

category.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))

.BulkInsert(disableIdentity: true);

}

Loading Products is just a little bit more complicated. Let’s create a similar method:

public static void LoadProducts()

{

var products = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Products>().ToList());

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))

.BulkInsert(disableIdentity: true);

}

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 Terminal application, we must 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()

{

var companies = Connector.Override(NW.Northwind.Connector).Using(\_ =>

Database.View<NW.Customers>()

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

.ToList());

companies.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 == null ? null : c.Fax.Replace(".", " "),

}).BulkInsert();

}

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()

{

var persons = Connector.Override(NW.Northwind.Connector).Using(\_ =>

Database.View<NW.Customers>()

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

.ToList());

persons.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 == null ? null : c.Fax.Replace(".", " "),

}).BulkInsert();

}

Check how we take the Customers witch ContactTitle 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.

To do that, we need to make the new properties nullable (so the column is nullable), but we can still explicitly add the NotNullValidator:

[StringLengthValidator(Min = 3, Max = 10), NotNullValidator]

public string? Title { get; set; }

[NotNullValidator]

public DateOnly? DateOfBirth { get; set; }

After that, we need to make the PersonEntity corruptible by adding the CorruptMixin.

[EntityKind(EntityKind.Shared, EntityData.Transactional), Mixin(typeof(CorruptMixin))]

public class PersonEntity : CustomerEntity

{

And finally, we need to set the IsApplicable lambda of the NotNullValidationAttributes to make them optional when corrupt like this:

static PersonEntity()

{

Validator.PropertyValidator((PersonEntity p) => p.DateOfBirth).IsApplicableValidator<NotNullValidatorAttribute>(p => Corruption.Strict);

Validator.PropertyValidator((PersonEntity p) => p.Title).IsApplicableValidator<NotNullValidatorAttribute>(p => Corruption.Strict);

}

The CorruptMixin will add a new Corrupt column to the table, and when saving will make two changes:

* When saving set the Corruption.Strict scope variable only if the entity is not Corrupted, so that some validation rules can be skipped.
* Before saving, try to check if the entity has been fixed by validating in strict mode, in this case, set 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 DATE NULL;

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

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

### Hooking in the Engine

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()

{

var shippers = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Shippers>().ToList());

shippers.Select(s => new ShipperEntity

{

CompanyName = s.CompanyName,

Phone = s.Phone!,

}.SetId(s.ShipperID))

.BulkInsert(disableIdentity: true);

}

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 also will need sub-query in order to load the OrderDetail collection.

A code like this should do the work:

public static void LoadOrders()

{

var orders = Connector.Override(NW.Northwind.Connector).Using(\_ => Database.View<NW.Orders>().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 = Database.View<NW.OrderDetails>().Where(od => od.OrderID == o.OrderID).Select(od => new OrderDetailEmbedded

{

Discount = (decimal)od.Discount,

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

Quantity = od.Quantity,

UnitPrice = od.UnitPrice,

}).ToMList(),

Customer = /\*What we do here??\*/,

IsLegacy = true,

}.SetId(o.OrderID)).ToList());

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.
* It’s a polymorphic relationship. Sometimes the customer will be a PersonEntity, sometimes a CompanyEntity.
* The IDs do not match this time.

Since we were unable to keep the old Ids for Customers, we will have to find them by contact name, again using a dictionary:

Dictionary<string, CustomerEntity> customers = new Dictionary<string, CustomerEntity>();

customers.AddRange(Database.Query<CompanyEntity>()

.Select(c => KeyValuePair.Create(c.ContactName, (CustomerEntity)c)));

customers.AddRange(Database.Query<PersonEntity>()

.Select(p => KeyValuePair.Create(p.FirstName + " " + p.LastName, (CustomerEntity)p)));

Finally, we can find the name in Northwind database by doing a sub-query like this.

Customer = customers.GetOrThrow(Database.View<NW.Customers>()

.Where(c => c.CustomerID == o.CustomerID)

.Select(a => a.ContactName).SingleOrDefaultEx()!),

This query will be quite efficient, since the sub query only returning one value the whole expression will be embedded in the parent query, making only one query instead of two.

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!

Fortunately, *Signum.Entities* has a declarative way to do thi:

* Add the attribute [BindParent] over the ‘Details’ field
* Call this.BindParent() in OrderEntity constructor
* 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!

### BindParentAttibute

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

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 events 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.
* Re-attach after deserialization.
* Re-attach in *PostRetrieving*

## 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 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 CorruptMixin and Corruption.Struct 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.

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