# Signum Framework Tutorials Part 6 – Southwind Web

## 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 web application using Asp.Net MVC and Signum.Web.

### Introduction

Signum.Web is the library with the controls and necessary infrastructure to create Asp.Net MVC applications with the Signum Framework, and has been added in Signum Framework 2.0.

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.Web, just as in Signum.Windows, are the edition controls (used as HtmlHelpers) and the search controls. There’s also a similar Navigator 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.Web, as opposed to Southwind.Windows, runs in the server and has access to the database and the logic.

## Making Contact

The first thing to do is to load Southwind.Web project.

There we can see a few folders that follow the Asp.Net MVC convention:

* Content: Contains static files such as CSS files and images
* Controllers: Contains Asp.Net MVC controller classes
* Models: Contains Asp.Net MVC model classes. By default the entities will be our model classes, but sometimes our views we will need to combine (or reduce) the information given by the entities, so we will create models.
* Scripts: Contains Javascript files. Signum.Web relies on jquery and jquery-ui, so at least those two libraries must be present in this folder.
* Views: Contains our Asp.Net MVC views. It has three subfolders, which also follow Asp.Net conventions.
  + Home will contain the entry point views for our application.
  + Shared will contain shared views for different areas of the application (the Layout page, the application menu, and the error page)
  + Southwind will contain the views of our entities.
* Code: This is the only non-standard Asp.Net MVC folder, and we use it to group code files used in our application.

We can also see some files:

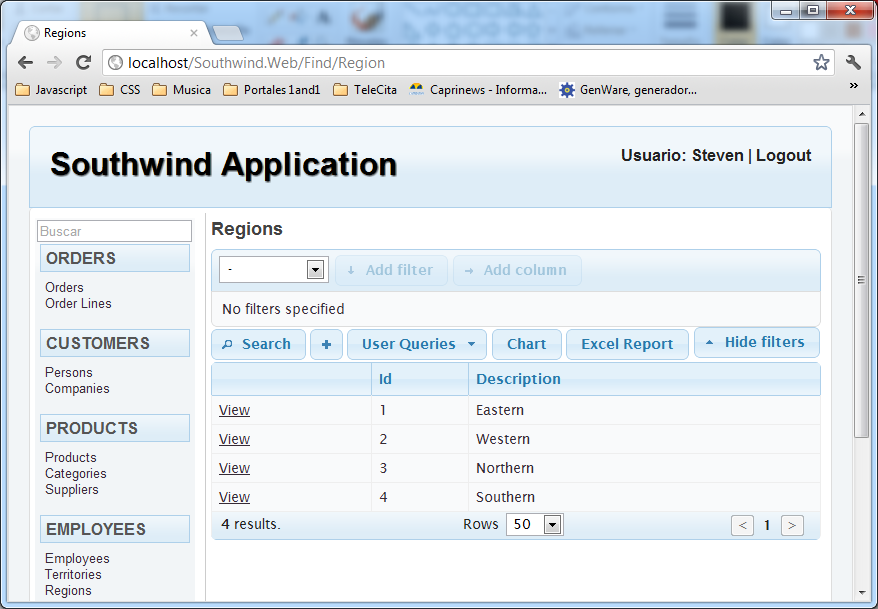
* Web.Config: Contains the configuration of the web application.
* Global.asax: Contains the initialization code of the web application. Registers routes, calls the application logic initialization, links the entities with their default views, etc.
* ServerSouthwind.svc: This is the IIS-Hosted WCF Service used as server interface for Signum.Windows applications (see Signum Framework Tutorials Part 5 – Southwind Windows). The contract of this service (IServerSouthwind) is defined in Southwind.Entities so both client and server have access to it.

## Regions and Territories

In the menu (Views/Shared/Menu.cshtml) 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 Code/SouthwindClient.cs we have a line trying to registrer MyEntityDN EntitySettings, let’s comment it out.

If we try to run now we will see a page with menu at the left, there we have a Regions entry that will open a page like this:



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

Note the route on the address bar. After the name of the application we have “/Find/*EntityName*”. That’s the default route of the search pages in Signum.Web, although it can be changed to suit our needs.

As in Signum.Windows, 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 clicking on the view link in the first column (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:

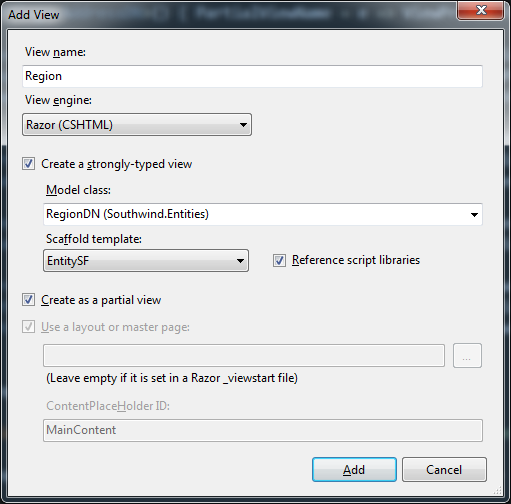
In the Views/Southwind folder we have MyEntity.cshtml with an example. Let’s get rid of it and generate a new one for Regions.

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.Web, 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 Views/Southwind folder
2. Add -> View
3. Write the name of the control (the same as the entity without the suffix by convention)
4. Check “Create a strongly-typed view” and select RegionDN from the model combo.
5. Set EntitySF as Scaffold template
6. Check “Create as partial view”
7. Click Add.

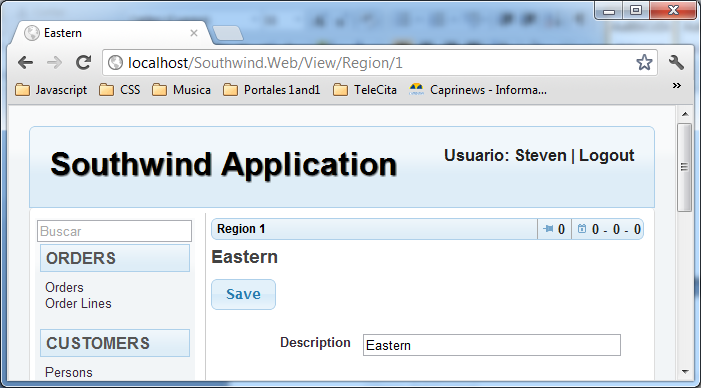


Finally, let’s turn back to the initialization code of our application, refactored to Code/SouthwindClient.cs, and let’s create an EntitySettings line, similar to the one we just commented out:

new EntitySettings<RegionDN>(EntityType.Admin) { PartialViewName = e => ViewPrefix.Formato(“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 Search Page mainly).

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



Also note the route “/View/*EntityName/Id*” which is the default route for viewing entities, although it can also be changed if necessary.

Let’s take a look at the generated code:

@using Southwind.Entities

@using (var rc = Html.TypeContext<RegionDN>())

{

@Html.ValueLine(rc, r => r.Description)

}

* The snippet created a normal Razor view
* It included your entities namespace. Signum Framework namespaces don’t need to be added in every view as they are referenced in the web.config.
* The first line after the imports extracts a TypeContext of type RegionDN from the Model property.
* Then it renders the properties. In this case it adds a ValueLine pointing to Description string property.

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

* All the surrounding controls and widgets (such as the title bar or the notes and alert widgets). They are not defined in the Layout either. They are defined in the shared NormalPage, in the Signum.Web, so you only have to focus on the partial view for your entity. However you could override NormalPage if you don’t like the default one.
* The Save button is already fully functional, but you can hide or override it if you want using EntitySettings. You can also register new buttons to be added to the button bar.
* 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 which control ValueLine uses for each data type if you want.
* By using a TypeContext a set of tasks has been executed, this pipeline is responsible of:
  + Setting the property value to the control, automatically taking care of:
    - Enable Asp.Net MVC ModelState validation and error notification.
    - 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, also sets the Implementations property.

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.

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

new EntitySettings<TerritoryDN>(EntityType.Admin) { PartialViewName = e => ViewPrefix.Formato(“Territory”) }

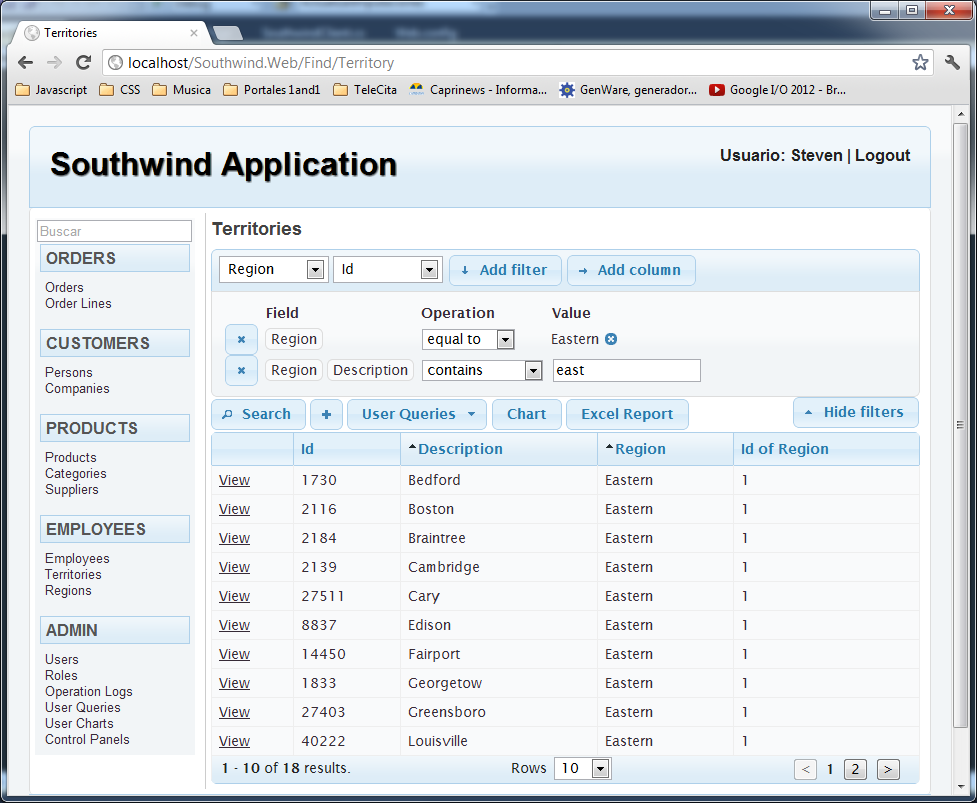
Finally let’s add another item in the menu:

new WebMenuItem { Link = new FindOptions(typeof(TerritoryDN)) },

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

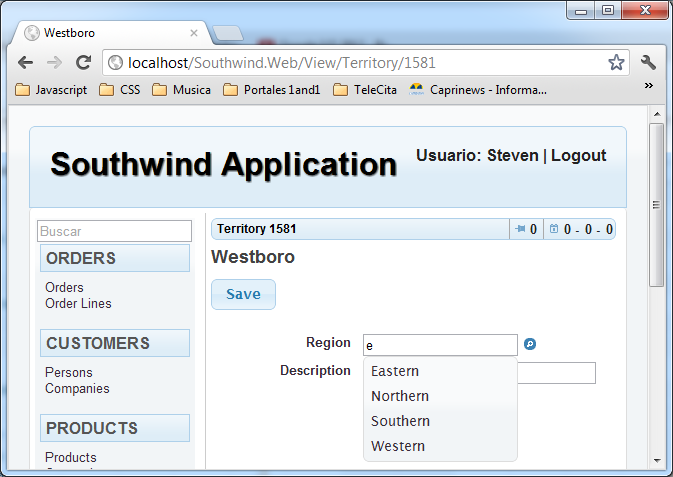
Let’s play a little bit with it now:

* If you click on every column header, you can order results as expected. Shift-click on a column header to add multiple orders.
* If you right click on a cell and select Add Filter, a new filter with that value preselected will appear. You are free to change the operation (equal to, distinct to, greater than, contains, etc… depending of the type) or the value. If you add the filter from a column header, the filter will appear, just without value, which is also sometimes useful.
* You can remove the filter by clicking the delete button at the left of the filter row.
* Note how depending of the field type of the filter the right control is set to select the value. Also, since the Region is a Lite<TerritoryDN>, not just a string, you can set the value using the autocomplete functionality or opening a modal search window.
* Finally, in the top buttons, select Region in the first combo, and notice how a second combo appears to dig into the Region properties, there select Description and click Add 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 Add Column. Click Search and see how now a new “Id of Region” column shows us the values.



If we click on the entity link of a result, we will open the newly created control. Apart from another ValueLine for edit description, we can now see the EntityLine.

Also, if you remove the region of the territory, notice how the EntityLine is smart enough to look at the type of the property (RegionDN) and offers you the possibility to search another region, by opening a modal search window, or using the autocomplete.



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

The controls usually have enough extension points to change the behavior when necessary 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) { PartialViewName = e => ViewPrefix.Formato(“Employee”)},

Then, let’s add it to the menu:

new WebMenuItem { Link = new FindOptions(typeof(EmployeeDN)) },

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

@using Southwind.Entities

@using (var ec = Html.TypeContext<EmployeeDN>())

{

@Html.ValueLine(ec, e => e.LastName)

@Html.ValueLine(ec, e => e.FirstName)

@Html.ValueLine(ec, e => e.Title)

@Html.ValueLine(ec, e => e.TitleOfCourtesy)

@Html.ValueLine(ec, e => e.BirthDate)

@Html.ValueLine(ec, e => e.HireDate)

using(var ac = ec.SubContext(e => e.Address))

{

@Html.ValueLine(ac, a => a.Address)

@Html.ValueLine(ac, a => a.City)

@Html.ValueLine(ac, a => a.Region)

@Html.ValueLine(ac, a => a.PostalCode)

@Html.ValueLine(ac, a => a.Country)

}

@Html.ValueLine(ec, e => e.HomePhone)

@Html.ValueLine(ec, e => e.Extension)

@Html.ValueLine(ec, e => e.Photo)

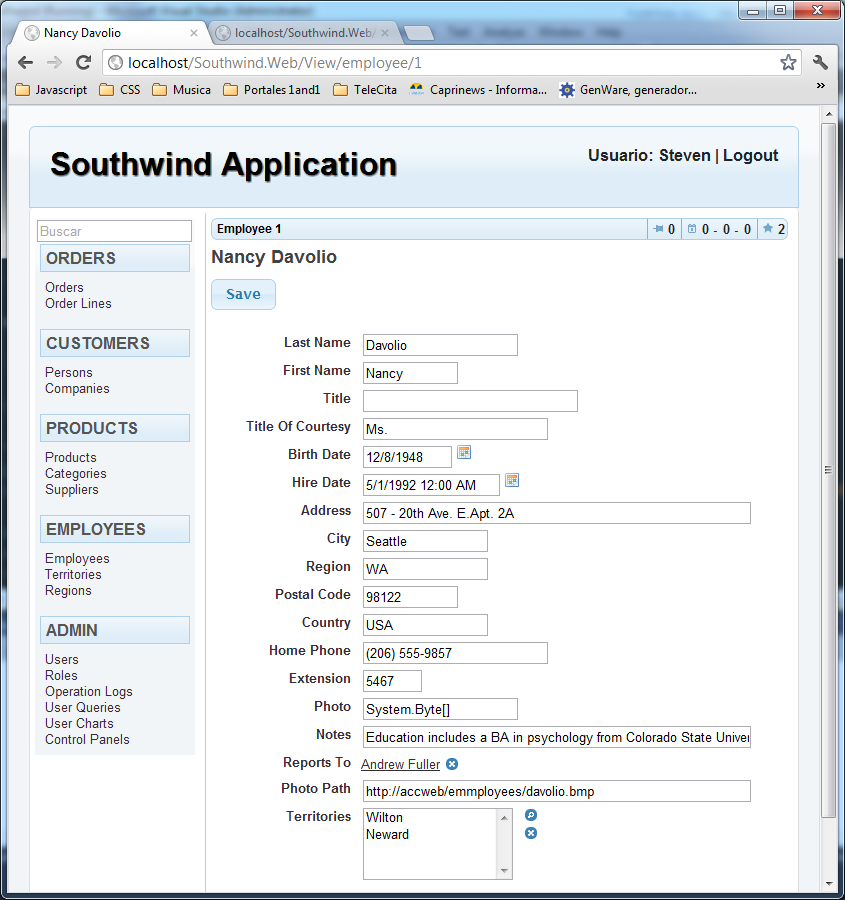
@Html.ValueLine(ec, e => e.Notes)

@Html.EntityLine(ec, e => e.ReportsTo)

@Html.ValueLine(ec, e => e.PhotoPath)

@Html.EntityList(ec, e => e.Territories)

}



## Improving Employee view

Even if the form already works, 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 the fields in a sensible way, such as Last Name, First Name and Title of Courtesy
  + Make a reusable Address control
* Show the picture!
* Make the notes a text area and change its colour.

### 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).

We will accomplish this task by embedding them inside a FieldInline region, which is available as a HtmlHelper. String valuelines will automatically set the length of the input controls to the maximum size of the property. In this case we will also limit the size of the Title of Courtesy, as well as override its label:

<fieldset>

<legend>Name</legend>

@using (Html.FieldInline())

{

@Html.ValueLine(ec, e => e.TitleOfCourtesy, vl =>

{

vl.ValueHtmlProps["size"] = 4;

vl.LabelText = "Title";

})

@Html.ValueLine(ec, e => e.FirstName)

@Html.ValueLine(ec, e => e.LastName)

}

</fieldset>



If we prefer the labels to be below the input controls, we can use the Valueline property ValueFirst, which basically places the value control before the label:

<fieldset>

<legend>Name</legend>

@using (Html.FieldInline())

{

@Html.ValueLine(ec, e => e.TitleOfCourtesy, vl =>

{

vl.ValueFirst = true;

vl.ValueHtmlProps["size"] = 4;

vl.LabelText = "Title";

})

@Html.ValueLine(ec, e => e.FirstName, vl => vl.ValueFirst = true)

@Html.ValueLine(ec, e => e.LastName, vl => vl.ValueFirst = true)

}

</fieldset>



ValueLines offer also other properties related to the label, for example LabelVisible to hide them, or LabelClass to change their appearance via CSS.

### 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 fieldset with the name of the property will group the fields. Like this:

@using Southwind.Entities

@using (var ac = Html.TypeContext<AddressDN>())

{

@Html.HiddenRuntimeInfo(ac)

<fieldset>

<legend>@(ViewData["Legend"] ?? typeof(AddressDN).NiceName())</legend>

@Html.ValueLine(ac, a => a.Address, vl =>

{

vl.LabelVisible = false;

vl.ValueHtmlProps["placeholder"] = Html.PropertyNiceName(() => ac.Value.Address);

})

@{ Action<ValueLine> modifier = vl => { vl.ValueHtmlProps["size"] = "20; }

@using (Html.FieldInline())

{

@Html.ValueLine(ac, a => a.City, modifier)

@Html.ValueLine(ac, a => a.Region, modifier)

}

@using (Html.FieldInline())

{

@Html.ValueLine(ac, a => a.PostalCode, modifier)

@Html.ValueLine(ac, a => a.Country, modifier)

}

</fieldset>

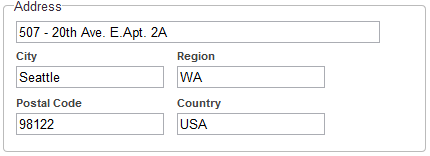
}

Note how first of all there is a helper that renders a hidden field with the RuntimeInfo information of the TypeContext. This is what will be used in the server when parsing the form data, to know if the entity being parsed is new or it’s an existing one. We expand a little bit into this in the following box.

Next we notice that the legend of the fieldset is taken from the ViewData if provided, or if not it uses the type nice name. This way we can use the same control but set different titles, such as Billing Address.

In the Address property, we hide the label and use a placeholder instead that will be written over the input when there’s no text until it receives the focus.

For the other fields, we place them inline in pairs as we already have seen, and also set the same size for all of them.



In the employee control, to embed the address control we just substitute all the ValueLines of the address properties with this single line:

@Html.EmbeddedControl(ec, e => e.Address)

By using always the TypeContext as source we will be transmitting all the needed information to the embedded controls.

### RuntimeInfo field

In Asp.Net MVC there is no concept of current state being saved anywhere, unlike traditional Asp.Net Web Forms, where ViewState is used to store state and several events are fired in the server to restore it in every postback.

This means that our form needs a way to provide enough information to build our model from scratch when we make a request to the server.

In Signum Framework we didn’t like the idea of using Session by default for this purpose, as it has several problems, so we accomplish this task by adding a hidden field for each rendered entity. This hidden field is the RuntimeInfo.

The RuntimeInfo format is the following: “*TypeName*;*Id*;*NewFlag*”:

* The first part contains the type name if the entity has value or blank otherwise.
* The second part contains the Id of the entity if it already exists in the database or blank otherwise.
* The *NewFlag* part is needed for EmbeddedEntities, because they don’t have Id even if they exist in the Database, so we need a way of knowing if they are new (value ‘n’), or old (value ‘o’)

Using these hidden fields in a request, the Signum Framework is able to construct an entity with the correct structure and apply the changes appropriately to the database. We will see this later on when we see the Mapping, which is the Signum.Web part that covers both the ValueProvider and ModelBinding of Asp.Net MVC.

EntityLines and other Line controls (EntityList, EntityCombo, etc.) automatically render these fields for us, but when we create controls to be embedded directly in other controls or pages, we need to take care of this field ourselves.

Another important topic is the naming convention for the html fields:

### Prefixes and field naming convention

Now we know how to provide information to the server about the current entities in the form using the RuntimeInfo. But there can be several entities (embedded or not) referenced by our main entity, so there will be a RuntimeInfo for each of them and we need a naming convention so there are never conflicts.

Following Asp.Net MVC default convention, we start with an empty prefix and each time we traverse a property we add its name. Property names are separated with “\_”.

Signum Framework sets both Id and Name html properties so as to be used by Javascript and server respectively. Let’s see some examples of Ids in the Employee form:

* The FirstName input field will just have “FirstName” as Id
* The City input field will have “Address\_City” as Id
* The Employee hidden runtime info field will have “sfRuntimeInfo” as Id
* The Address hidden runtime info field will have “Address\_sfRuntimeInfo” as Id

Everytime a modal popup is shown al the fields in it will inherit a common prefix, so as to not make conflict with the rest of the controls of the page:

* If the popup has been opened by a Line control it will also use the name of the property. For example if we open a search window from the ReportsTo EntityLine, the search button will have the following id: “ReportsTo\_qbSearch”
* If the popup has been opened by a custom button or an operation (see Operation module), by default it will have the “New\_” prefix.

A common case is having two or more SearchControls embedded in a page, maybe in different tabs. They must be given different prefixes when declaring them, so as the actions made to one of them will not interfere with the other ones.

TypeContext and Line controls will give us their current prefix with the ControlID property.

### Showing the picture

We have the photo as a byte[]. We need to convert this byte[] to a valid source for the img tag. For this we could create an action that reads this byte[] and returns it as an MVC FileResult:

In the employee form we have:

@if (!ec.Value.IsNew)

{

<div class="photo">

<img src="@(Url.Action<HomeController>(hc => hc.EmployeePhoto(ec.Value.ToLite())))" />

</div>

}

Note how we call the controller in a strongly type way using Signum.Web url generic extensions.

The controller will be just:

public FileResult EmployeePhoto(Lite<EmployeeDN> employee)

{

return File(employee.InDB().Select(e => e.Photo).Single(), "image/jpeg");

}

### Notes

Out last improvement will be to change the Note ValueLine to be a textarea instead of a text input. Each type has a default control in ValueLine, for example:

* DateTimes are represented as text inputs with a jquery-ui DatePicker beside them
* Enums are represented as dropdown lists
* Numeric fields are represented as text inputs that accept only numeric keypresses
* Boolean fields are represented as check inputs.

But we can change the default control changing the ValueLineType like this:

<fieldset>

<legend>@Html.PropertyNiceName(() => ec.Value.Notes)</legend>

@Html.ValueLine(ec, e => e.Notes, vl =>

{

vl.ValueLineType = ValueLineType.TextArea;

vl.LabelVisible = false;

vl.ValueHtmlProps["cols"] = "70";

vl.ValueHtmlProps["rows"] = "4";

vl.ValueHtmlProps["class"] = "notes";

})

</fieldset>

Also note how we can add properties to ValueHtmlProps. They will be added as properties to the value html control, in this case the textarea.

We add a CSS class just as any other attribute. In this case we have set the “notes” class to our textarea, which will correspond to the following rule:

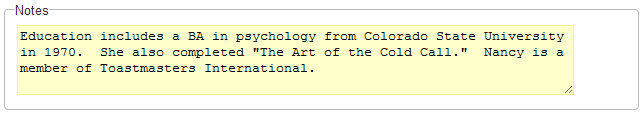
.notes

{

background-color: #FFC;

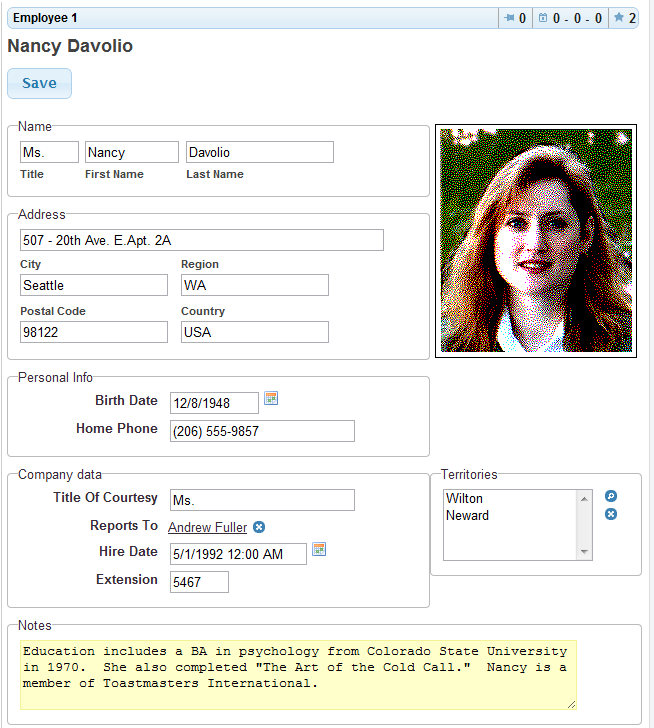
border-color: #F3F395;

}



However, for the notes, we could always use the Notes widget on the top right corner, which allows us to create notes, as well as view the current ones. If we don’t like it, it can be disabled for specific entities or for all our application, as all the other widgets.

So this is basically it, grouping the rest of the fields, we could have the following Employee form, which is a bit more user friendly:



## SearchControl Column Formatters

The last improvement we could make is not in the Employee control itself, but in the way the SearchControl shows Employees images. It doesn’t make sense to show ‘System.Byte[]’, instead it 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 html template 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 html templates 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>> propertyRoute,

Func<HtmlHelper, object, MvcHtmlString> formatter) where T : IRootEntity

The fist argument allows you to specify the property with a strongly typed style, and in the second one you have to provide a MvcHtmlString as template for the column, given an HtmlHelper and the value itself.

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

QuerySettings.RegisterPropertyFormat(

(EmployeeDN e) => e.Photo,

(html, obj) => obj == null ? null :

new MvcHtmlString("<img src=\"data:image/jpg;base64," + Base64Thumbnail((byte[])obj) + "\" />"));

The formatter calls the following helper method, which resizes the image and converts it to JPG, which you can see in the final Southwind code if you are interested:

public static string Base64Thumbnail(byte[] image)

{

using(MemoryStream ms = new MemoryStream(image))

using(Bitmap bmp = new Bitmap(ms))

using(Bitmap target = Resize(bmp, new Size(48,48)))

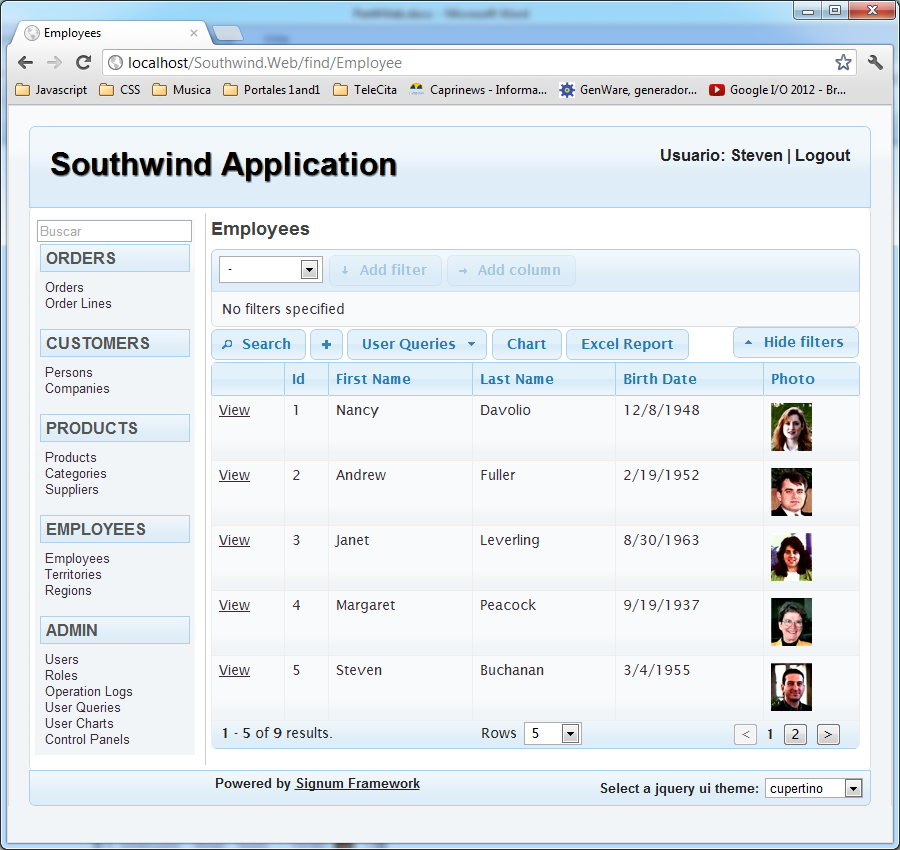
{

return Convert.ToBase64String(target.SaveJPG100());

}

}

After registering the property formatter, 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

Now that we have seen some of the features of Signum.Web we can get faster in making the user interfaces for Products.

### Suppliers

We will start with suppliers. We use EntitySF wizard as usual and then we modify the results. It has an AddressDN so we can reuse the control that we made.

Just a simple but fully functional form for suppliers:

@using Southwind.Entities

@using (var sc = Html.TypeContext<SupplierDN>())

{

@Html.ValueLine(sc, s => s.CompanyName)

@Html.EmbeddedControl(sc, s => s.Address)

@Html.ValueLine(sc, s => s.ContactName)

@Html.ValueLine(sc, s => s.ContactTitle)

@Html.ValueLine(sc, s => s.Phone)

@Html.ValueLine(sc, s => s.Fax)

@Html.ValueLine(sc, s => s.HomePage)

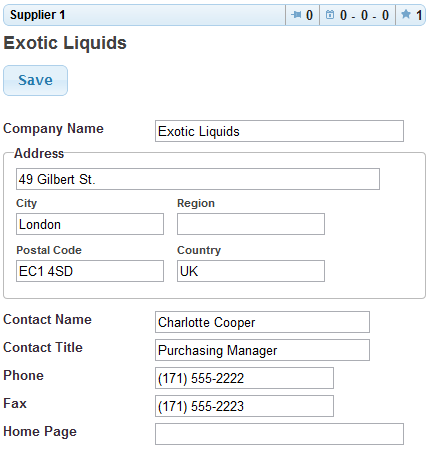
}

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

new EntitySettings< SupplierDN >(EntityType.Admin) { PartialViewName = e => ViewPrefix.Formato(“Supplier”)},

Then, let’s add it to the menu:

new WebMenuItem { Link = new FindOptions(typeof(SupplierDN)) },

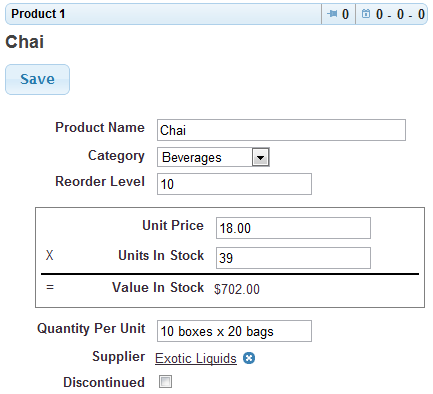


To achieve a bit more uniformity in the form I overloaded all the line labels in this page to be left-aligned. The CSS class of all line labels is *sf-label-line.*

## Products

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:



To accomplish this we just need to place the ValueLines inside a table or a few divs as we prefer, and some basic CSS rules. Let’s add them to a table, which might be more convenient to focus just on the Signum Framework pieces:

<div id="productExternal">

<div id="productInternal">

<table>

<tr>

<td></td>

<td>@Html.ValueLine(pc, p => p.UnitPrice,

vl => vl.ValueHtmlProps["onblur"] = "updateStockValue('" + pc.ControlID + "');")

</td>

</tr>

<tr id="trMultiply">

<td>X</td>

<td>@Html.ValueLine(pc, p => p.UnitsInStock,

vl => vl.ValueHtmlProps["onblur"] = "updateStockValue('" + pc.ControlID + "');")

</td>

</tr>

<tr>

<td>=</td>

<td>@Html.Field(Html.PropertyNiceName(() => pc.Value.ValueInStock),

Html.Span("stockValue", pc.Value.ValueInStock.ToString("c"), "sf-value-line"))</td>

</tr>

</table>

</div>

</div>

<script type="text/javascript">

function updateStockValue(prefix) {

var sum = parseFloat($("#" + SF.compose(prefix, "UnitPrice")).val()) \*

parseFloat($("#" + SF.compose(prefix, "UnitsInStock")).val());

$("#" + SF.compose(prefix, "StockValue")).html(sum);

}

</script>

Note there’s a small script that updates the sum value if you change the values in Unit Price, or Units In Stock.

Note how we can attach events also via the ValueHtmlProps, although it could be a better option to add a CSS class to them and attach the event afterwards with jquery.

However the important part of this update script is how it handles the naming convention we saw previously. Remember that we said that if a control is shown in a popup, it will have a prefix in all the field names.

So as to make our script work in a popup or as the main entity of the page, we use the TypeContext *ControlID* property, which gives us the current prefix of the Product entity.

This way in Javascript, we can use the *SF.compose* function, which will concatenate the prefix if exists with the property names using the “\_” separator.

It’s also very interesting to note the Html.Field helper. Sometimes we want to add manual information to a page, but we want it to look exactly as the ValueLines. We can use this helper, which receives a string as a label, and a custom MvcHtmlString as value, which can be anything we need. It will automatically adds the first parameter as a label with the correct *sf-label-line* CSS class, but it cannot do it to the value, as it could be anything. So if we would like the value to inherit the Line styles, we just have to add it the *sf-value-line* CSS class.

## 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 it will make sense to put the CompanyName in the top and an Address control.

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 it 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’:

new WebMenuItem

{

Text = "Customers",

Children =

{

new WebMenuItem { Link = new FindOptions(typeof(CustomerDN)) },

new WebMenuItem { Link = new FindOptions(typeof(PersonDN)) },

new WebMenuItem { Link = new FindOptions(typeof(CompanyDN)) },

}

},

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

Here we have the Details property, which is an MList<OrderDetailsDN>, which in turn is an EmbeddedEntity.

For MLists, we have three possible Line Controls. In all of them you can create new elements, remove current ones or add existing elements to the collection using a SearchControl. The differences are in the display:

* EntityList: Shows each entry as an option in a select multiple html element. When opening them, it will be in a popup or a new tab, depending on the EntityType.
* EntityDetailList: Shows the different entries as the EntityList, but displays the selected one in a master-detail manner just below or in a specified div in the page.
* EntityRepeater: Renders each entry as a control, each of them with a remove button.

In this example we will use the EntityRepeater, as it allows us to present some information without any click or navigation needed.

This is the full markup. I have highlighted the most important parts:

@using (var oc = Html.TypeContext<OrderDN>())

{

**<div class="sf-tabs">**

<fieldset id="orderMain">

<legend>Order</legend>

<table>

<tr>

<td>

@Html.ValueLine(oc, o => o.ShipName)

@Html.EntityLine(oc, o => o.Customer)

</td>

<td>

@Html.ValueLine(oc, o => o.TotalPrice)

@Html.EntityLine(oc, o => o.Employee)

</td>

</tr>

<tr>

<td style="padding: 10px 0 0 10px">

**@Html.EmbeddedControl(oc, o => o.ShipAddress, ec => ec.ViewData = new Dictionary<string, object> { { "Legend", Html.PropertyNiceName(() => oc.Value.ShipAddress) } })**

</td>

<td style="vertical-align: top; padding: 10px 0 0 10px">

@Html.ValueLine(oc, o => o.OrderDate)

@Html.ValueLine(oc, o => o.RequiredDate)

@Html.ValueLine(oc, o => o.ShippedDate)

@Html.EntityCombo(oc, o => o.ShipVia)

@Html.ValueLine(oc, o => o.Freight)

@Html.ValueLine(oc, o => o.IsLegacy)

</td>

</tr>

</table>

</fieldset>

<fieldset id="orderDetails">

<legend>Details</legend>

**@Html.EntityRepeater(oc, o => o.Details)**

</fieldset>

</div>

}

The external div with class *sf-tabs* is the way to quickly create tabs. There’s some code that runs each time content is added to the body that triggers several actions. One of them is the creation of jquery-ui tabs. It looks in the new HTML for a div with the class *sf-tabs*, and for each direct child fieldset, it will create a new tab, using its legend as tab title. More on this topic in the following box.

The other highlighted line is the call of the Address EmbeddedControl, as it shows how to pass ViewData information when rendering them.

### sf-new-content event

Every time a page is loaded, this event gets fired, giving us the opportunity to modify the new html or construct some jquery-ui elements following certain conventions.

We can do whatever actions suit our needs. Just go to the Scripts folder and open newcontent.js. There you’ll see how it binds to the event and performs several actions, more precisely:

* Creates jquery-ui buttons to all elements marked with the *sf-button* CSS class.
* Creates jquery-ui datepickers to all elements marked with the *sf-datepicker* CSS class
* Creates jquery-ui style dropdown menus to all elements with *sf-dropdown* CSS class setting as items its children with *sf-menu-button* CSS class
* Creates the EntityLines autocomplete functionality
* Creates the placeholder functionality for text inputs and textarea
* Creates jquery-ui tabs from all div elements marked with the sf-tabs CSS class, creating a tab from each direct child fieldset, as we have seen in the previous example.

We can modify this script add other actions or remove any of the default ones that doesn’t suit our needs. We could also override any of them, to change the way buttons are created for example.

At the end of the file we see that it triggers the event. As the Layout includes this script, this is the way it’s always triggered on load.

Framework ajax calls, such as opening popups, or operations also fire this event on the success event of the calls, but if we write code that creates adds new HTML to the page, and we would like to run this HTML pre-filters we must take care ourselves to fire it. For example:

$.post('ajax/test.html', function(newHtml) {

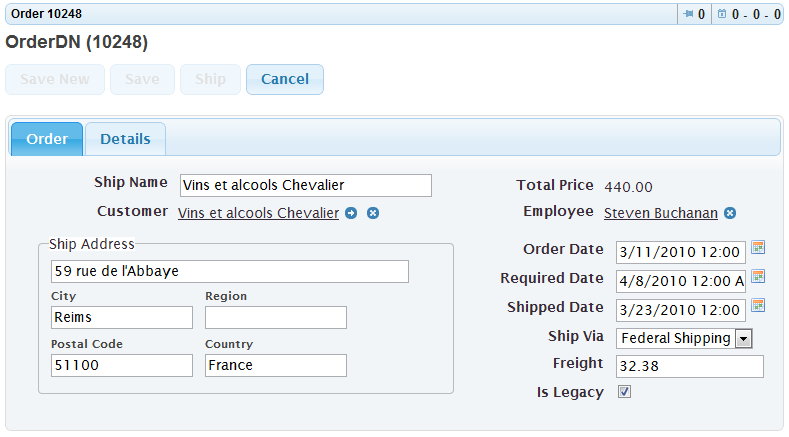
$("#myDiv").html(newHtml);

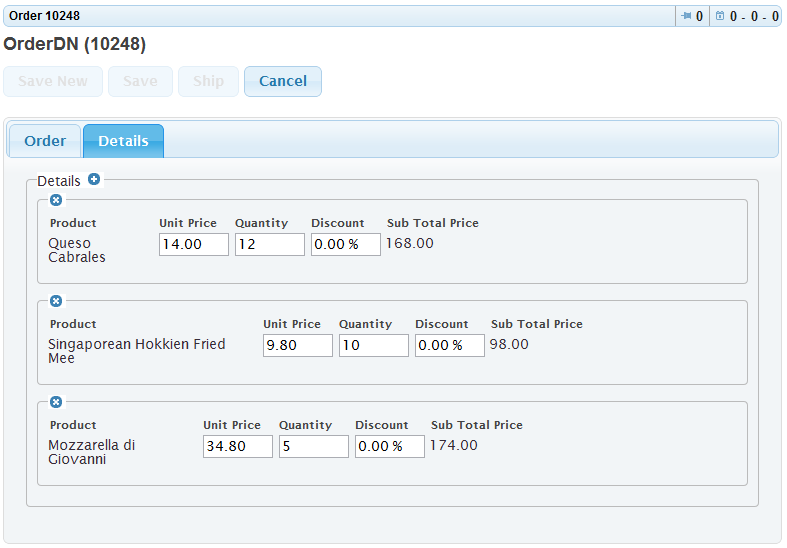
SF.triggerNewContent($("#myDiv")).

});

Make sure to trigger the event just for the new HTML, otherwise you could be creating twice the DatePickers, or the Tabs. });

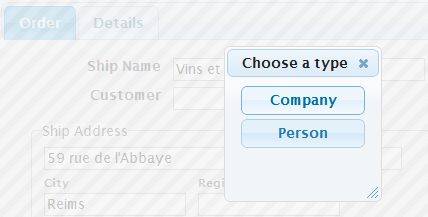
The Order form will look like this:





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 Finding using the EntityLine properties modifier:

@Html.EntityLine(oc, o => o.Customer, el =>

{

el.Finding = "{0}.typedFind({1})".Formato(el.ToJS(),

new JsFindOptions { FindOptions = new FindOptions(typeof(CustomerDN)) }.ToJS());

})

All the Line controls have two overridable properties per operation:

* Create and Creating
* View and Viewing
* Remove and Removing
* Find and Finding

The first of each pair is a Boolean that shows or hides the operation, and the second one is of type String, and contains Javascript code to be executed when clicking the action buttons.

All the Line controls have an equivalent class in Javascript, and all the action buttons operate the same way. First they initialize the Javascript object and then they call the create/find/remove/view method. For example the EntityLine Finding default Javascript code for the Customer property is the following:

new SF.ELine({prefix: 'Customer', …}).find({

navigatorControllerUrl: '/Southwind.Web/Signum/PartialFind',

searchControllerUrl: '/Southwind.Web/Signum/Search'

}, '/Southwind.Web/Signum/GetTypeChooser')

The *find* method takes care of opening the type chooser and then calls the *typedFind* method with the selected type. So overriding the EntityLine method as we did, the resulting Javascript is:

new SF.ELine({prefix: 'Customer', …}).typedFind({

navigatorControllerUrl: '/Southwind.Web/Signum/PartialFind',

searchControllerUrl: '/Southwind.Web/Signum/Search',

webQueryName: 'CustomerDN'})

We have called directly the typedFind method passing the CustomerDN queryName as parameter.

Overriding the Line buttons as well as opening popups, type choosers, search controls directly from Javascript is a more advanced topic that will be covered in a future tutorial. This was just a quick glance at it just to know that it can be done, but after seeing some examples it will all make sense and it won’t seem as frightening as it seems now.

## Conclusion

In this tutorial we have seen how fast it can be to make a complete fully working Asp.Net MVC user interface for our application, thanks to the search controls and the clever default behavior of the entity controls, and a little bit of help of code generation.

We have also seen how the Line Controls are just a set of HtmlHelpers, so they can be used and mixed with any markup we need, letting you express your creativity when necessary with zero extra is cost over a ‘pure’ Asp.Net MVC user interface.

After this tutorial, we have a full understanding of almost all the big pieces of an application built with Signum framework: Entities, Logic, Load and Web user interface. In the next one we will cover Windows user interfaces and we will have covered all the basics.

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

For those interested specifically in Signum.Web, future tutorials will cover how to use the different parts of Signum.Web manually, for example, how to create a button that opens a search control in a popup, and when you select an item from that search control, send it to the server to execute some action with it. With this knowledge we will later on expand on overriding Line action buttons.

I hope the tutorial has been useful for you.