# Signum Framework Tutorials Part 5 – Southwind Windows

## About Signum Framework 2.0

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.

We have just released Signum Framework 2.0 and we are preparing a series of tutorials to explain what is capable of.

## 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 (MVC) 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
* Signum Framework Tutorials Part 3 – Southwind Load
* Signum Framework Tutorials Part 4 – Southwind Web

In this tutorial we will close the circle and we will show how easy is to create a WPF user interface using WPF and Signum.Windows.

### Introduction

For those who were using Signum Framework 1.0, Signum.Windows is an old friend. For the new ones, Signum.Windows is the library with the controls and necessary infrastructure to do in WPF what we already did in the last tutorial.

Signum Framework is a cohesive library where all the pieces are designed to work nicely together. We tried to keep a consistent API and a philosophy between Signum.Web and Signum.Windows, simplifying moving from one to the other.

The two biggest pieces of Signum.Windows, just as in Signum.Web, are the edition controls (created using EntityControls) and the search windows. There’s also a similar Navigation class, EntitySettings, etc…

Being this said, we also try not to force the hosting platform (ASP.Net MVC or WPF) with unnatural conventions. We want to be good citizens both in Windows and Web, allowing the programmer to express their creativity whenever the resources of our libraries don’t fit the requirements.

Southwind.Windows, as opposed to Southwind.Web, runs in another machine and has no access to the database, neither the logic. However it uses a WCF service with simple .Net Serialization (NetDataContract) witch allows sending the entities through the service without losing the type information, the validations, the localization of the entities names and properties, etc…

No DTO (Data Transfer Objects) are necessary, but you are free to use them for certain scenarios if you want.

## Making Contact

The first thing to do is to load Southwind.Windows, the last unloaded project.

In there we can see a few files.

* App.Config: Contains the configuration of the client application, the most important thing is the URL of the service exposed in Southwind.Web.
* Program.cs: Contains the Main method that loads the application and connects with the server and define the global exception handler.
* App.xaml: In this XAML file we take control of some global aspects of the application, like registering views, initializing client modules, and register the global exception handler.
* Main.xaml: Contains the startup screen of the application, by default just a menu but you are free to do anything fancy there.
* Controls folder: In there we will place all the controls defined for our entities.

Also, maybe we didn’t realize in the last tutorial but Southwind.Web contains a ServerSouthwind.svc. This is the IIS-Hosted WPF Service and the contract of this service (IServerSouthwind) is defined in Southwind.Entities so both client and server have access to it.

Before starting, in order to run the Windows application we will need set-up our solution to start the Server (Signum.Web) and the Client (Signum.Windows) at the same time.

1.- Right click in Southwind solution

2.- Under Common Properties 🡪 Startup Project

3.- Multiple startup projects and choose Action to Start for Southwind.Windows and Southwind.Web

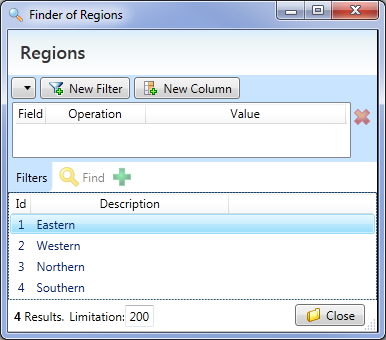
Also, let’s force our CultureInfo to English (en-US) for now since Northwind is an English application.

## Regions and Territories

In the Main.xaml menu we still have a line that makes reference to MyEntityDN, let’s replace it to RegionDN to point it to the query associated with typeof(RegionDN).

In App.xaml we have a line trying to registrer MyEntityDN EntitySettings, let’s comment it out.

If we try to run now we will see a simple menu, inside Entities we have a Regions submenu that will open a window like this:



This is the search window. It uses the queries that we registered in the DynamicQueryManager in our Logic code.

As in Signum.Web, it’s able to filter, sort, add, remove and rename columns. It’s also able to create new entities of this type (Regions) as well as navigating to the current one by double clicking on a row (Entity column).

If we try, however, we won’t go so far. The problem is that we have no view associated with RegionDN. Let’s create it:

In the Controls folder we have MyEntity.xaml with an example. Let’s get rid of it and generate a new one for Regions.

Code generation is a double-edged sword, you get a lot of work for free but then you have to maintain a lot of code also or, in the worst case, you cannot even change it.

In Signum.Windows, just as in any other part of the framework, we tried to encourage a clever runtime behavior over code generation, struggling to make EntityControls just one line of code for example, and then for the last step we have a Visual Studio Item Template to generate the edit control of every entity, saving you some time.

Let’s create the controls for Regions:

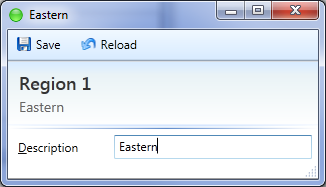
1. Right click in Controls folder
2. New Item
3. Select Signum Entity Control and click Add (don’t worry about the name)
4. A Dialog will ask for the assembly where you have the entities. Let’s click open and point to: {your folder}\Southwind\Southwind\Southwind.Entities\bin\Debug\Southwind.Entities.dll
5. In the second Combo, select Southwind.Entities.RegionDN, and click Ok
6. A dialog asks you to rename Entity1 to Region, answer Yes.

Finally, let’s turn back to App.xaml and create an EntitySettings line, similar to the one we just commented out:

new EntitySettings<RegionDN>(EntityType.Admin) { View = e => new Region() }

What we are doing here is associating RegionDN type with the Region control we just created. Also we set Region to be of type Admin. This will set some sensible default for it so it’s edited it in very special Admin scenarios (only on the Region’s SearchWindow mainly).

If we run again and open the Region search window, and double click on Easter Region we will see a fully functional window like this:



Let’s take a look at the generated code:

<UserControl x:Class="Southwind.Windows.Controls.Region"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:m="clr-namespace:Signum.Windows;assembly=Signum.Windows"

xmlns:d="clr-namespace:Southwind.Entities;assembly=Southwind.Entities"

m:Common.TypeContext="d:RegionDN"

MinWidth="300">

<StackPanel>

<m:ValueLine m:Common.Route="Description" />

</StackPanel>

</UserControl>

* The snippet created a normal UserControl
* It imported the default xmlns and xmln:x namespaces pointing to the standard WPF libraries. So far quite normal.
* It also included the Signum.Windows namespace with the prefix ‘m’,
* It also included your entities namespace with the prefix ‘d’.
* It sets m:Common.TypeContext attached property to the type d:RegionDN saying that this control expects DataContext of type RegionDN.
* It sets the MinWidth to 300, just so the window doesn’t crush.
* It creates a simple StackPanel as the layout for the EntityControls
* In this case, it adds a ValueLine pointing to Description string property using m:Common.Route attached property.

Now take a look to what we didn’t write:

* All the surrounding windows, widgets, etc… are defined in the shared NormalWindow, you only have to focus on the control for your entity. You could override NormalWindows if you don’t like the default one.
* The Save and Reload button are already fully functional and communicate with the server, but you can hide them if you want using EntitySettings.
* By using ValueLine, the right control for the right data type (string, numbers, dateTimes…) is determined automatically, as long as the type has value semantics. For entities, other controls are better suited (EntityLine, EntityCombo…). You can change witch control ValueLine uses for each data type if you want.
* By using m:Common.Route, a set of tasks has been executed, this pipeline is responsible of:
  + Binding the control to the property, automatically taking care of:
    - Bind the right target property, in this case ValueLine.Value.
    - Enable WPF built-in validation (NotifyOnValidationError, ValidatesOnDataErrors, ValidatesOnExceptions)
    - Add the necessary converters in special cases.
  + In the case of read-only properties, the control is automatically set to ReadOnly
  + The labelText of the control is set to the NiceName of the property, taking care of localization.
  + In the case of properties of type entity, using Common.Route also sets the Implementations property.
  + Updating TypeContext so m:Common.Route works on the inner controls as well.

The pipeline is extensible, so modules can add tasks with no code changes!

* + The Property Auth module (in Signum Extensions) uses it to hide or set as readonly some some properties.
  + The Help module is able to add contextual information related to the properties in the user interface.

Finally, by using m:Common.TypeContext instead of calling GetType over the object in DataContext, we give static type information to the controls in Signum.Windows. This way the controls can use it to make all this decisions on load time, not every time the DataContext changes.

After all this theory, let’s create a control for TerritoryDN and add them the appropriate EntitySettings to Navigator class as we did:

new EntitySettings<TerritoryDN>(EntityType.Admin) { View = e => new Territory() }

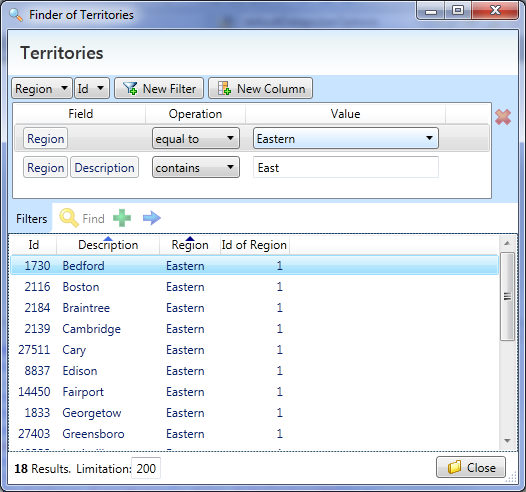
Finally let’s add another item in the menu:

<MenuItem Tag="{m:Explore QueryName={x:Type d:TerritoryDN}}"/>

If we try to run the application now and go to territories, we will see a search window similar to the last one.

Let’s play a little bit with it now:

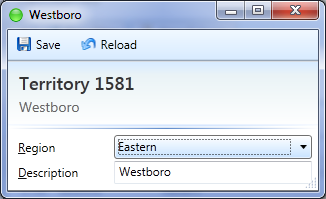
* If you click on every column, you can order results as expected. Shift-click on a column to add multiple orders.
* If you right click on a column and select New Filter, a new filter that sets this column to the current selected value will appear. You are free to change the operation (equal to, distinct to, greater than, contains, etc… depending of the type) or the value.
* You can remove the filter selecting the row all together and click the X on the top right side of the filter list.
* Note how depending of the field type of the filter the right control is set. Also, since the Region is a Lite<TerritoryDN>, not just a string, a combo with all the possible Regions is set automatically. If we would have set Region to be a EntityType.Default (instead of Admin) it will show the Region field as a link in the result panel making it easy to navigate to related entities.
* Finally, in the top buttons, select Region in the first combo, and notice how a second combo appears to dig into the Region entity, there selects Description and click New Filter. Look what a powerful and simple way of adding complex filters.
* It gets even more interesting when you realize you can add your own columns: Using the combos, select Region 🡪 Id and click New Column. Click Find and see how now a new “Id of Region” column shows us the values.



If we double-click on one result, we will open the newly created control. Apart from another ValueLine for edit description, we can now see the EntityCombo.

If Territory would have been EntityType.Default, the entity combo will also show a view button to see the entity, you can make other buttons appear just by explicitly setting them to true (Create, Find, View or Remove).

Also, notice how the EntityCombo is smart enough to look at the type of the property (RegionDN) and ask the server for the right kind of items.



This kind of automatic clever behavior, taking profit of the information available on the entities, it’s what makes Signum.Windows (and Signum.Web) so damn productive.

The controls usually have enough extension points to change the behavior when necessary (EntityCombo.LoadData allows changing the ItemSource for example) or you can chose not to use the control all together and build your own, but having a solid entity model allows you to make smarter controls that work ‘by itself’.

## Employees

Let’s try now with more complex entity. Let’s generate the control for EmployeeDN, and add EntitySettings for it as usual:

new EntitySettings<EmployeeDN>(EntityType.Admin) { View = e => new Employee() },

Then, let’s add it to the menu, and let’s rename the parent menu to Employees also, so we kept it organized.

<MenuItem Tag="{m:Explore QueryName={x:Type d:EmployeeDN}}"/>

If we run the application and take a look at it now the auto-generated control this will be the result:

<m:ValueLine m:Common.Route="UserName" />

<m:ValueLine m:Common.Route="PasswordHash" />

<m:ValueLine m:Common.Route="LastName" />

<m:ValueLine m:Common.Route="FirstName" />

<m:ValueLine m:Common.Route="Title" />

<m:ValueLine m:Common.Route="TitleOfCourtesy" />

<m:ValueLine m:Common.Route="BirthDate" />

<m:ValueLine m:Common.Route="HireDate" />

<GroupBox Header="Address" m:Common.Route="Address">

<StackPanel>

<m:ValueLine m:Common.Route="Address" />

<m:ValueLine m:Common.Route="City" />

<m:ValueLine m:Common.Route="Region" />

<m:ValueLine m:Common.Route="PostalCode" />

<m:ValueLine m:Common.Route="Country" />

</StackPanel>

</GroupBox>

<m:ValueLine m:Common.Route="HomePhone" />

<m:ValueLine m:Common.Route="Extension" />

<m:ValueLine m:Common.Route="Photo" />

<m:ValueLine m:Common.Route="Notes" />

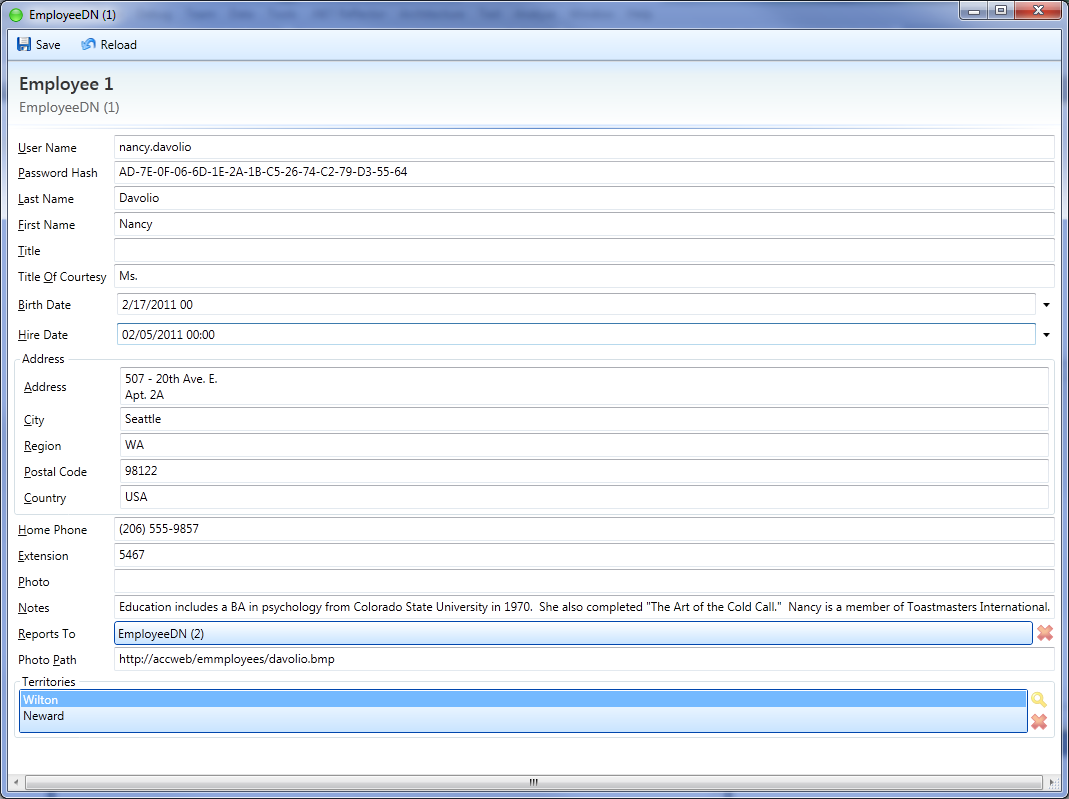
<m:EntityLine m:Common.Route="ReportsTo" />

<m:ValueLine m:Common.Route="PhotoPath" />

<GroupBox Header="Territories">

<m:EntityList m:Common.Route="Territories" />

</GroupBox>



## Improving Employee view

Even if the for is fully working, the design is,at least, spartan. There are plenty of improvements we can do to make happy this special girl/boy in HR department:

* Arrange the information
  + Group Last Name, First Name and Title of Courtesy on a sensible way.
  + Make a nice-looking Address control that we can re-use.
* Hide Password Hash, and make a button Change Password instead.
* Show the picture!
* Make the notes a text area, and maybe make it more note-looking by making it post-it yellow.

### Grouping names

In order to show the names in a more intuitive and nice-looking way, we will create a grid so all the textboxes are the same line and in the correct order (TitleOfCourtesy, FirsName, LastName) and the labels below.

ValueLine have no capability show de labels below, but what we can do is to hide the labels altogether (m:Common.LabelVisible) and place them manually in the lower row. We can, however, use attached property m:Common.Route to give the label Content the nice name of the property. The fewer sentences you write in the user interface, the less you will have to translate.

Finally, we will have to create a style for the labels so they look gray, slightly smaller and closer to the upper textbox. This is how the three ValueLines will look after our creativity rush:

<Grid m:Common.LabelVisible="False">

<Grid.Resources>

<Style TargetType="Label">

<Setter Property="FontSize" Value="11"/>

<Setter Property="Foreground" Value="Gray"/>

<Setter Property="VerticalAlignment" Value="Top"/>

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="Padding" Value="1"/>

</Style>

</Grid.Resources>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<m:ValueLine m:Common.Route="TitleOfCourtesy" />

<m:ValueLine m:Common.Route="FirstName" Grid.Column="1" />

<m:ValueLine m:Common.Route="LastName" Grid.Column="2"/>

<Label Content="Title" Grid.Row="1"/>

<Label m:Common.Route="FirstName" Grid.Row="1" Grid.Column="1" />

<Label m:Common.Route="LastName" Grid.Row="1" Grid.Column="2" />

</Grid>

### Address control

One nice thing of using EmbeddedEntity for AddressDN, instead of repeating the same fields and validators in every entity, is that now we can also create a control for Address and use it in every entity.

We can create the control for AddressDN using the same procedure we did before. Afterwards, we will change it to make a grid with a big textbox for the address with no label, and four ValueLines for City, Region, PostalCode and Country. A GroupBox with the name of the property will group the fields. Like this:

<UserControl x:Class="Southwind.Windows.Controls.Address"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:m="clr-namespace:Signum.Windows;assembly=Signum.Windows"

xmlns:d="clr-namespace:Southwind.Entities;assembly=Southwind.Entities"

m:Common.DelayedRoutes="True"

MinWidth="300">

<GroupBox m:Common.LabelOnlyRoute="">

<StackPanel>

<m:ValueLine m:Common.Route="Address" Grid.ColumnSpan="3" m:Common.LabelVisible="False" Grid.Column="0"/>

<Grid m:Common.LabelWidth="80">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<m:ValueLine m:Common.Route="City" Grid.Row="0" Grid.Column="0" />

<m:ValueLine m:Common.Route="Region" Grid.Row="0" Grid.Column="1" />

<m:ValueLine m:Common.Route="PostalCode" Grid.Row="1" Grid.Column="0" />

<m:ValueLine m:Common.Route="Country" Grid.Row="1" Grid.Column="1"/>

</Grid>

</StackPanel>

</GroupBox>

</UserControl>

Note how in the header, instead of m:Common.TypeContext, there’s a m:Common.DelayRoutes=”true” attribute. Controls for EmbeddedEntity are a little trickier and need this change. If you are interested in the details:

### PropertyRoute in Signum.Windows

In the last tutorial we saw the concept of PropertyRoute.

A PropertyRoute is a sequence properties that is able to point to a database column… short off. For example (EmployeeDN).Name, (EmployeeDN).Address.City and (SupplierDN).Address.City are three different PropertyRoutes. This concept is important especially for authorization purposes and allows that some roles are able to see the City field of the Employees but not for the Suppliers even with and EmbeddedEntity in between.

Every time you instantiate a control, the TypeContext determines the root PropertyRoute (usually a type that inherits from IdentifiableEntity) and by using m:Common.Route you concatenate Properties to this PropertyRoute.

Once your property route get’s into the properties of another IdentifiableEntity, it reset itself, so for example (EmployeeDN).ReportsTo.Name becomes (EmployeeDN).Name because we are speaking about the same database column.

This effect doesn’t happen to EmbeddedEntities though, because they generate different columns, that’s why we have to delay the Route concatenation from instantiation phase, to Loading phase, when the control actually knows where it is.

Also we use m:Common.LabelOnlyRoute=”” in the GroupBox. This properties does the same that Common.Route, but affects only the LabelText, Label or Header of the control. By using it with the empty string, we use the parent PropertyRoute (the one the Address UserControl is bound to).

### Change password dialog

Storing the password hash, instead of password, is a good practice, but showing it to the user makes no sense. Instead we will put a button next to the username to allow changing the password.

<DockPanel LastChildFill="True">

<Button Content="New Password" DockPanel.Dock="Right" VerticalAlignment="Center" x:Name="btnNewPassword" Click="btnNewPassword\_Click" />

<m:ValueLine m:Common.Route="UserName" Margin="1,0" />

</DockPanel>

Once the users clicks the button, a dialog will ask the user to repeat the password. If they are equal the value will be hashed and added to the Employee field.

### Showing the picture

Showing the picture should be as simple as adding an image and binding the source to the Photo property.

<Border Grid.Column="1" BorderThickness="1" BorderBrush="Gray" Margin="2" HorizontalAlignment="Center" VerticalAlignment="Center">

<Image x:Name="image" Source="{Binding Photo, Converter={x:Static s:SouthwindConverters.ImageConverter}}" Stretch="None" MinWidth="150" MinHeight="180" MouseUp="image\_MouseUp" />

</Border>

Notice that in order to convert the byte array to a valid ImageSource we have to create a converter. Using the Markup Extension ‘Static ‘ we can reach the static field that will contain our converter, that we can create easily using Signum’s ConverterFactory:We just need to provide the lambda that makes the conversion and store the converter in a static variable. We will group all our converters in SouthwindConverters static class.

public static class SouthwindConverters

{

public static IValueConverter ImageConverter =   
 ConverterFactory.New((byte[] array) =>

{

BitmapImage image = new BitmapImage();

image.BeginInit();

image.StreamSource = new MemoryStream(array);

image.EndInit();

return image;

});

}

There’s a problem thought. The photos loaded on Northwind database, apart from being old fashion and low quality, have some kind of 78-bits OLE header. We could try to detect it and skip them but what we will do, instead, is create a method in Southwind.Load to get rid of this prefixes:

public static void FixImages()

{

foreach (var employee in Database.RetrieveAll<EmployeeDN>())

{

if (employee.Photo != null)

{

byte[] clean = new byte[employee.Photo.Length - 78];

Array.Copy(employee.Photo, 78, clean, 0, employee.Photo.Length - 78);

employee.Photo = clean; // employee.Photo.Skip(78).ToArray();

employee.Save();

}

}

}

After adding the method to the menu, and running it, the images should look ok.

### Notes

Out last improvement will be to make notes field look like a note. We will use a TextArea control, that already set-ups a TextBox with AcceptTabs, AcceptReturns and TextWrap.

We will also use a GroupBox with m:Common.LabelOnlyRoute, this will fill the Header property of the control, as well as it will make it easier to hide the whole group altogether in the case the field is not visible (Signum.Authorization).

<GroupBox m:Common.LabelOnlyRoute="Notes" Background="#FFFFEEBB" Grid.Row="2" Grid.ColumnSpan="2" >

<m:ShyBorder Horizontal="True" Vertical="False">

<m:TextArea m:Common.Route="Notes" MinHeight="100" />

</m:ShyBorder>

</GroupBox>

Finally notice about the ShyBorder.

Even if we user a TextArea, that internally uses a TextBox with text wrapping enabled, the 2-pass layour system of WPF will ask for how much space it needs and the TextBox will prefer not to wrap if possible. Since this control is going to be hosted in a NormalWindow, and this window has automatic scrollbars just in case the control is too big, the layout system will please the TextBox and make it too wide if there are long sentences.

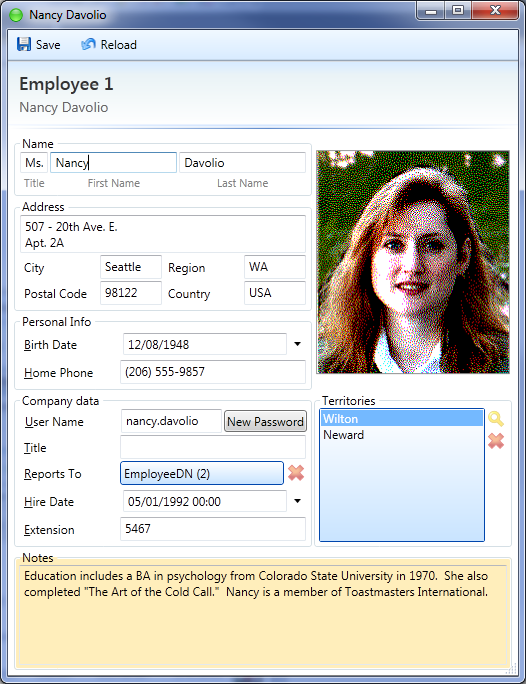
I don’t think there’s anything out of the box to fix this in WPF, so we ‘invented’ ShyBorder. What ShyBorder does is giving the nested control all the available space, but won’t allow him to ask for more. You can enable this behavior horizontally, vertically or both (default).

In this case since the GroupBox is in a Grid, it means that the remaining controls will determine the width, and the TextBox will take it all, but not more.

### Last improvements

This is how the Employee control looks after our new look change.

OVERRIDE TOSTRING in SIGNUM WEB



However, if you change the First or Last Name, the ToString in the Title bar and in gray below the button bar doesn’t update until you click save.

The reason for that is that we are not **notifying** that the ToString has changed every time we change the properties, just the property itself using Set method. This is something we can do in our EmployeeDN entity easily:

public string LastName

{

get { return lastName; }

set

{

if (Set(ref lastName, value, () => LastName))

Notify(() => ToStringMethod); // or NotifyToString();

}

}

Since the set returns true then the value changed (Equals is false) we can then Notify that ToStringMethod property changed, affecting the title bar (WPF can’t bind to methods). Since this is a common scenario there’s also a shortcut to do this using SetToStr method.

public string LastName

{

get { return lastName; }

set { SetToStr(ref lastName, value, () => LastName); }

}

## SearchControl Column Formatters

The last improvement we could make is not in the Employee control itself, but in the way the SearchWindow shows Employees images. It doesn’t make sense to show ‘System.Byte[]’ instead would be cool to show a tiny photo in there.

QuerySettings has two static members that will allow you to do that:

* PropertyFormatters that defines DataTemplate for columns associated with a particular PropertyRoute (this way it works even when you add the column from another query)
* FormatRules that defines general purpose DataTemplates based in the column type and attributes.

In this case we will use PropertyFormatters, there’s already a convenient method to register one that looks like this:

public static void RegisterPropertyFormat<T>(

Expression<Func<T, object>> property,   
Func<Binding, DataTemplate> formatter)

The fist argument allows you to specify the property with a strongly typed style, and in the second one you have to provide a DataTemplate for the column, given a Binding to the underlying ResultTable Column.

In our case we need to create a DataTemplate that creates a small image, and binds to the column with the appropriate converter.

QuerySettings.RegisterPropertyFormat((EmployeeDN e)=>e.Photo, b=>{

b.Converter = SouthwindConverters.ImageConverter;

return //Create DataTemplateHere

});

Creating a DataTemplate programmatically is not that easy in WPF. There are basically three ways:

* **Using XamlReader:** Is the recommended way by Microsoft, but personally I don’t like too much string literals in my code, double quotes and namespaces declarations.

QuerySettings.RegisterPropertyFormat((EmployeeDN e)=>e.Photo, b=>

{

var text = @"<DataTemplate

xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation""

xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml""

xmlns:s=""clr-namespace:Southwind.Windows;assembly=Southwind.Windows"">

<Image MaxHeight=""32"" Stretch=""Uniform"" Source=""{Binding " + b.Path.Path + @", Converter={x:Static s:SouthwindConverters.ImageConverter}, Mode=OneWay}""/>

</DataTemplate>";

return (DataTemplate)XamlReader.Parse(text);

});

* **Creating FrameworkElementFactory manually:** Deprecated and an unfriendly API, especially when there are nested elements.

QuerySettings.RegisterPropertyFormat((EmployeeDN e)=>e.Photo, b=>

{

b.Converter = SouthwindConverters.ImageConverter;

FrameworkElementFactory factory = new FrameworkElementFactory(typeof(Image));

factory.SetBinding(Image.SourceProperty, b);

factory.SetValue(Image.StretchProperty, Stretch.Uniform);

factory.SetValue(Image.MaxHeightProperty, 32.0);

factory.SetValue(RenderOptions.BitmapScalingModeProperty, BitmapScalingMode.Linear);

return **new** DataTemplate { VisualTree = factory };

});

* **Using Fluent.GetDataTemplate method:** It takes a lambda expression that creates a FrameworkElement and transforms it in a DataTemplate or a FrameworkElementFactory.

You need to use object and collection initializes, or the extension methods in the Fluent class for attached properties and bindings. Like this:

QuerySettings.RegisterPropertyFormat((EmployeeDN e)=>e.Photo, b=>

{

b.Converter = SouthwindConverters.ImageConverter;

return Fluent.GetDataTemplate(() =>

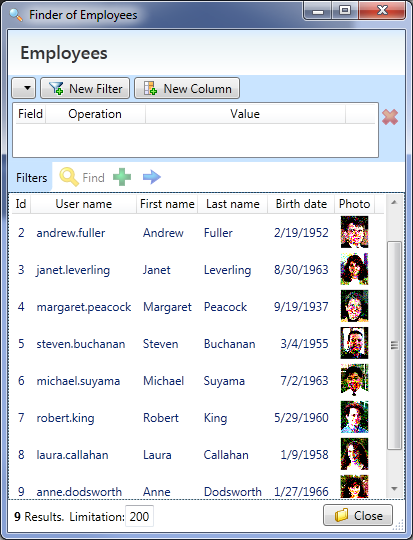
new Image { MaxHeight = 32.0, Stretch = Stretch.Uniform }

.Bind(Image.SourceProperty, b)

.Set(RenderOptions.BitmapScalingModeProperty, BitmapScalingMode.Linear));

});

After using any of this methods, and running the application, we could see the thumbnails of the photo of every Employee.



Of course, this could have performance problems in some scenarios since a whole Photo is being downloaded, and would be a better approach to save a thumbnail version of the photo in the server instead.

## Getting fast with Products

Not that we have seen some of the features of Signum.Windows we can get faster in making the user interfaces for Products.

### Suppliers

We will start with suppliers. We use Signum Entity Control wizard as usual and then we modify the results. It has an AddressDN so we can reuse the control that we made, also let´s use a UniformGrid to arrange ContactTitle, ContactName, Phone and Fax Number.

Just a simple but fully functional form for suppliers:

<StackPanel>

<m:ValueLine m:Common.Route="CompanyName" />

<sc:Address m:Common.Route="Address"/>

<UniformGrid>

<m:ValueLine m:Common.Route="ContactName" />

<m:ValueLine m:Common.Route="ContactTitle" />

<m:ValueLine m:Common.Route="Phone" />

<m:ValueLine m:Common.Route="Fax" />

</UniformGrid>

<m:ValueLine m:Common.Route="HomePage" />

</StackPanel>

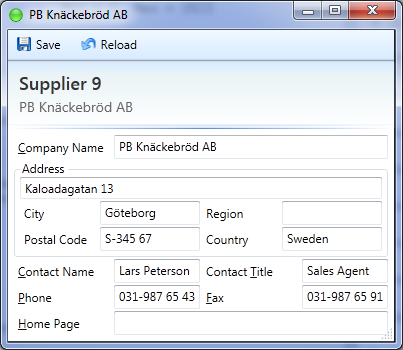
As usual, let’s add Supplier EntitySettings’ to the Navigator,

new EntitySettings<SupplierDN>(EntityType.Admin) { View = e => new Supplier() },

And a menu that opens a search dialog for Suppliers.

<MenuItem Tag="{m:Explore QueryName={x:Type d:SupplierDN}}"/>

OVERRIDE TOSTRING in SIGNUM WEB



### Category

Category is a very simple entity, with just a CategoryName, Description and an Image that we will allow to change by double-clicking, just as we did with the employee photo.

<StackPanel>

<m:ValueLine m:Common.Route="CategoryName" />

<m:ValueLine m:Common.Route="Description" />

<Border Grid.Column="1" BorderThickness="1" BorderBrush="Gray" Margin="2"  
HorizontalAlignment="Center" VerticalAlignment="Center">

<Image x:Name="image"   
Source="{Binding Picture, Converter={x:Static s:SouthwindConverters.ImageConverter}}" Stretch="None" MinHeight="100" MinWidth="100" MouseDown="image\_MouseDown" />

</Border>

</StackPanel>

We add the EntitySettings and entry in the Menu and, before running, we will have to turn back to Southwind.Load to creater an script that removes the Ole header on the picture, just as we did with employees, and run it.

public static void FixCategoryImages()

{

foreach (var category in Database.RetrieveAll<CategoryDN>())

{

if (category.Picture != null)

{

category.Picture = RemoveOlePrefix(category.Picture);

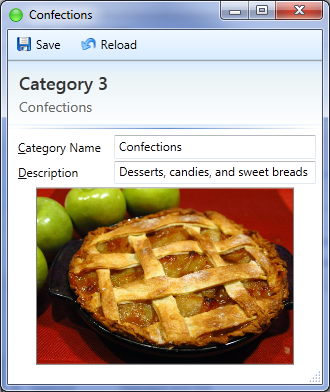
category.Save();

}

}

}

After this, we should be able to run the application and open some categories, and just by double clicking in the image, change it for one that looks from this century:

### Products

The product entity has no picture just data, in order to make it more fancy we will focus on the Stock Value.

Let’s modify the generated control by placing the classification properties (Category and ReorderLevel) up, supplier and discontinued down.

Also, in order to clarify the meaning of the properties UnitPrice, UnitsInStock and ValueInStock, we will make the interface represent the formula UnitPrice x UnitsInStock = ValueInStock, like this:

<Border BorderBrush="Gray" BorderThickness="1" HorizontalAlignment="Center" Padding="3">

<Grid m:Common.LabelVisible="False" >

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="Auto" MinWidth="100"/>

<ColumnDefinition Width="Auto" MinWidth="100"/>

</Grid.ColumnDefinitions>

<m:ValueLine m:Common.Route="UnitPrice" Grid.Column="1" />

<Label m:Common.LabelOnlyRoute="UnitPrice" Grid.Column="2" />

<TextBlock Text="X" Grid.Row="1" Margin="3" VerticalAlignment="Center" />

<m:ValueLine m:Common.Route="UnitsInStock" Grid.Row="1" Grid.Column="1" />

<Label m:Common.LabelOnlyRoute="UnitsInStock" Grid.Row="1" Grid.Column="2"/>

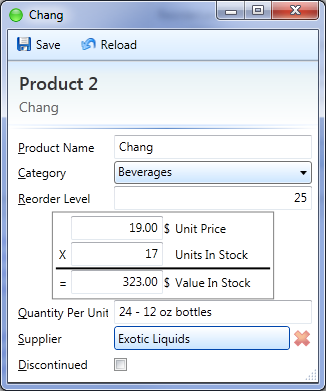
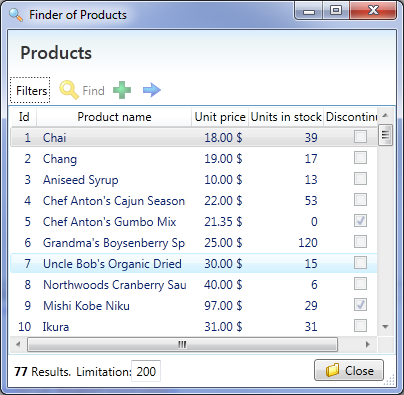
<TextBlock Text="=" Grid.Row="3" Margin="3" VerticalAlignment="Center" />

<m:ValueLine m:Common.Route="ValueInStock" Grid.Row="3" Grid.Column="1"/>

<Label m:Common.LabelOnlyRoute="ValueInStock" Grid.Row="3" Grid.Column="2"/>

<Border BorderBrush="Black" BorderThickness="1" Grid.Row="2" Grid.ColumnSpan="3" />

</Grid>

</Border>  
  
Also, let’s place a UnitAttribute over the UnitPrice property, in Product entity, so the ValueLines, as well as the search dialog, shows that the price is in dollars.   
  
 

There’s room for improvement though. If you change the values in Unit Price, or Units In Stock, Value In Stock doesn’t get updated.

In order to fix this we need to notify the change of ValueInStock, in the setters of UnitPrice :

public decimal UnitPrice

{

get { return unitPrice; }

set

{

if (Set(ref unitPrice, value, () => UnitPrice))

Notify(() => ValueInStock);

}

}

And do the same for UnitsInStock.

## Shippers

There’s nothing special about this simple entity, just create the control with the wizard and register it with EntitySettings and add the entry to the menu.

## Customers

Since Customer is an abstract class, we don’t need to implement and interface for it. Instead we will make interfaces for Person and Company.

Both entities are quite similar to the ones we have done before.

For companies will make sense to put the CompanyName in the top and an Address control, and use a UniformGrid for the remaining controls.

<StackPanel>

<m:ValueLine m:Common.Route="CompanyName" />

<sc:Address m:Common.Route="Address"/>

<UniformGrid>

<m:ValueLine m:Common.Route="ContactTitle" />

<m:ValueLine m:Common.Route="ContactName" />

<m:ValueLine m:Common.Route="Phone" />

<m:ValueLine m:Common.Route="Fax" />

</UniformGrid>

</StackPanel>

And Persons could be just the same, replacing the CompanyName for a control that shows the Title, FirstName and LastName just the way we did in employee.

Lets register both entities as we have done many times already, and add the related queries to a new sub menu called ‘Customers’:

<MenuItem Header="Customers">

<MenuItem Tag="{m:Explore QueryName={x:Type d:CustomerDN}}"/>

<MenuItem Tag="{m:Explore QueryName={x:Type d:PersonDN}}"/>

<MenuItem Tag="{m:Explore QueryName={x:Type d:CompanyDN}}"/>

</MenuItem>

Persons to find the persons, Company for the companies, and Customer to find persons and companies altogether as if they were in the same table, as we saw in the previous tutorial.

## Order

Even if we spent some time in customizing and polishing the previous controls, they will rarely going to be used, just from time to time for admin stuff.

Where the user is going to be spending time is in the Order form, so here is where we should focus our efforts.

### Default Template

Let’s see what the Signum Entity Control template does for our Order entity:

<StackPanel>

<m:EntityLine m:Common.Route="Customer" />

<m:EntityLine m:Common.Route="Employee" />

<m:ValueLine m:Common.Route="OrderDate" />

<m:ValueLine m:Common.Route="RequiredDate" />

<m:ValueLine m:Common.Route="ShippedDate" />

<m:ValueLine m:Common.Route="CancelationDate" />

<m:EntityLine m:Common.Route="ShipVia" />

<m:ValueLine m:Common.Route="ShipName" />

<GroupBox Header="ShipAddress" m:Common.Route="ShipAddress">

<StackPanel>

<m:ValueLine m:Common.Route="Address" />

<m:ValueLine m:Common.Route="City" />

<m:ValueLine m:Common.Route="Region" />

<m:ValueLine m:Common.Route="PostalCode" />

<m:ValueLine m:Common.Route="Country" />

</StackPanel>

</GroupBox>

<m:ValueLine m:Common.Route="Freight" />

<GroupBox Header="Details">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<m:EntityList m:Common.Route="Details" ViewOnCreate="False" Grid.Column="0" />

<m:DataBorder m:Common.Route="Details/" Grid.Column="1">

<StackPanel>

<m:EntityLine m:Common.Route="Product" />

<m:ValueLine m:Common.Route="UnitPrice" />

<m:ValueLine m:Common.Route="Quantity" />

<m:ValueLine m:Common.Route="Discount" />

<m:ValueLine m:Common.Route="SubTotalPrice" />

</StackPanel>

</m:DataBorder>

</Grid>

</GroupBox>

<m:ValueLine m:Common.Route="TotalPrice" />

<m:ValueLine m:Common.Route="IsLegacy" />

<m:ValueLine m:Common.Route="State" />

</StackPanel>

Everything looks familiar, except for the way Details has been handled. Since is an MList of OrderDetailDN (an EmbeddedEntity), notice how it has created a grid with two columns:

In one side there’s an EntityList, this control is the list equivalent of an EntityLine, it allows to create, navigate, remove and find entities in a list. Just as an EntityLine, it allows using EmbeddedEntities (Like OrderDetailDN).

On the other side, there’s a DataBorder with the OrderDetailsDN field. A DataBorder is a Border control with three interesting features:

* Clears the content when the DataContext is null, to represent that there’s no entity.
* Blinks the content visibility on DataContextChanged, to indicate the the user.
* If AutoChild is set to true, looks at the type of the entity in the DataContext and finds the appropriate control registered in the Navigator EntitySettings’.

Finally, notices how m:Common.Route in the DataBorder allows the ‘/’ notation to indicate the current selected item in the collection, propagating the TypeContext.

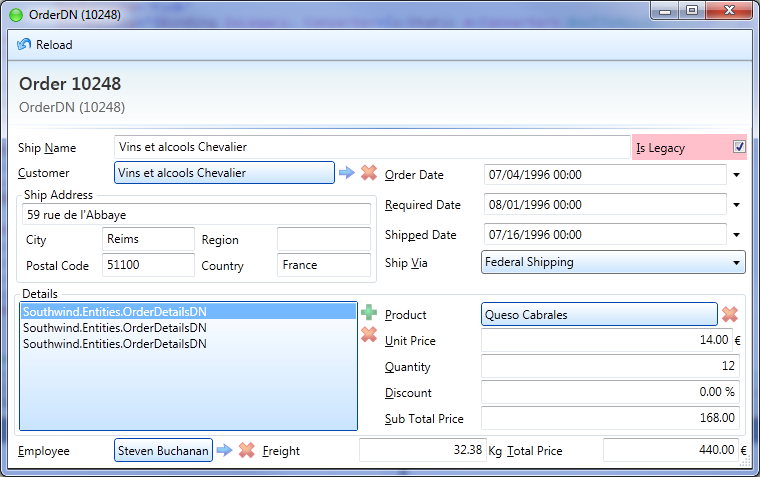
Even if this sophisticated default template for MList of EmbeddedEntities could be useful for other entities, for an Order entity we will need a proper Grid in there. Let’s arrange the rest of the fields first, and then let’s see what we can do for the Order Details.

### Arranging the fields

In order to make the form look like an Order, we will make the following changes:

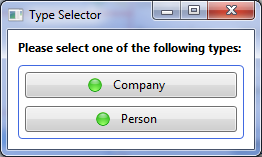
* Replace the ShipAddress generated contents for our Address control.
* Make the control MinWidth 600, instead of 300.
* Make a grid in the top, with customer and ShipAddress on one columnd and all the dates, and ShipVia, in the other.
* On the very top, we will place ShipName, with IsLegacy in the same line and docked to the Rights, visible only if it’s true and pink.
* On the bottom, we will make a 1 row uniform grid with the Employee, Freight and TotalPrice.

Just with these changes, the result looks more like an Order form:



Notice how Customer EntityLine, since is an ImplementedBy CompanyDN and PersonDN, has a polymorphic auto-complete that queries both tables.

Also, when the user press create, or find, a type picker is sown first.



Maybe this is a convenient default behavior for creating, but for find we could use out CustomerDN query, that mixes both entities. In order to do that let’s override Creating using the EntityLine event:

<m:EntityLine m:Common.Route="Customer" Finding ="EntityLine\_ Finding " />

private object EntityLine\_ Finding ()

{

return Navigator.Find(new FindOptions(typeof(CustomerDN)));

}

Finding event expects the Lite or Entity found by the user or null it the user canceled. Navigator.Find methods just provide an abstraction over opening a Search Dialog for a queryName, in this case CustomerDN. That simple.

Another usability improvement will be to set the address every time the user chooses (or creates) a customer. The easiest way is to attach to EntityChanged event in the Customer EntityLine.

<m:EntityLine m:Common.Route="Customer" Finding="EntityLine\_Finding" EntityChanged="EntityLine\_EntityChanged" />

A convenient userInteraction parameter indicates weather the change is triggered by the user or is a binding issue.

OrderDN OrderEntity

{

get { return (OrderDN)DataContext; }

}

private void EntityLine\_EntityChanged(object sender, bool userInteraction, object oldValue, object newValue)

{

if (userInteraction)

this.OrderEntity.ShipAddress = ((CustomerDN)newValue).TryCC(a => a.Address.Clone());

}

Finally, we will have to add a Clone instance method on the Address:

public AddressDN Clone()

{

return new AddressDN

{

Address = this.Address,

City = this.City,

Region = this.Region,

PostalCode = this.PostalCode,

Country = this.Country

};

}

## The Grid

Signum framework has no native support for grids, three’ no such thing as an EntityGrid, but the m:Common.Route patter is trained to understand the standart DataGrids, adding the following features:

* Set the header text (and unit)
* Bound the column (if inherits from DataGridBoundColumn)
* Set the TypeContext for the columns, usefull for DateGridTemplateColumns

With this information, let’s start with a simple DataGrid and some EntityButtons docked in the Right side.

<DockPanel Margin="10">

<m:EntityButtons x:Name="ebDetails" Orientation="Vertical" DockPanel.Dock="Right"

Create="False" Viewing="ebDetails\_Viewing"

Removing="ebDetails\_Removing" Finding="ebDetails\_Finding"/>

<DataGrid x:Name="dgDetails" m:Common.Route="Details" MinHeight="100"

AutoGenerateColumns="False"

SelectedCellsChanged="dgDetails\_SelectedCellsChanged">

<DataGrid.Columns>

<DataGridTextColumn m:Common.Route="Product"/>

<DataGridTextColumn m:Common.Route="UnitPrice" IsReadOnly="True"/>

<DataGridTextColumn m:Common.Route="Quantity"/>

<DataGridTextColumn m:Common.Route="Discount"/>

<DataGridTextColumn m:Common.Route="SubTotalPrice" IsReadOnly="True"/>

</DataGrid.Columns>

</DataGrid>

</DockPanel>

On the code behind,

OrderDetailsDN CurrentOrderDetails

{

get { return dgDetails.SelectedItem as OrderDetailsDN; }

}

private void ebDetails\_Finding(object sender, RoutedEventArgs e)

{

var product = Navigator.Find<ProductDN>();

if (product == null)

return;

OrderDetailsDN details = new OrderDetailsDN

{

Product = product,

Quantity = 1,

UnitPrice = product.Retrieve().UnitPrice,

Discount = 0,

};

OrderEntity.Details.Add(details);

}

private void ebDetails\_Removing(object sender, RoutedEventArgs e)

{

if (CurrentOrderDetails == null)

return;

OrderEntity.Details.Remove(CurrentOrderDetails);

}

private void dgDetails\_SelectedCellsChanged(object sender, SelectedCellsChangedEventArgs e)

{

ebDetails.View = ebDetails.Remove = CurrentOrderDetails != null;

}

private void ebDetails\_Viewing(object sender, RoutedEventArgs e)

{

if (CurrentOrderDetails == null)

return;

Navigator.Navigate(CurrentOrderDetails.Product);

}

With this code we can emulate an ‘EntityGrid’ with the difference than, when pressing Finding instead of looking for an OrderDetailDN, we look for a ProductDN and construct a OrderDetailDN from there, which makes more sense. Same for Viewing.

### AutoComplete

The product column, however, is not editable since there’s no way to convert an string to a product. What we will do instead is to change the column for a DataGridTemplateColumn with an AutoCompleteTextBox on edit mode:

<DataGridTemplateColumn m:Common.LabelOnlyRoute="Product" MinWidth="300" x:Name="colProduct">

<DataGridTemplateColumn.CellEditingTemplate>

<DataTemplate>

<m:AutoCompleteTextBox AutoCompleting="AutoCompleteTextBox\_AutoCompleting"

SelectedItem="{Binding Product, Mode=TwoWay}"

Closed="AutoCompleteTextBox\_Closed"

Loaded="AutoCompleteTextBox\_Loaded" />

</DataTemplate>

</DataGridTemplateColumn.CellEditingTemplate>

<DataGridTemplateColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding Product}"/>

</DataTemplate>

</DataGridTemplateColumn.CellTemplate>

</DataGridTemplateColumn>

We will use an AutoCompleteTextBox, instead of an EntityLine, since we don’t need the entity buttons and the blue pill. Then, on the code behind we have to define how the auto-complete will get the elements:

private IEnumerable AutoCompleteTextBox\_AutoCompleting(string arg)

{

return Server.Return((IBaseServer s) =>

s.FindLiteLike(typeof(ProductDN), null, arg, 5));

}

Note how we use FindLiteLike, the method that EntityLine use internally to provide auto-complete.

Finally we have to set the text, select all and focus when the AutoCompleteTextBox gets loaded:

private void AutoCompleteTextBox\_Loaded(object sender, RoutedEventArgs e)

{

var actb = (AutoCompleteTextBox)sender;

actb.Text = CurrentOrderDetails.Product.TryCC(a => a.ToStr);

actb.SelectAndFocus();

}

### Immediate Grid

When the user moves from one cell to the other the changes don’t get reflected in the sub-total price of the line and total price of the order. Let’s do that by making the changes in our OrderDetailsDN to notify affected properties after a change:

[Unit("€")]

public decimal UnitPrice

{

get { return unitPrice; }

set

{

if (Set(ref unitPrice, value, () => UnitPrice))

Notify(() => SubTotalPrice);

}

}

And the same changes on Quantity and Discount. In order to notify TotalPrice, the parent entity, we will have to override ChildCollectionChanged (for changes in the details collection) and ChildPropertyChanged (for changes in the elements itself) and add the corresponding attributes to the details feld.

[ValidateChildProperty, NotifyChildProperty, NotifyCollectionChanged]

MList<OrderDetailsDN> details;

protected override void ChildCollectionChanged(object sender, NotifyCollectionChangedEventArgs args)

{

if (sender == details)

Notify(() => TotalPrice);

}

protected override void ChildPropertyChanged(object sender, PropertyChangedEventArgs e)

{

if (sender is OrderDetailsDN)

Notify(() => TotalPrice);

}

Also, in order to have rollback behavior by pressing esc before committing the row, we have to implement IEditableObject in our OrderDetailsDN entity:

public class OrderDetailsDN : EmbeddedEntity, IEditableObject

{

(…)

[Ignore]

OrderDetailsDN clone;

public void BeginEdit()

{

clone = new OrderDetailsDN

{

Product = product,

Quantity = quantity,

UnitPrice = unitPrice,

Discount = discount

};

}

public void CancelEdit()

{

Product = clone.product;

Quantity = clone.quantity;

Discount = clone.discount;

}

public void EndEdit()

{

clone = null;

}

}

After we make all this changes, if we run and see the results… still the sub total doesn’t get updated after editing each cell!. The reasons is because DataGrids have the ‘feature’ of keeping the bindings from setting changes until the user commits the row (using BindingGroup). We have to find a way to disable this:

<http://stackoverflow.com/questions/5028556/wpf-datagrid-source-updating-on-cell-changed>

Let’s create a new ImmediateGrid class , inheriting from DataGrid, that does this behavior:

public class ImmediateGrid : DataGrid

{

protected override Size MeasureOverride(Size availableSize)

{

var desiredSize = base.MeasureOverride(availableSize);

ClearBindingGroup();

return desiredSize;

}

private void ClearBindingGroup()

{

// Clear ItemBindingGroup so it isn't applied to new rows

ItemBindingGroup = null;

// Clear BindingGroup on already created rows

foreach (var item in Items)

{

var row = ItemContainerGenerator.ContainerFromItem(item) as FrameworkElement;

if (row != null)

row.BindingGroup = null;

}

}

}

Now let’s replace our DataGrid for a ImmediateGrid (taking care of the xml namespaces) and, finally, the behavior should be as expected.

### Style

The last improvements, let’s add the following style to the DataGrid so we readonly colums are not focusable and dark gray.

<Style TargetType="{x:Type DataGridCell}">

<Style.Triggers>

<Trigger Property="IsReadOnly" Value="True">

<Setter Property="Foreground" Value="#606060"/>

<Setter Property="Focusable" Value="False"/>

</Trigger>

</Style.Triggers>

</Style>

And, in App.xaml, let’s merge the Resources with the one in the Framework (Resources.xaml) to ensure that all the styles are applied correctly, and BlueMenu, to change the color of the menu to a light blue one, that fit’s better with the rest of the Signum.Windows style.

<ResourceDictionary.MergedDictionaries>

<ResourceDictionary Source="pack://application:,,,/Signum.Windows;Component/Resources.xaml"/>

<ResourceDictionary Source="pack://application:,,,/Signum.Windows;Component/BlueMenu.xaml"/>

</ResourceDictionary.MergedDictionaries>

## Conclusion

Even if this tutorial has get a little longer that I wanted, we have seen how fast could it be to make a complete windows user interface for our application, thanks to the clever default behavior of the entity controls, and a little bit of help of code generation.

Having entities that already implement INotifyPropertyChanged for notifying changes in the properties, IDataErrorInfo for validation, and that are easly serializable and sent to the server is the best possible scenario when working with WPF bindings.

On the other side, since Signum.Windows is just a set of WPF controls, using the framework doesn’t impose any kind of restriction, letting you express your creativity when necessary with zero extra is cost over a ‘pure’ XAML user interface, as we have seen when we integrated the picture on the Employee or the DataGrid on the Order entity control.

After this tutorial, we have a full understanding of all the big pieces of an application built with Signum framework: Entities, Logic, Load, and Web and Windows user interfaces.

In the next tutorials we will focus on cross-cutting problems, like localization, authorization, time zones, etc..

I hope the tutorial has been useful for you.