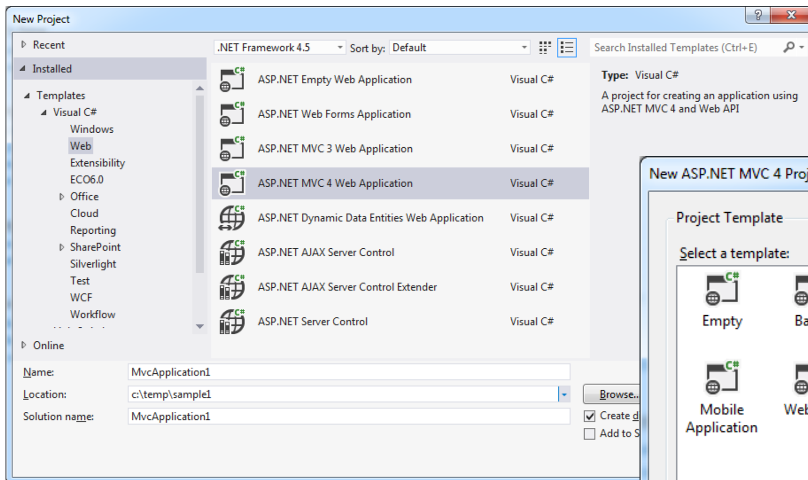
CapableObjects Model Driven Pure and Simple

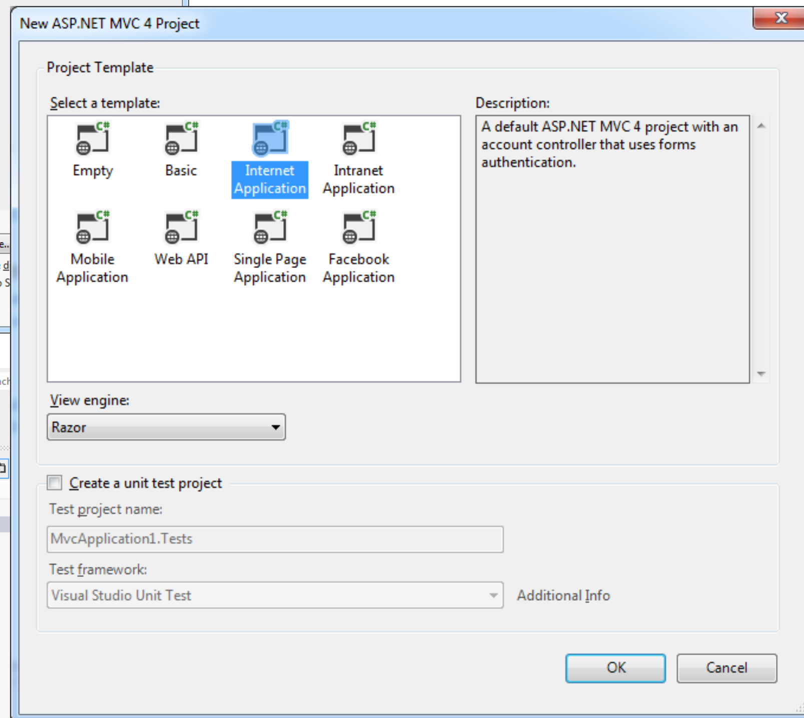
Doing MVC with CapableObjects – a white paper in 4 parts

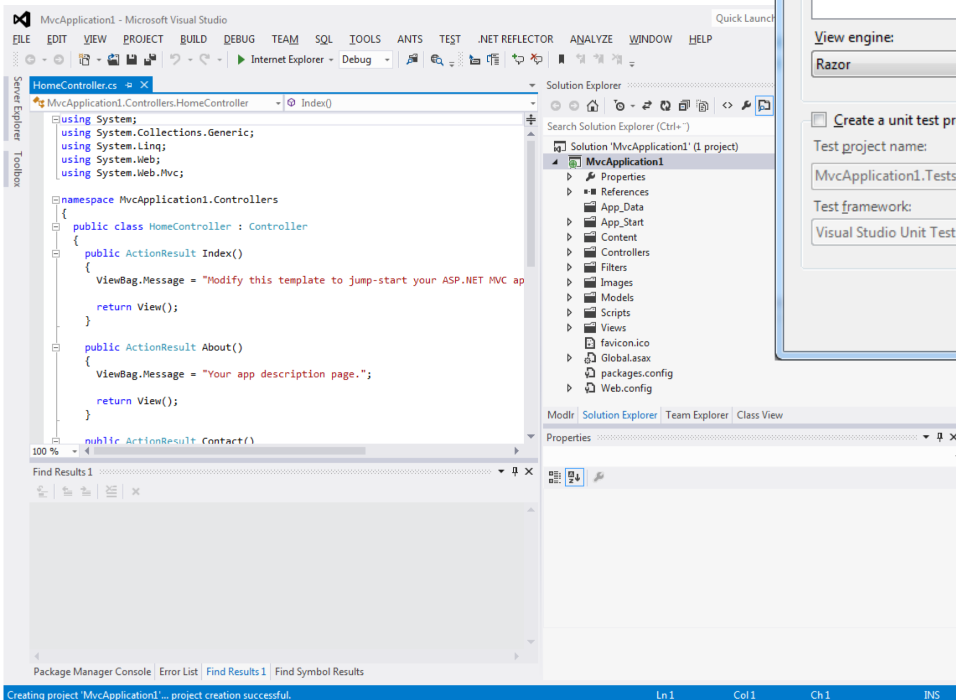
CapableObjects are all about model driven development. The reason why we believe so firmly in model driven development is – all things considered – speed and quality.

MVC – Part1

Model-View-Controller pattern for the web. You can read tons of tutorials and descriptions all around the web – here I will focus on the key aspects of using CapableObjects tools for MVC.

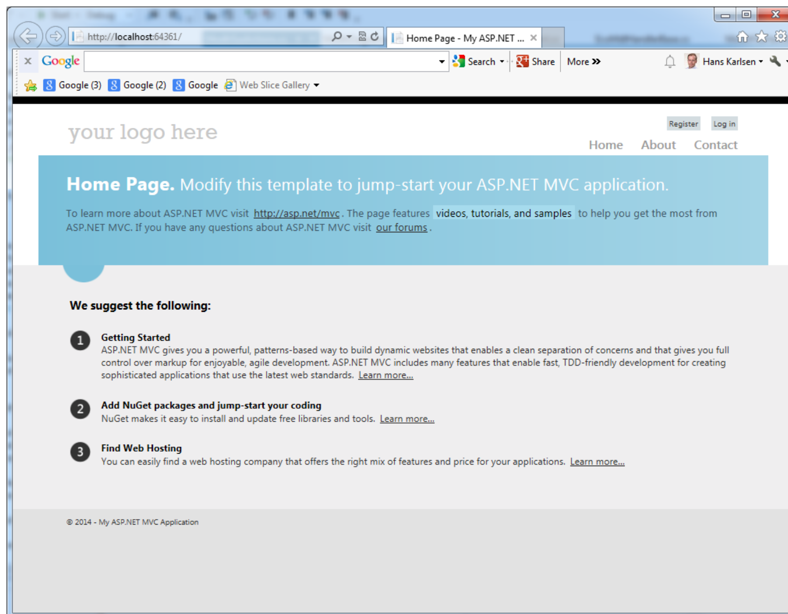






Hit F5 to run it

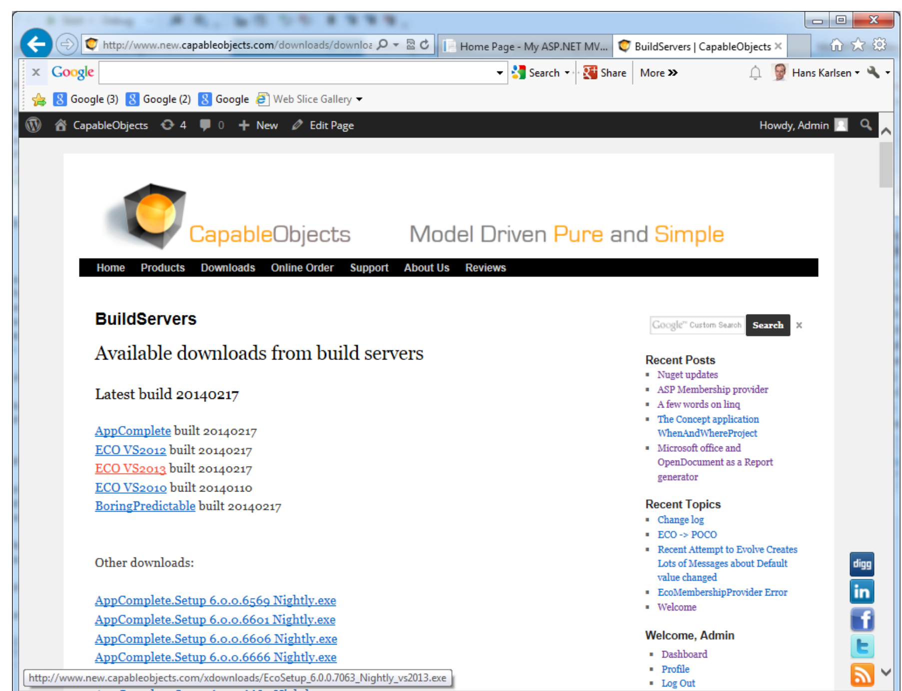
Hans Karlsen M.sc, CTO



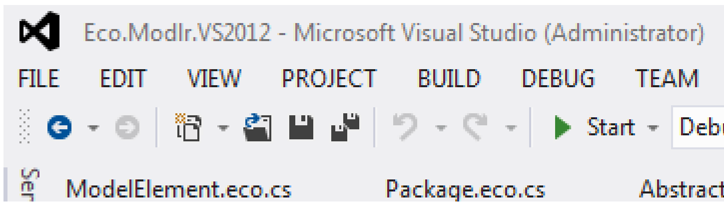
This was standard by the book and this is our starting point. From here CapableObjects can help you with a couple of things:

1. Faster creation of your models
2. Faster creation of your applied models that are geared to a certain controller – aka a ViewModel
3. Managing all things regarding your database schema
4. Easier managing of a constant evolving model – keeping your data as you go along

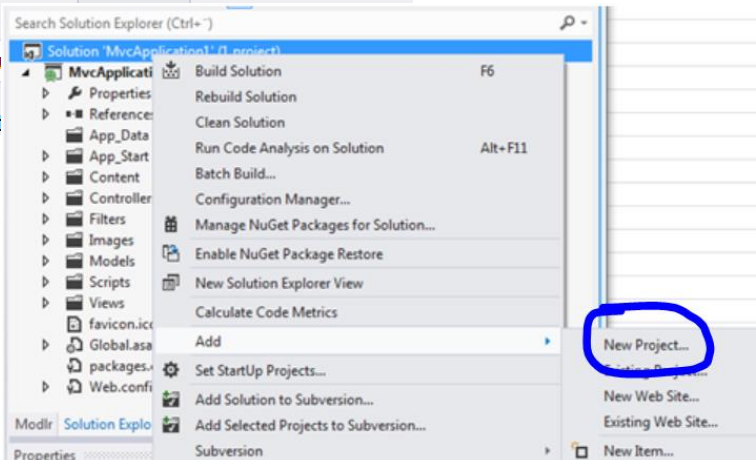
In order to get started we download the Eco install from capableObjects. This is a VisualStudio (2013,2012) plugin.

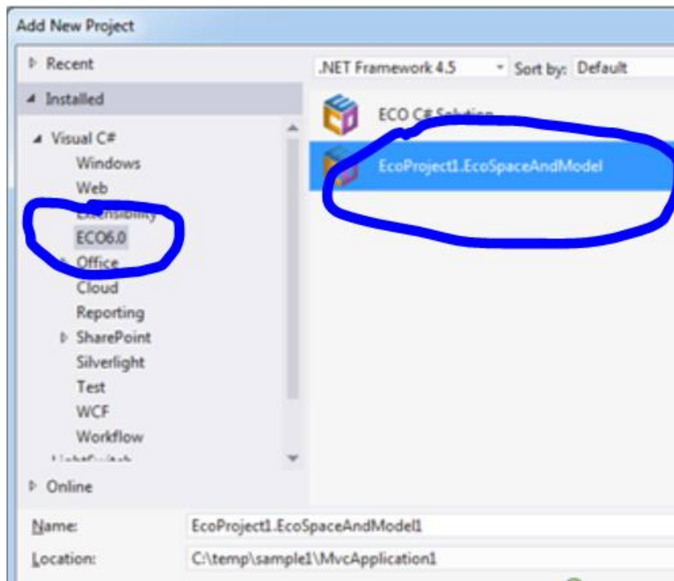


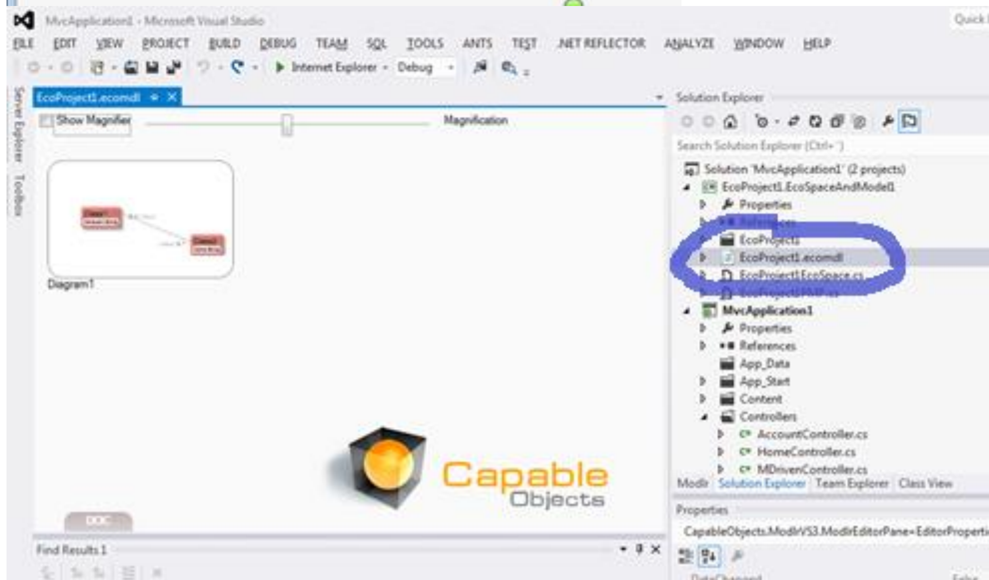
Once you have it you see this:



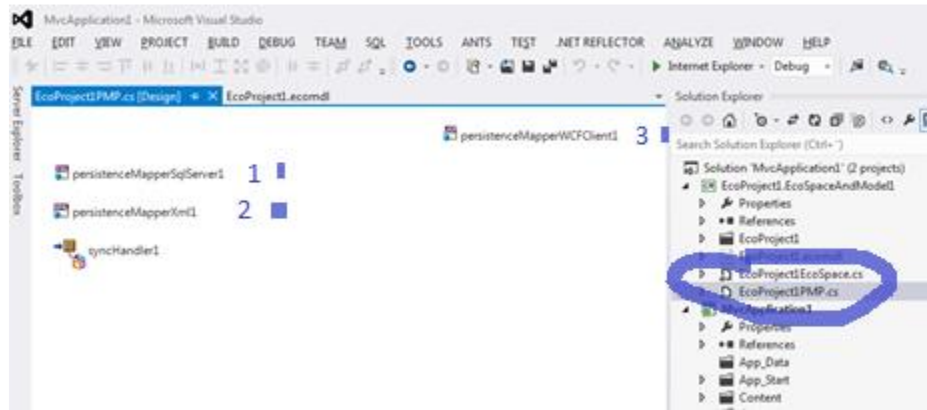
Add a new project to the MVC solution:





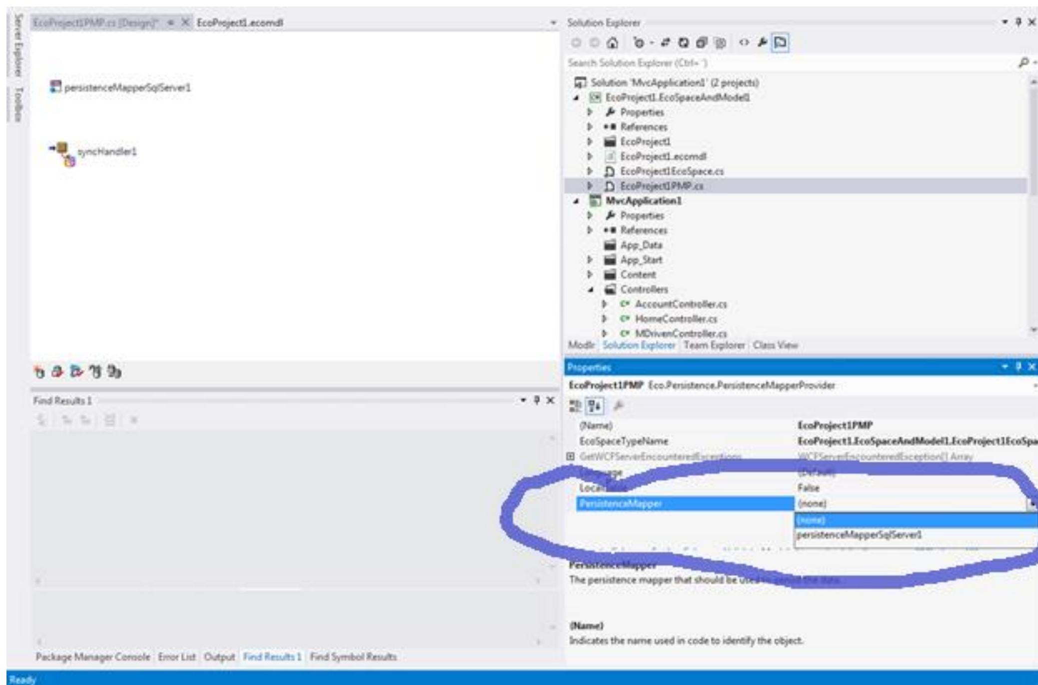


With CapableObjects all entities belong to an EcoSpace – in this sample it is called EcoProject1EcoSpace.cs – we also have the persistence mapper provider (PMP) – this governs how the objects (in the ecospace) are actually stored – in this project it is called EcoProject1PMP.cs. Open it:

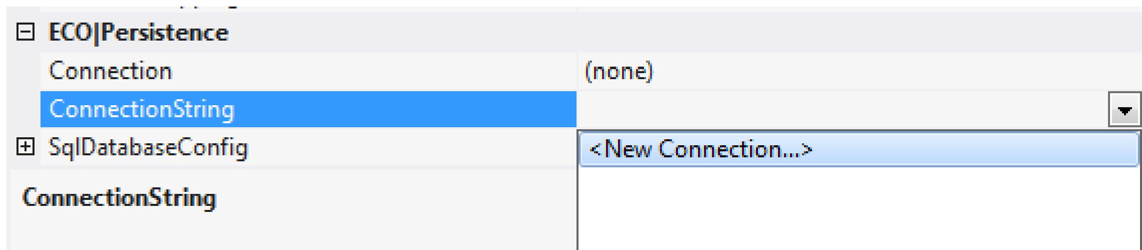


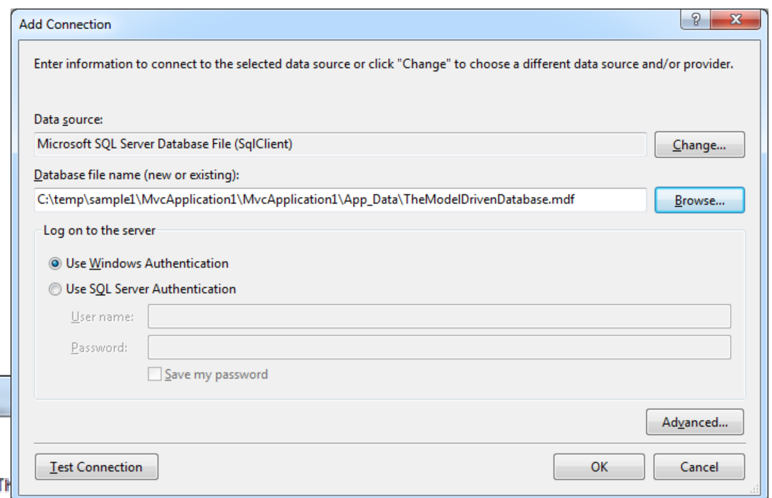
There are three different Persistence mappers there already – we can only use one at a time. Delete persistenceMapperWCFClient1 and persistenceMapperXml1.

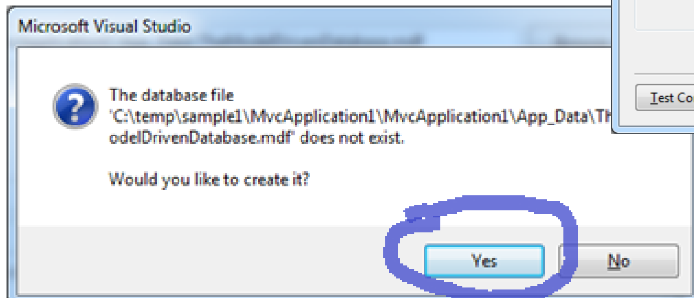
Click the white space in the EcoSpace so that you see the properties in the properties window. Then set the persistenceMapper to the added persistenceMapperSqlServer:



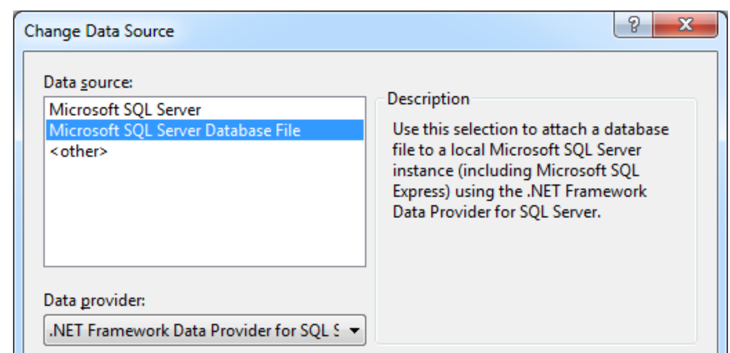
Select the persistenceMapperSqlServer component and set the properties:

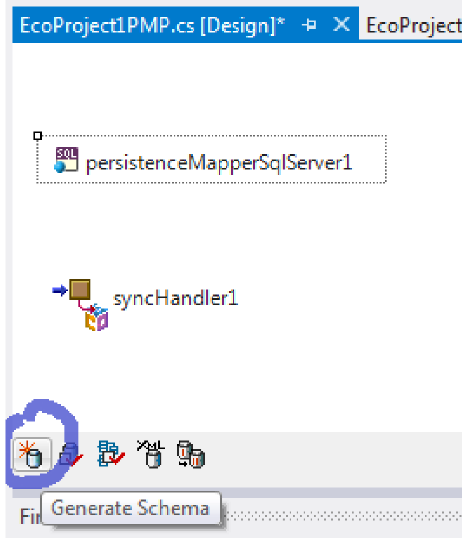


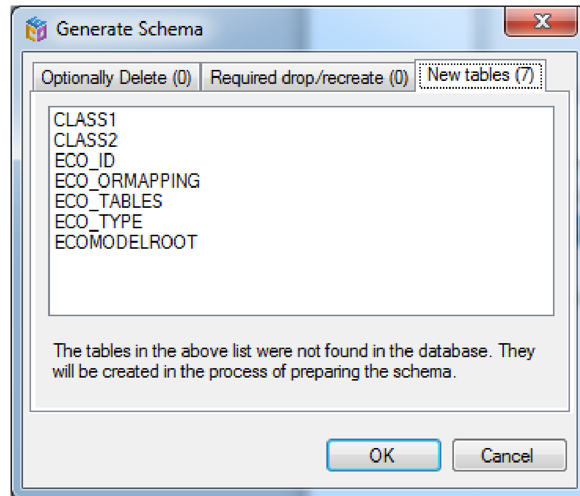


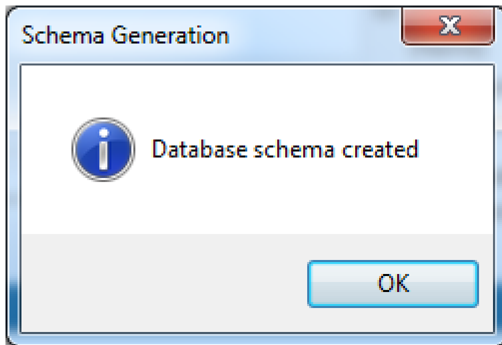


This means we have an empty database.
We need to instruct it to follow our model:



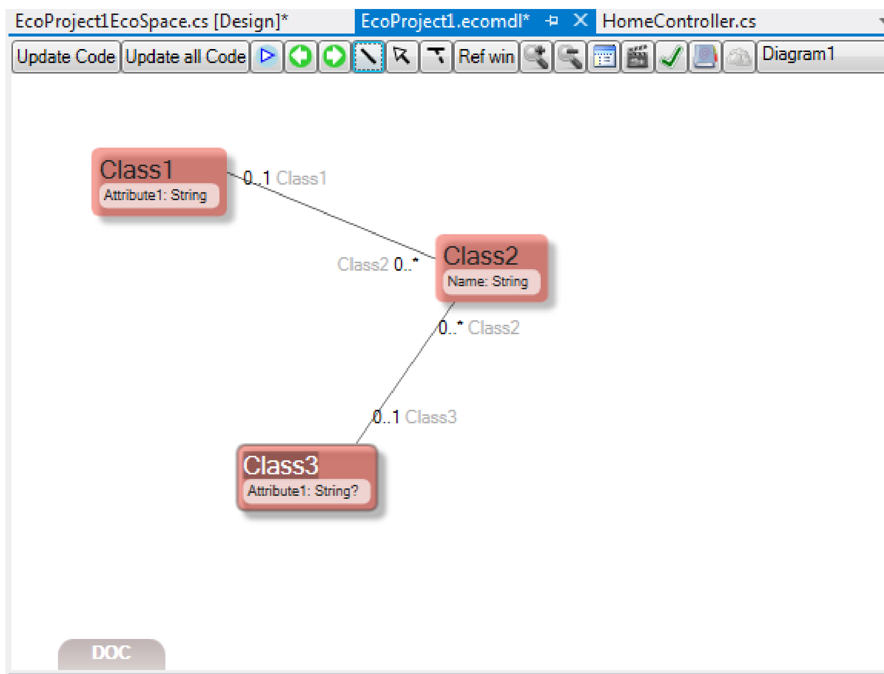






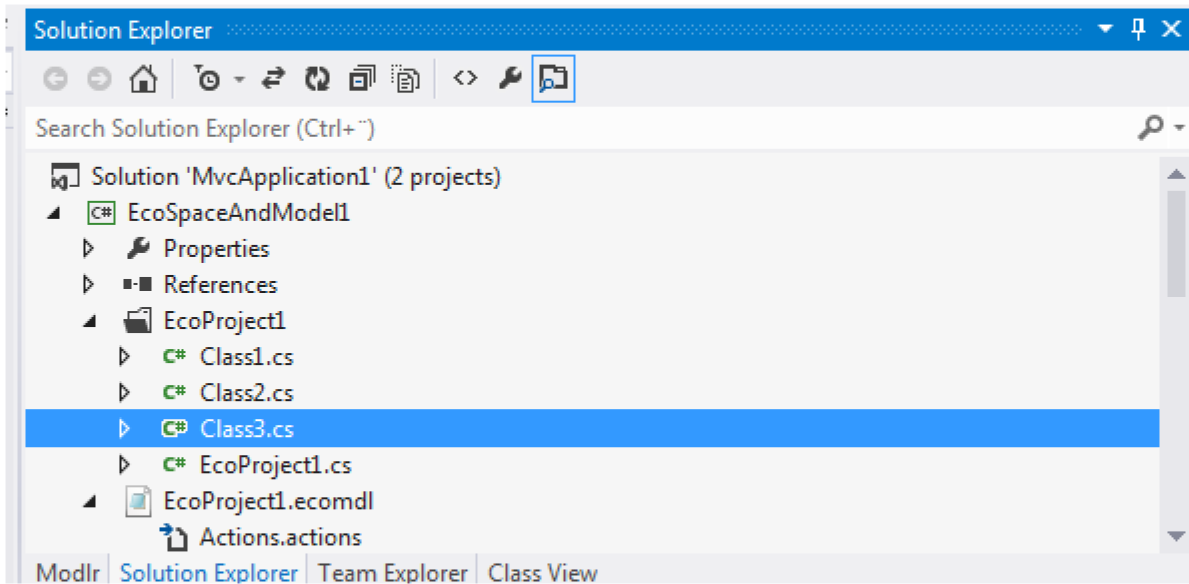
Seven new tables – the ones starting with ECO are maintained by the framework in order to help you with future evolves of the database schema – they are always just five – do not worry, they are there for practical reasons – as you learn more you will find ways to store this model information in other ways if you need to.

The two first tables **CLASS1** and **CLASS2** are the ones from our model.

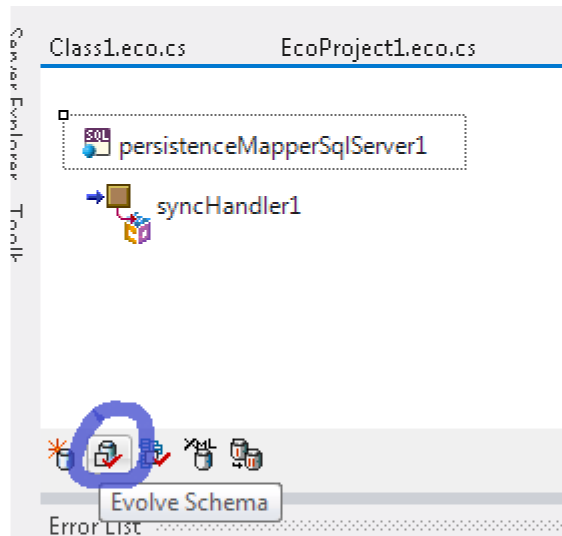


Switch to Model and Add a Class3, add an Attribute and a relation to Class2 :

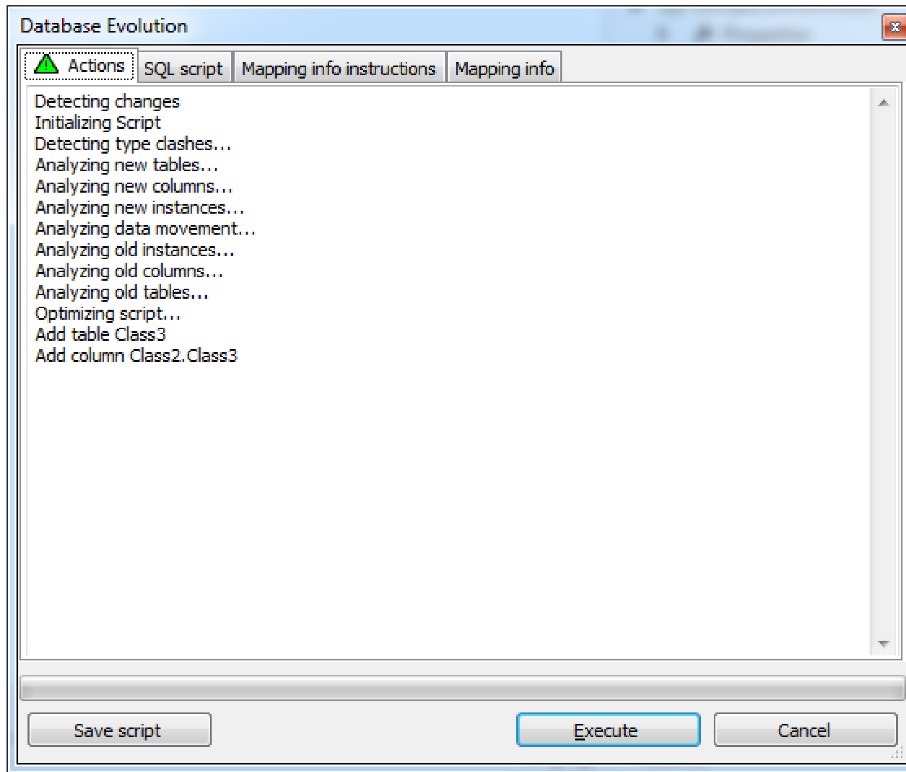
Press Update Code and then Rebuild. A file was added:



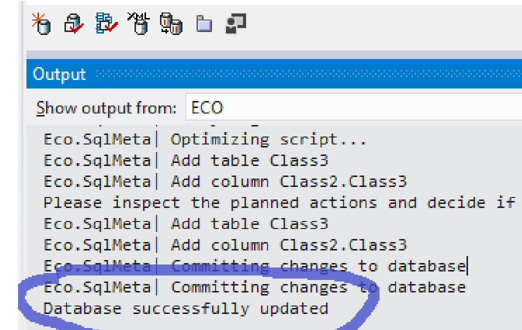
Back to the persistence mapper – we need to instruct the database to evolve to our new model:



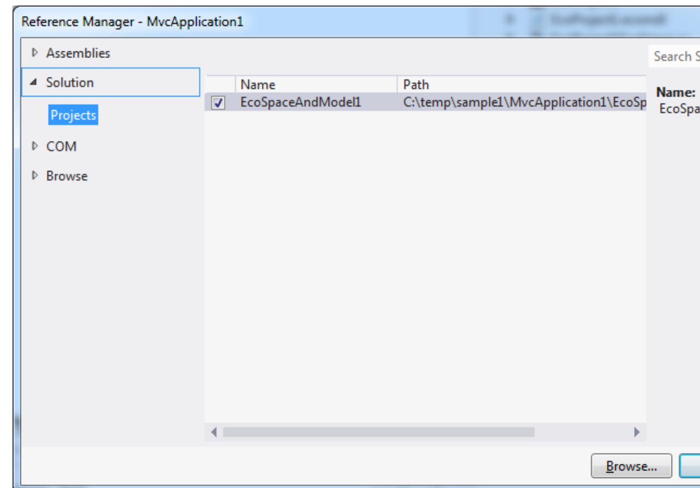
Notice the difference between “Generate Schema” as we did first and “Evolve schema”. If you want to clean your database from all data and start anew -> “Generate”, but if you want to evolve your database – keeping as much data as possible -> “Evolve”. This is a roundtrip you will take many times when you are agile and model driven with CapableObjects.

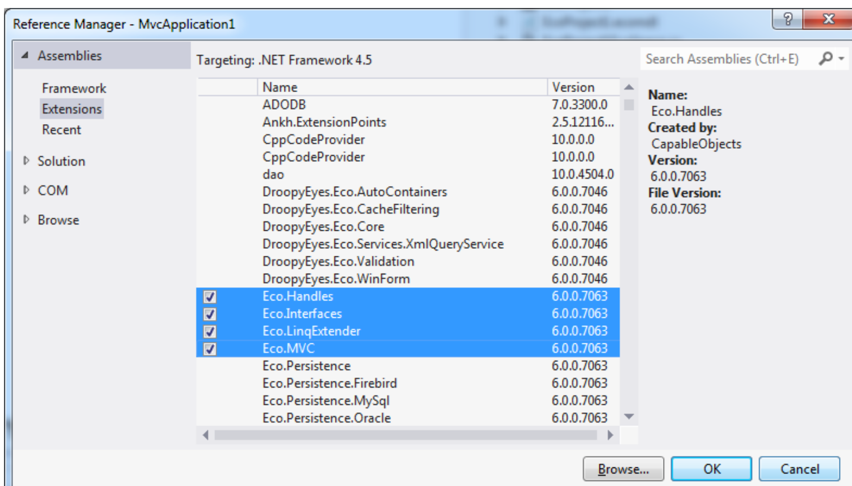


The database schema evolve detected the need for new table, and also a new column on class 2 to hold the foreign key to Class3 (the association we added to the model).

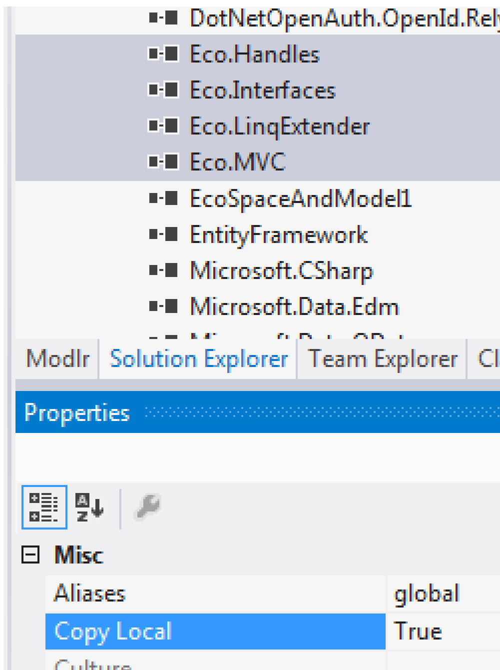


Now it is Time to make use of our model in the MVC project.
Add the following references to the MVC project:

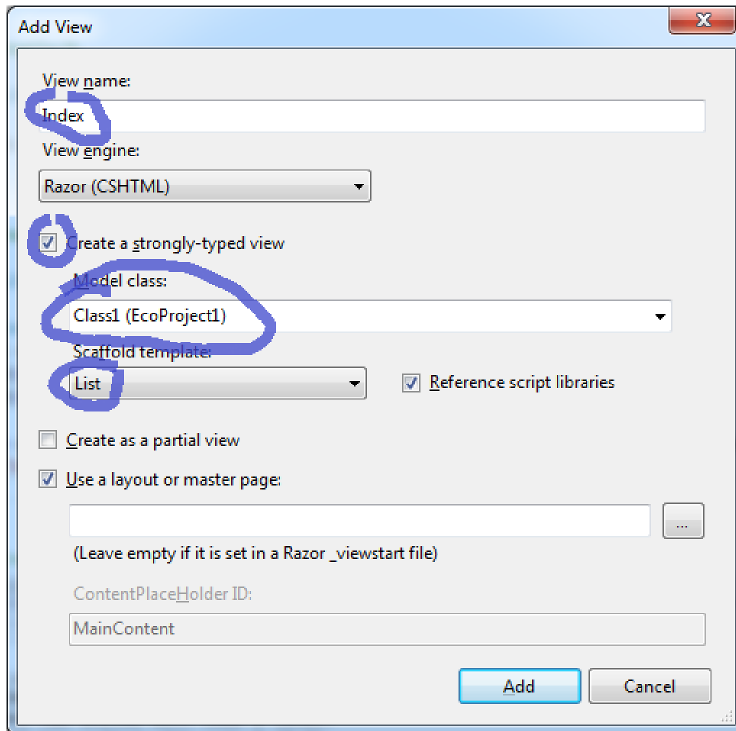


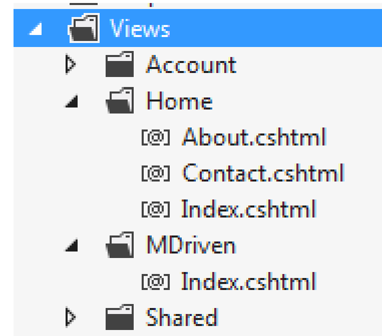


Set Copy Local on these:

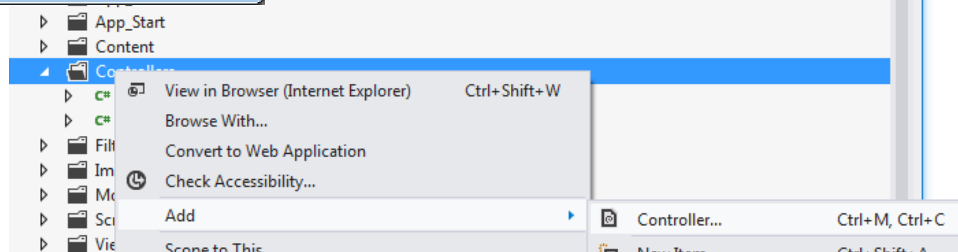


Create a new View "Index" in a new Views folder MDriven, choose Model class Class1, Choose scaffold template "List":





Since this view was placed in MDriven it will look for MDrivenController to handle all actions. So create MDrivenController:



We are going to change the baseclass from a standard controller to a

controller that knows that we are model driven and that knows that we want to use our ecospace:

```
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class MDrivenController : Controller
    {
```

Like so:

```
using System.Web.Mvc;
using Eco.Handles;
using Eco.Linq;
using Eco.MVC;
using EcoProject1;

namespace MvcApplication1.Controllers
{
    public class MDrivenController : EcoController<EcoProject1EcoSpace>
    {
```

Add these;

```
using Eco.Handles;
using Eco.Linq;
using Eco.MVC;
using EcoProject1;
using EcoProject1.EcoSpaceAndModel1;
```

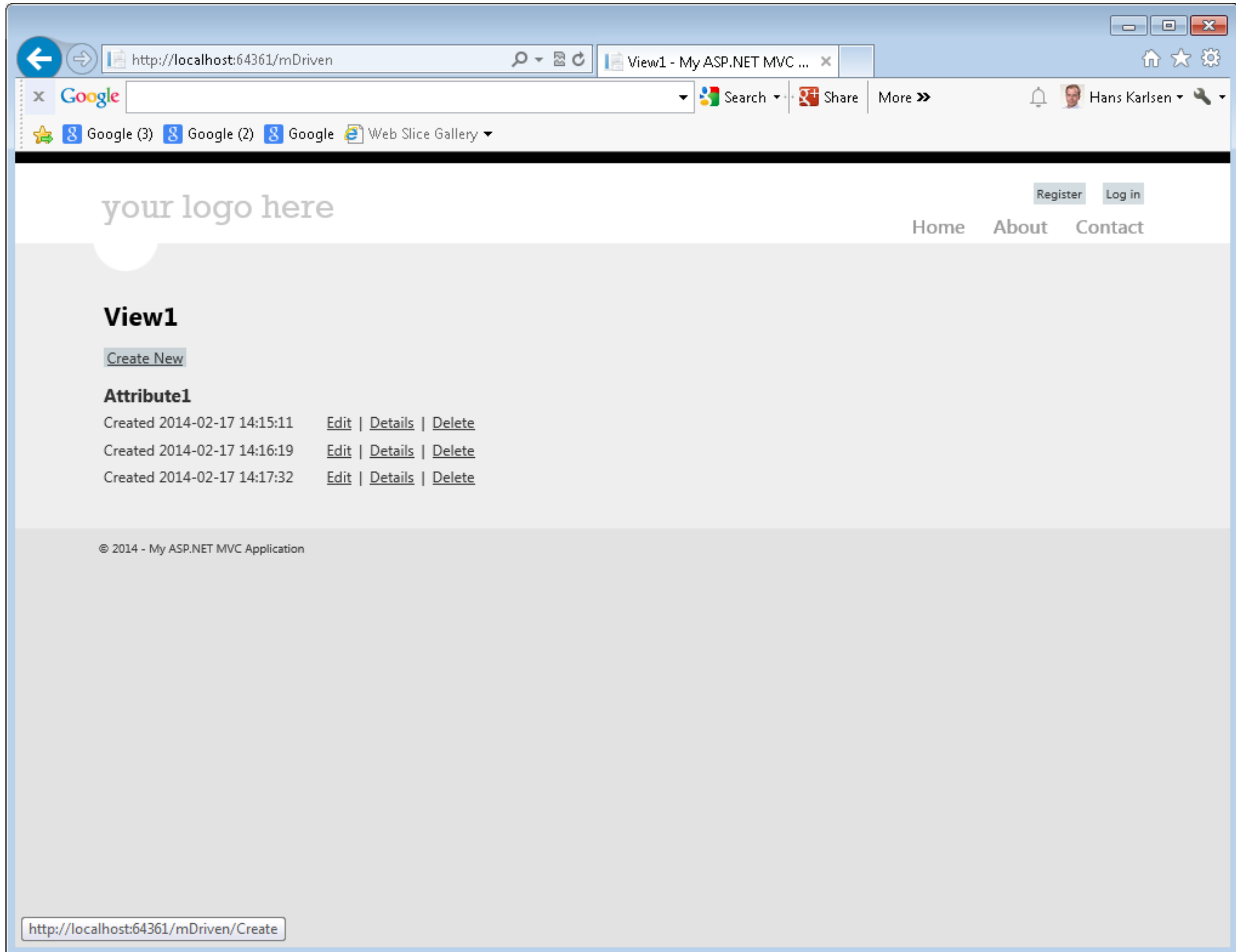
Now we can use the ecoSpace to get to the model driven data:

```
public ActionResult Index()
{
    var z = (from x in EcoSpace.PSQuery<Class1>() orderby (x.Attribute1) select (x)).Take(15).ToList();
    return View(z);
}

public ActionResult Create()
{
    var c1=new Class1(EcoSpace);
    c1.Attribute1 = "Created " + DateTime.Now.ToString();
    c1.Class2.Add(new Class2(EcoSpace) { Name = "First c2 of this c1" });
    c1.Class2.Add(new Class2(EcoSpace) { Name = "Second c2 of this c1" });
    Commit();
    return RedirectToAction("Index");
}
```

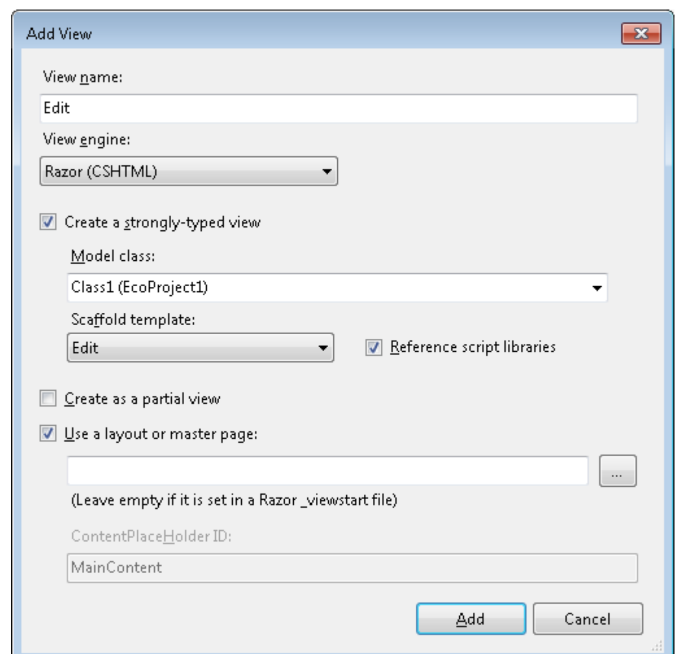
When we create an object that should be owned by our ecospace – we send in a reference to this ecospace in the constructor. In the code above we use Linq to – thru our persistence mapper – ask the database of all the instances of Class1.

Our controller has a method Commit that saves the objects via the persistence mapper to the database.

We run:



Let us a take a look at edit. Adding another view in the MDriven folder:



I fix the Index view's action so that the Class1 ExternalId (every object has one with us) is passed into edit:

Add a using statement at the top of the cshtml file;

```
@using Eco.ObjectRepresentation
```

```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.ExternalId()  }) |
    @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */
```

And add an action handler to the controller:

```
public ActionResult Edit(string id)
{
  var x=ObjectForId(id).GetValue<Class1>();
  if (x != null )
    return View("Edit",x);
  return View();
}
```

The method gets an identity and fetches the corresponding object that is sent to the new Edit view:

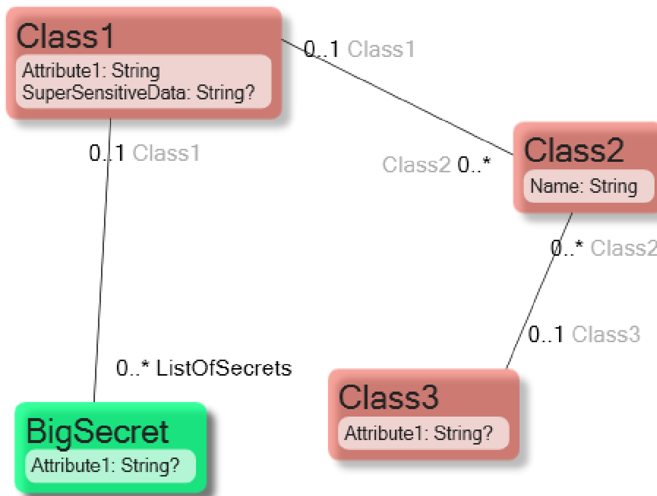
And when I click save the form is posted back to the controller. I have added this method to handle this:

```
[HttpPost]
public ActionResult Edit(string id,Class1 c1)
{
    try
    {
      var x = ObjectForId(id).GetValue<Class1>();
      if (x != null )
      {
        ApplyValues(c1, x);
        Commit();
      }
      return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The c1 we get as an argument is a data-wise copy of the ecospace Class1 object we sent to edit –it has round tripped to the browser as html and been posted back as form data. We call such objects for Offline-objects. We use the a method of our modeldriven controller to apply values from offline to the online counterpart. Done. We have listed, edited and updated. We have been strongly typed all the time and we are well equipped for model evolution.

MVC with CapableObjects – Part 2

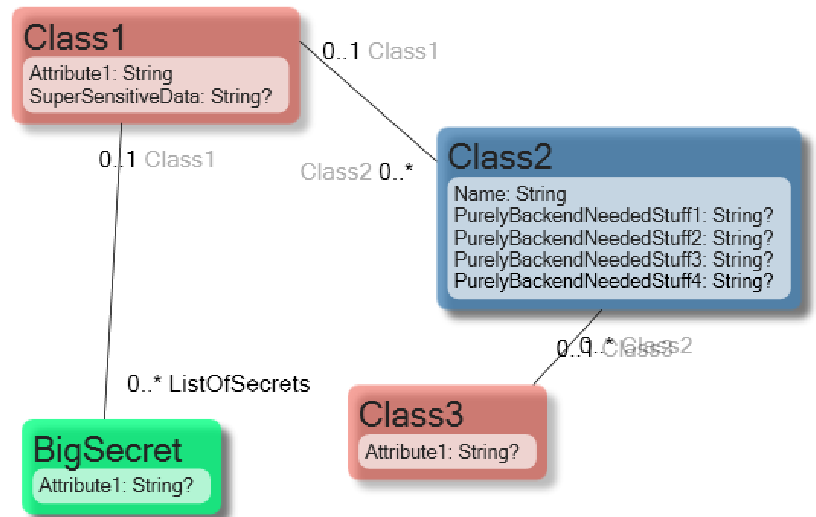
What if our model looked like this:



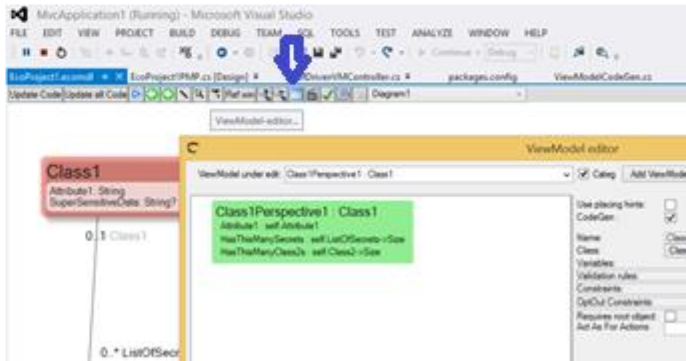
If the model resembles this you may not be comfortable with throwing around those BigSecrets or supersensitivedata in your MVC views?

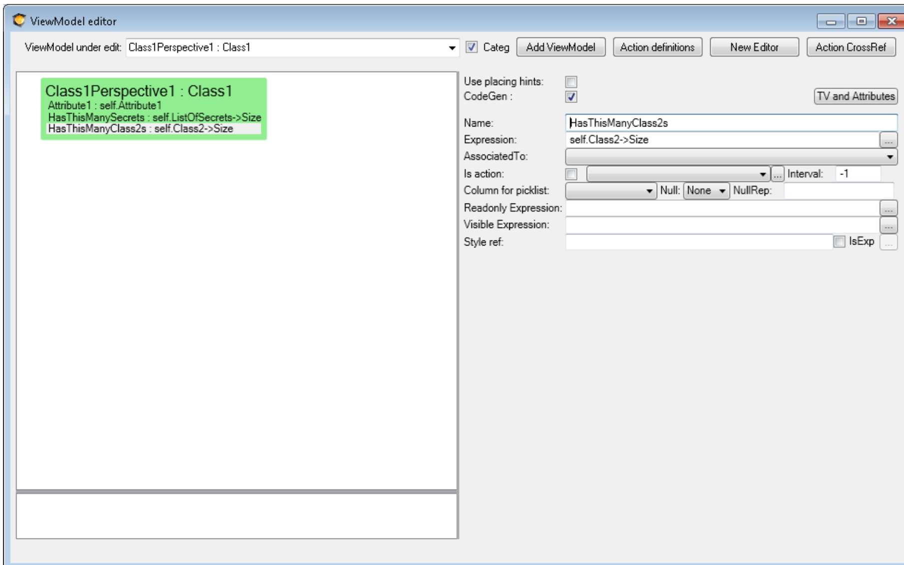
Or what if it looks like this:

It is at best confusing to show all the internal stuff to the front end developers. There is always the risk that you as a frontend developer expose something you should not.



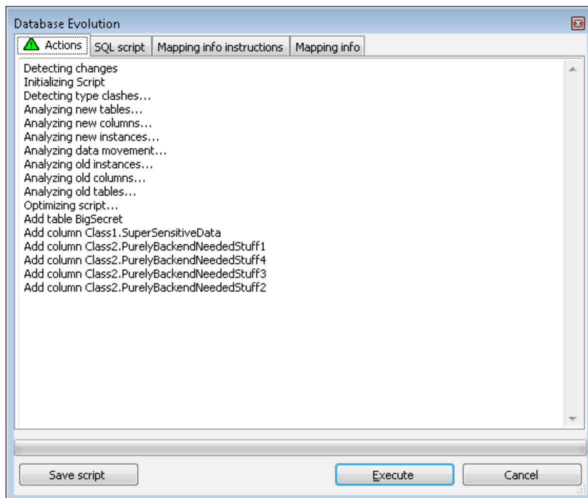
These are the main two reasons for slicing your data in better – more to the point – chunks. Chunks adapted to given tasks. Such chunks are views or perspectives of your whole model. We call them ViewModels.





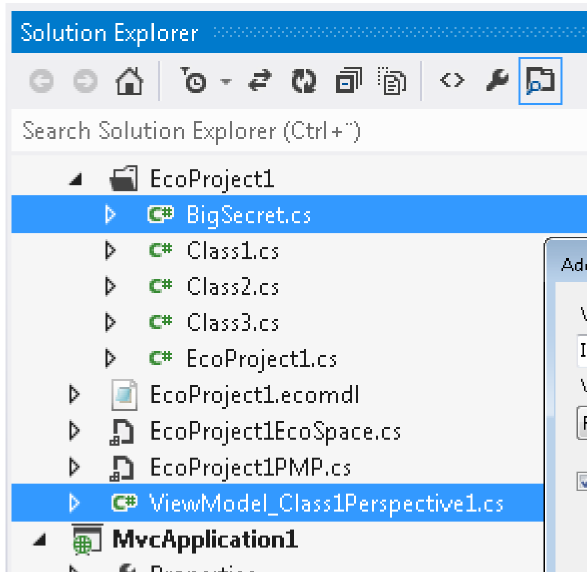
ViewModels as CapableObjects defines them are always rooted in a ModelClass. You then add named ViewModel properties – called ViewModelColumns to the ViewModelClass. A ViewModelColumn is always derived with an expression in OCL. The expressions can be very simple like "self.Attribute1" or more complex as "self.ListOfSecrets->size". It is easy and fast to declare ViewModels in the model.

When a ViewModel has the "CodeGen"-checkbox checked – it will manifest itself as code that is maintained and updated whenever you press Generate code in the model surface.

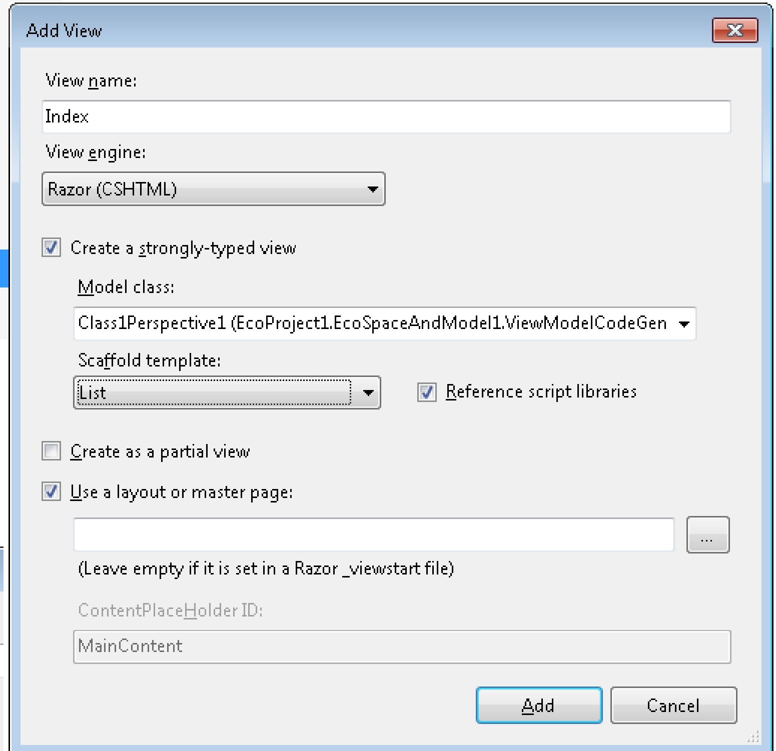


So I Generate code, and build, execute Evolve Model from the PMP – this to update the database with all the changes I did to the model:

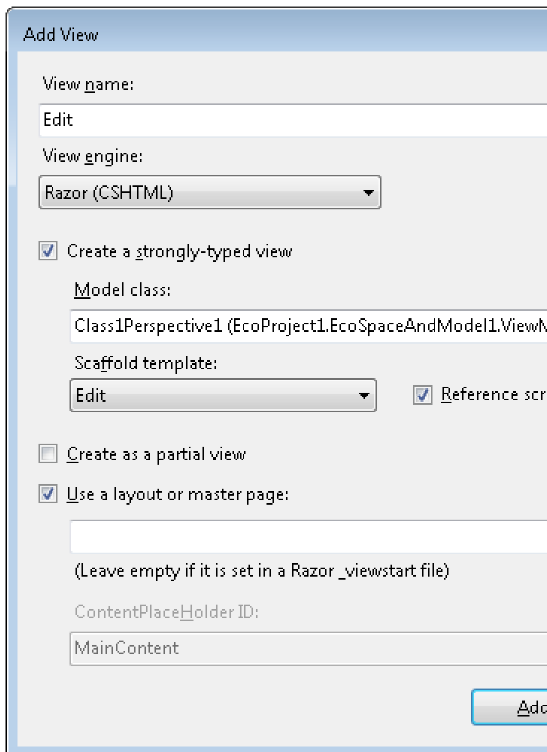
The new files are added to the project:

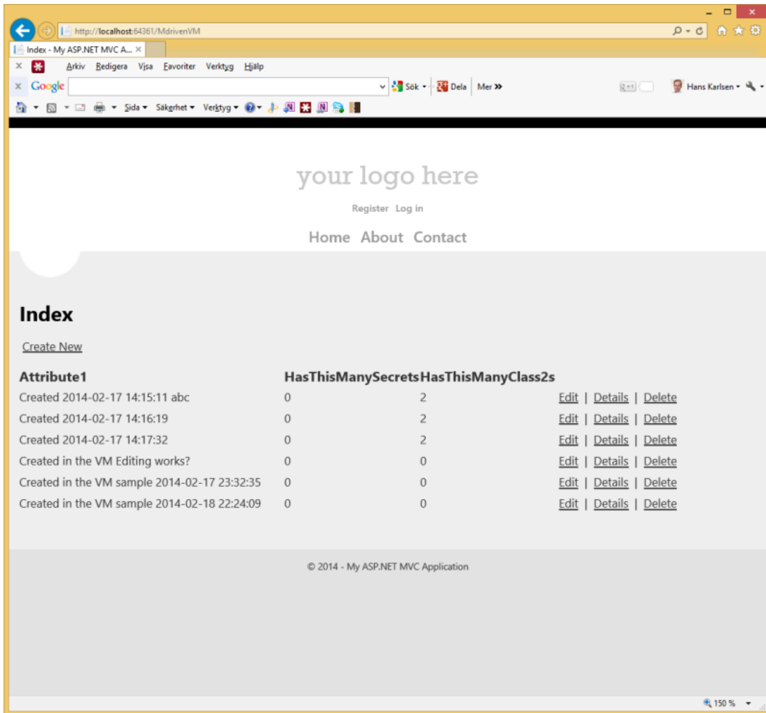


I know want to use the Class1Perspective instead of the Class1.
I do this in a new controller MDrivenVM – Lets do the View first:

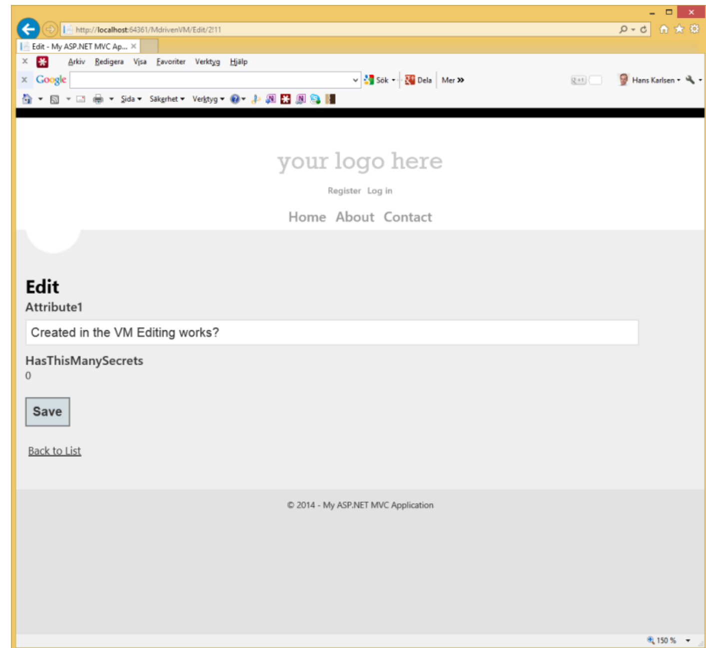


And an edit view:





And edit:



To handle this I added this code in the controller:

```
public ActionResult Index()
{
  var z = (from x in EcoSpace.PSQuery<Class1>() order
  var zvm= from x in z select(Class1Perspective1.Crea
  return View(zvm);
}

public ActionResult Create()
{
  var c1perspective1 = Class1Perspective1.Create(EcoS
  c1perspective1.Attribute1 = "Created in the VM samp
  Commit();
  return RedirectToAction("Index");
}
```

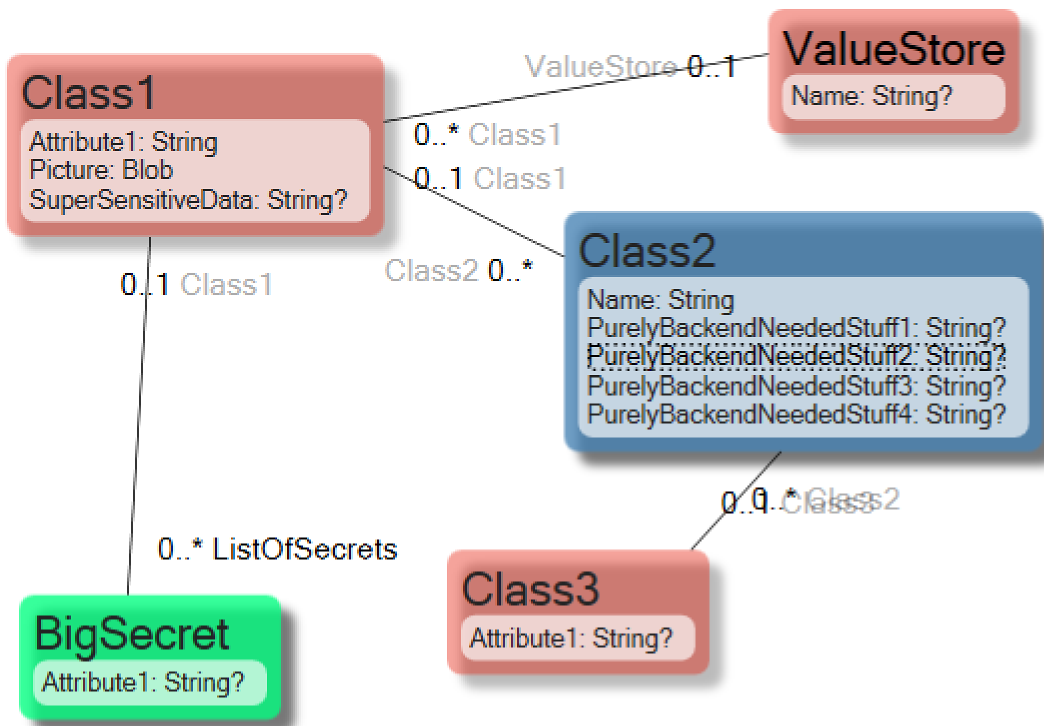
And

```
public ActionResult Edit(string  id)
{
  var x = ObjectForId(id).GetValue<Class1>();
  var c1perspective1 = Class1Perspective1.Create(EcoSpace, x);
  return View("Edit", c1perspective1);
}

[HttpPost]
public ActionResult Edit(string id, Class1Perspective1 Class1Perspective1Offline)
{
  try
  {
    // TODO: Add update logic here
    var x = ObjectForId(id).GetValue<Class1>();
    if (x != null )
    {
      ApplyValues(Class1Perspective1Offline, Class1Perspective1.Create(EcoSpace, x));
      Commit();
    }
    return RedirectToAction("Index");
  }
  catch
  {
    return View();
  }
}
```

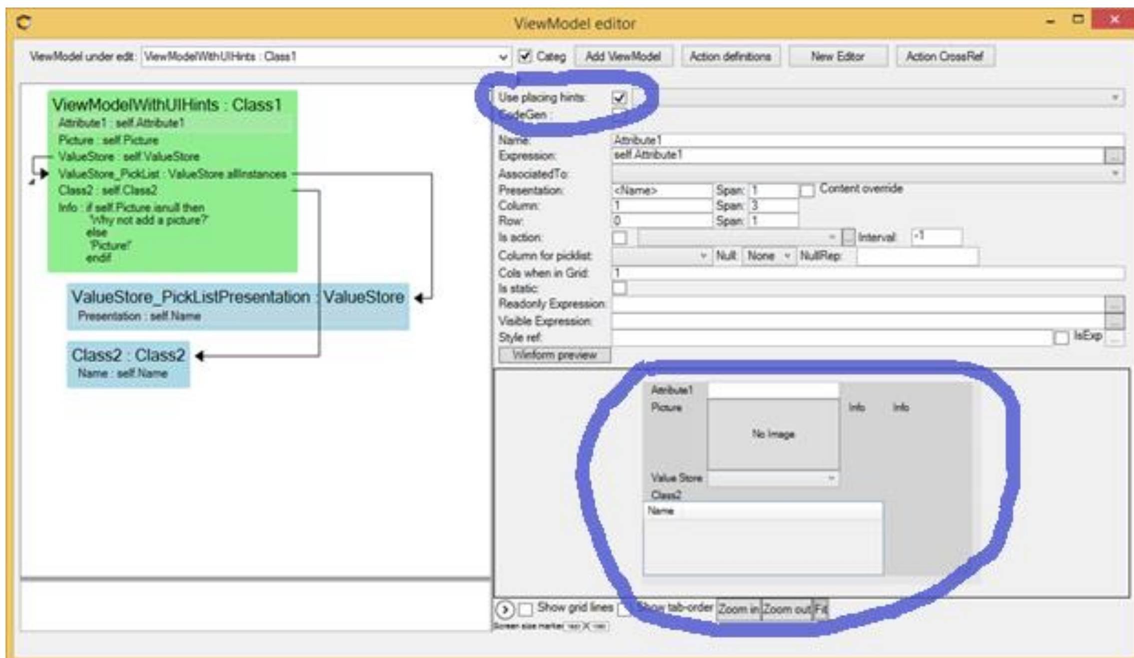
MVC with CapableObjects Part 3

Have you ever wished that the view scaffolding would be a little more flexible? That it could do more stuff directly like images and selectlists? That it would follow your model even when you change it? If it were just a tiny bit better we could really wip something together superfast and see what the user thinks of it: Keep? Improve? ChangeMyMind?

With CapableObjects ViewModels you can. Each ViewModel column can hold user interface hints.
Changing the model again:

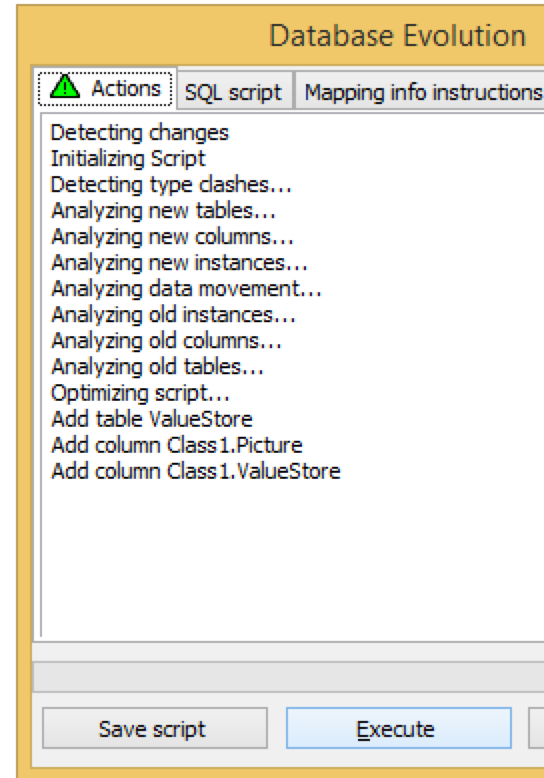


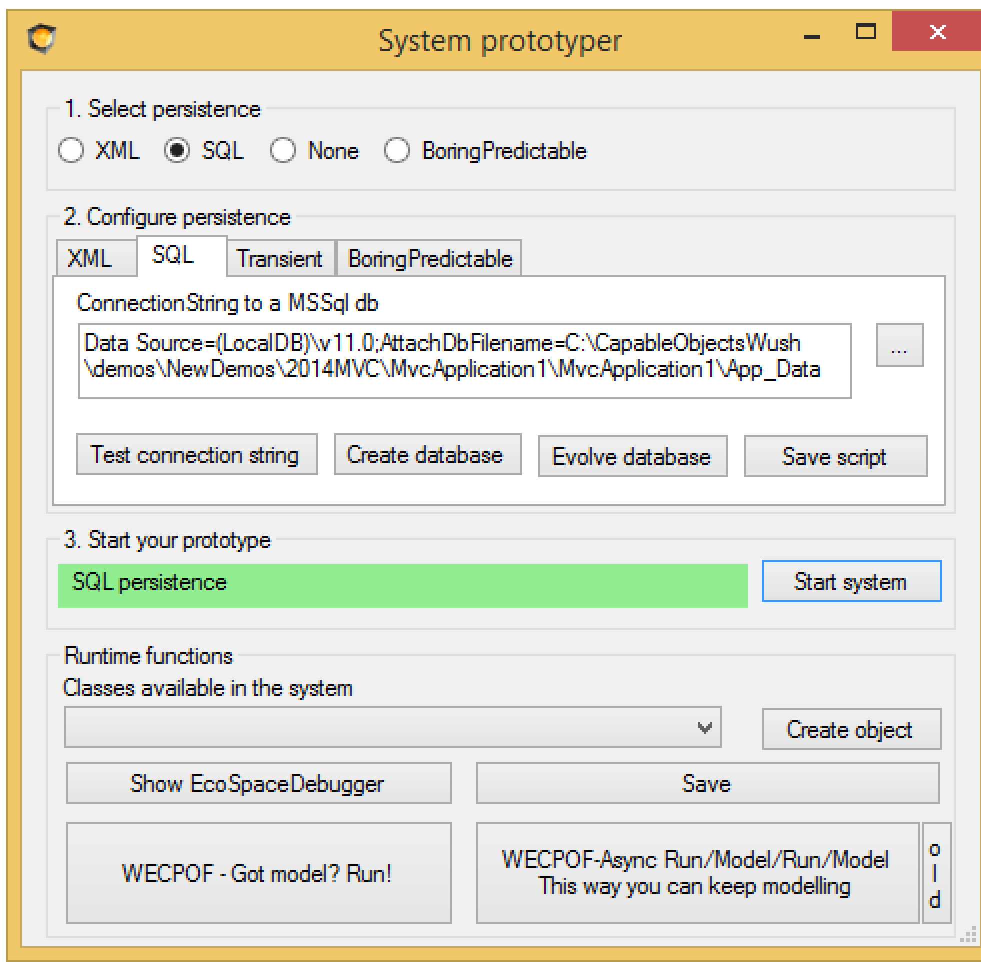
And a new ViewModel – now with UI-hints:



It is easy to get carried away since you have big help of the model awareness that gives you shortcuts to the most obvious things. I added a text field, a picture and a selectlist, followed by a grid. I then also added a static info field that will write something depending on if the picture is set or not.

Evolve the database again:





I will click up the prototype tool from the model surface:

I can enter the connection string to our database click Show EcoSpaceDebugger.

Here I can create some objects for the ValueStore class:

I change our editing actionhandlers in the MDrivenVMController to use the new ViewModel:

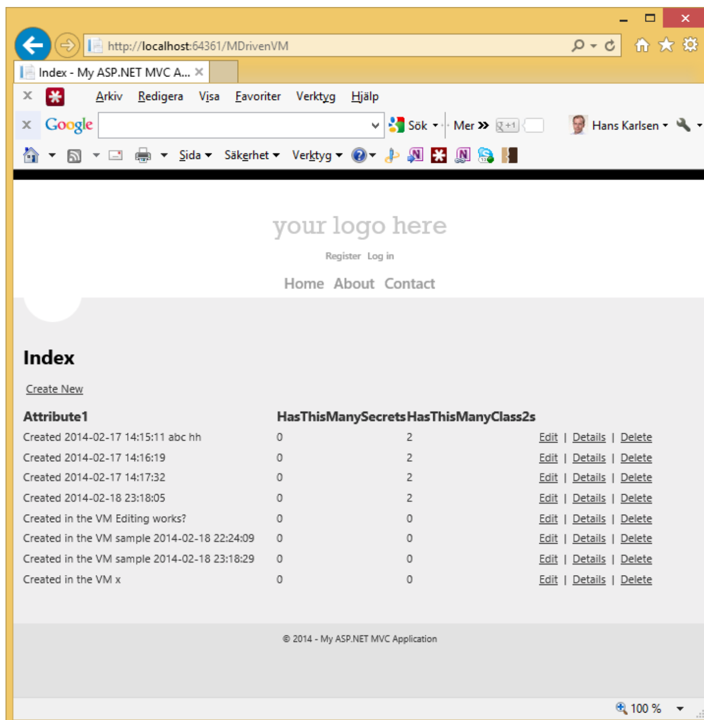
```
public ActionResult Edit(string  id)
{
  var x = ObjectForId(id).GetValue<Class1>();
  var c1perspective1 = ViewModelWithUIHints.Create(EcoSpace, x);
  return View("Edit", c1perspective1);
}

[HttpPost]
public ActionResult Edit(string id, ViewModelWithUIHints aViewModelWithUIHints)
{
  try
  {
    // TODO: Add update logic here
    var x = ObjectForId(id).GetValue<Class1>();
    if (x != null )
    {
      ApplyValues(aViewModelWithUIHints, ViewModelWithUIHints.Create(EcoSpace, x));
      Commit();
    }
    return RedirectToAction("Index");
  }
  catch
  {
    return View();
  }
}
```

I must also change the View. It uses another model – and I also want it to use all the UI-hints from the model. I make this happen with the DisplayWecpofUI call:

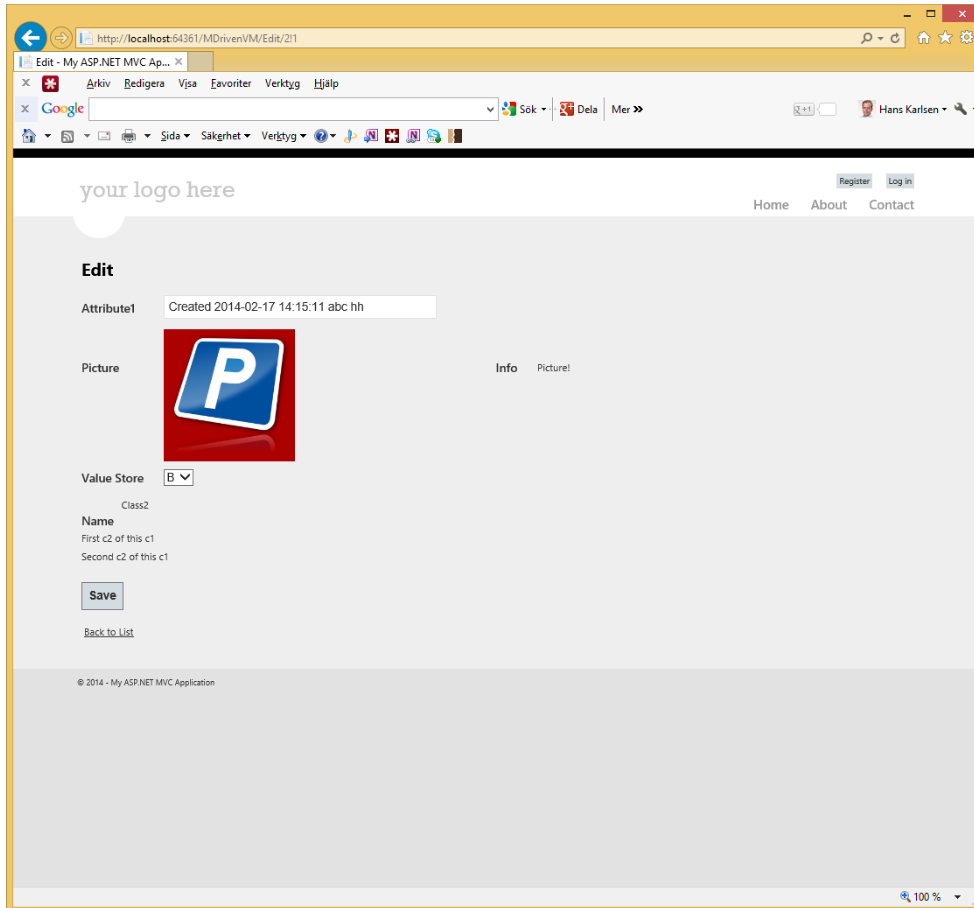
Edit.cshtml ViewModel_ViewModelWithUIHints.cs EcoProject1.ecomdl EcoProject1PMP.cs [Design]

```
@model EcoProject1.EcoSpaceAndModel1.ViewModelCodeGen_ViewModelWithUIHints.ViewModelWithUIHints
@using Eco.MVC
@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>ViewModelWithUIHints</legend>

        <div>@Html.DisplayWecpofUI()</div>

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

And run:



Edit:



We get picklist, an image, an editbox – all databound to viewmodel data, we also get actions that can execute any method you add in your model – in c#.
Combine your UI hints with Style references to your css to get the look you want.

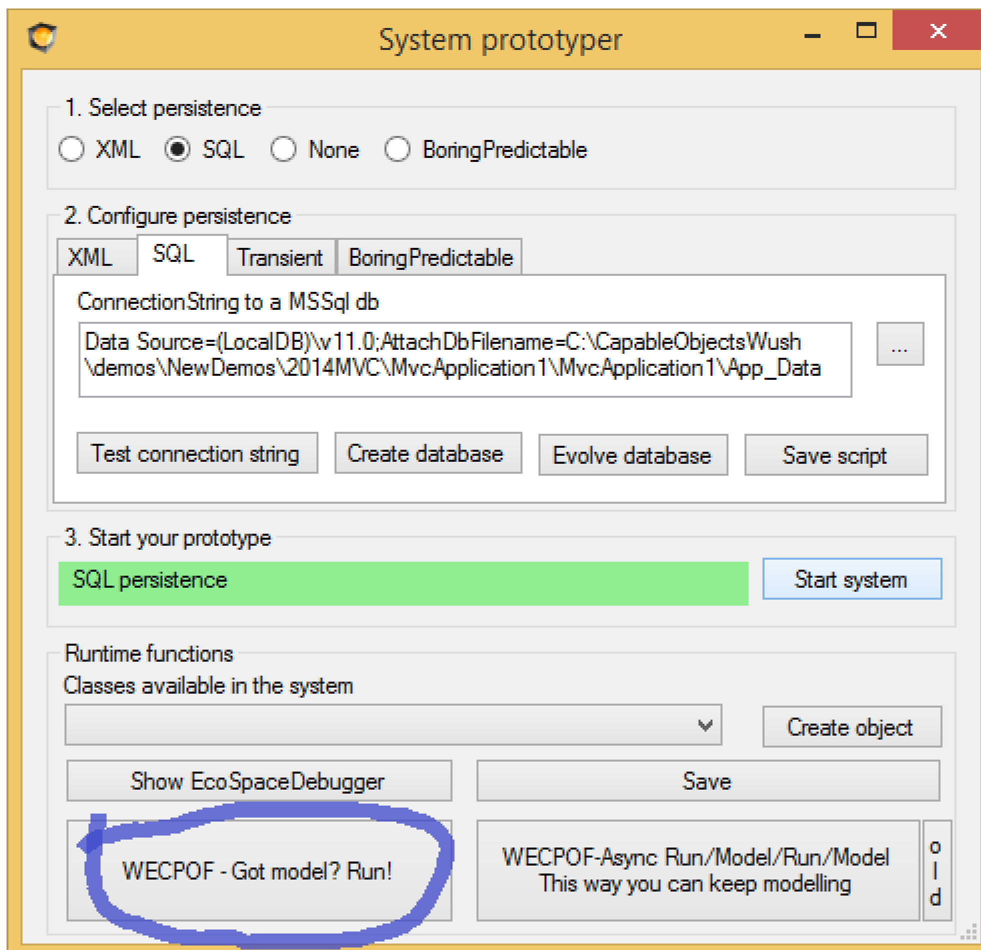
So using UIHints in the viewmodel can be a real boost in development time. If you can manage all the simple and standard admin UI like this – imagine how much time you free up to do the advanced cool stuff that you dream about.

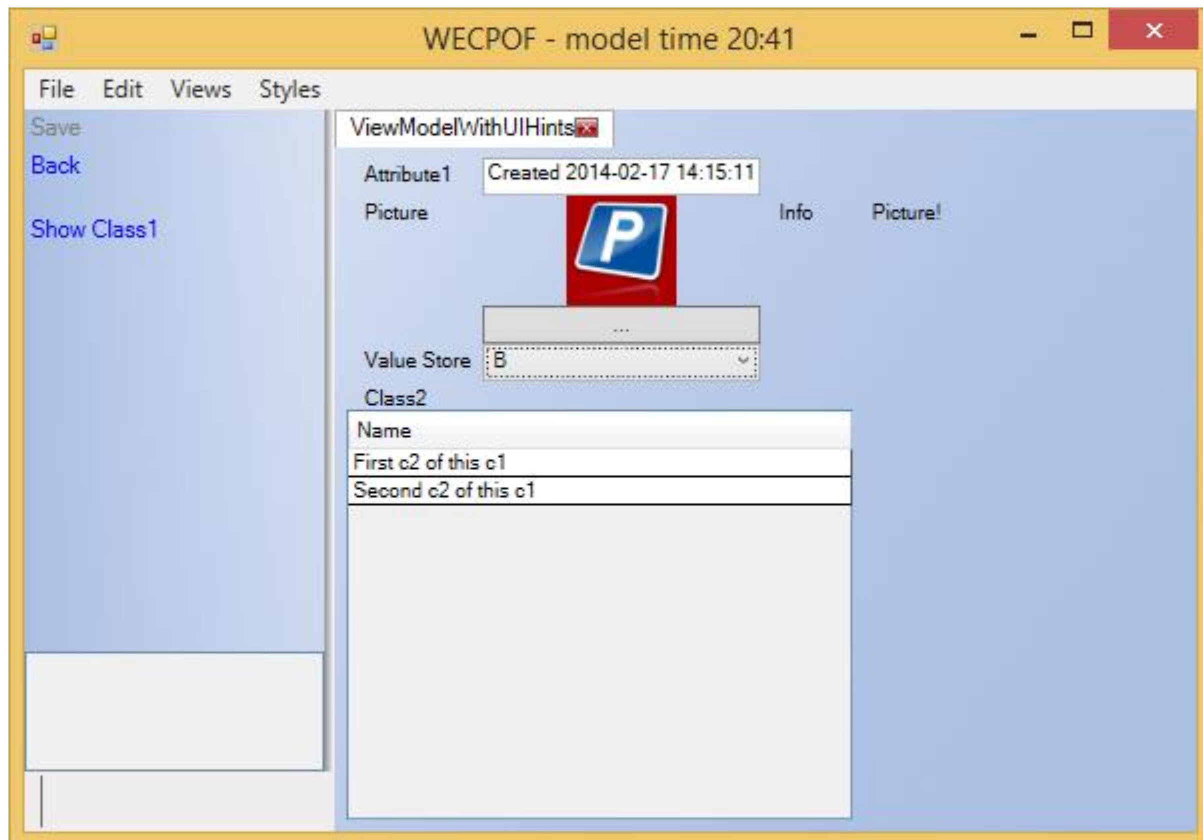
CapableObjects Model Driven Pure and Simple

MVC with CapableObjects Part 4

At CapableObjects we are constantly looking for new ways of moving definition from code into the model, our aim is to build things faster with higher quality. The result of our strive is a concept we call wecpof. Wecpof is the ability to declaratively define what views a system has, how navigation is done between these views and what actions that can be done on the information. Wecpof was first tested in WPF applications – and the play button in the model-surface will take you to a wecpof driven executing version of your model.

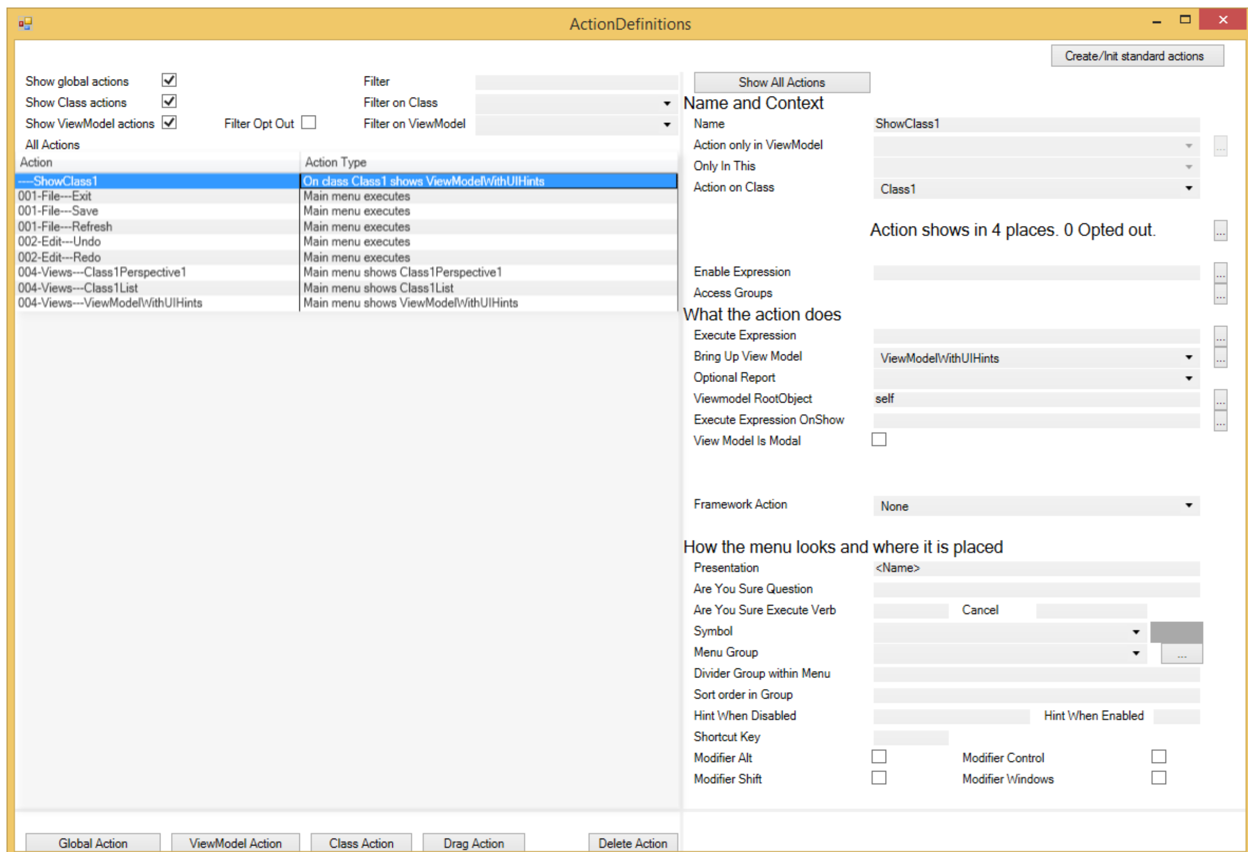
Take a look:





Same things as you saw in MVC are now in WPF.

To get WECPOF working we need actions. Actions can either do stuff (change data) or navigate (change view). Take a look at the actions currently in our model



You can very fast build advanced WPF applications with Wecpof – and as I will show you next – you can also use the wecpof concept in MVC.

I will create yet another controller, this time I let it inherit from ModelDrivenControllerBase. I will only implement the Index action handler – telling it to use the GenericView view that I will add next:

GenericView.cshtml | MDrivenWecpofController.cs | Edit.cshtml | EcoProject1.ecomdl | EcoProject1

MvcApplication1.Controllers.MDrivenWecpofController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Eco.MVC;
using EcoProject1.EcoSpaceAndModel1;
using EcoProject1.EcoSpaceAndModel1.ViewModelCodeGen_Class1List;

namespace MvcApplication1.Controllers
{
    public class MDrivenWecpofController : ModelDrivenControllerBase<EcoProject1EcoSpace>
    {
      public ActionResult Index()
      {
        return View("GenericView", Class1List.Create(EcoSpace, null));
      }

    }
}
```

Now I add the view that wecpof will use, GenericView;

The GenericView use VMClass as model – VMClass is the superclass of all code generated ViewModels.

The View has two htmlhelpers – DisplayWecpofActions and DisplayWecpofUI:

```
@using Eco.ViewModel.Runtime
@using Eco.MVC
@model VMClass

@{
    ViewBag.Title = "GenericView";
}

<h2>@Html.Label(Model.GetType().Name)</h2>

@using (Html.BeginForm("Submit","MDrivenWecpof",FormMethod.Post))
{

    <fieldset>
        <legend></legend>

       <table border="0">
          <tr>
              <td style="vertical-align:top;">@Html.DisplayWecpofActions()</td>
              <td style="padding:10px;border-left: lightgray thin solid ;">@Html.DisplayWecpofUI()</td>
          </tr>
          <tr>
              <td ></td>
              <td><input type="submit" value="Submit" /></td>
          </tr>
      </table>
    @Html.ValidationSummary(true)


    </fieldset>
}
```

Run:

http://localhost:64361/MDrivenWecpof

MDrivenWecpof - My ASP....

Arkiv Redigera Visa Favoriter Verktyg Hjälp

Google Sök Mer Hans Karlsen

Sida Säkerhet Verktyg

your logo here

Register Log in

Home About Contact

MDrivenWecpof

Show Class1

All Instances

Attribute1

Created 2014-02-17 14:15:11 abc hh

Created 2014-02-17 14:16:19

Created 2014-02-17 14:17:32

Created in the VM x

Created in the VM Editing works?

Created in the VM sample 2014-02-18 22:24:09

Created 2014-02-18 23:18:05

Created in the VM sample 2014-02-18 23:18:29

Submit

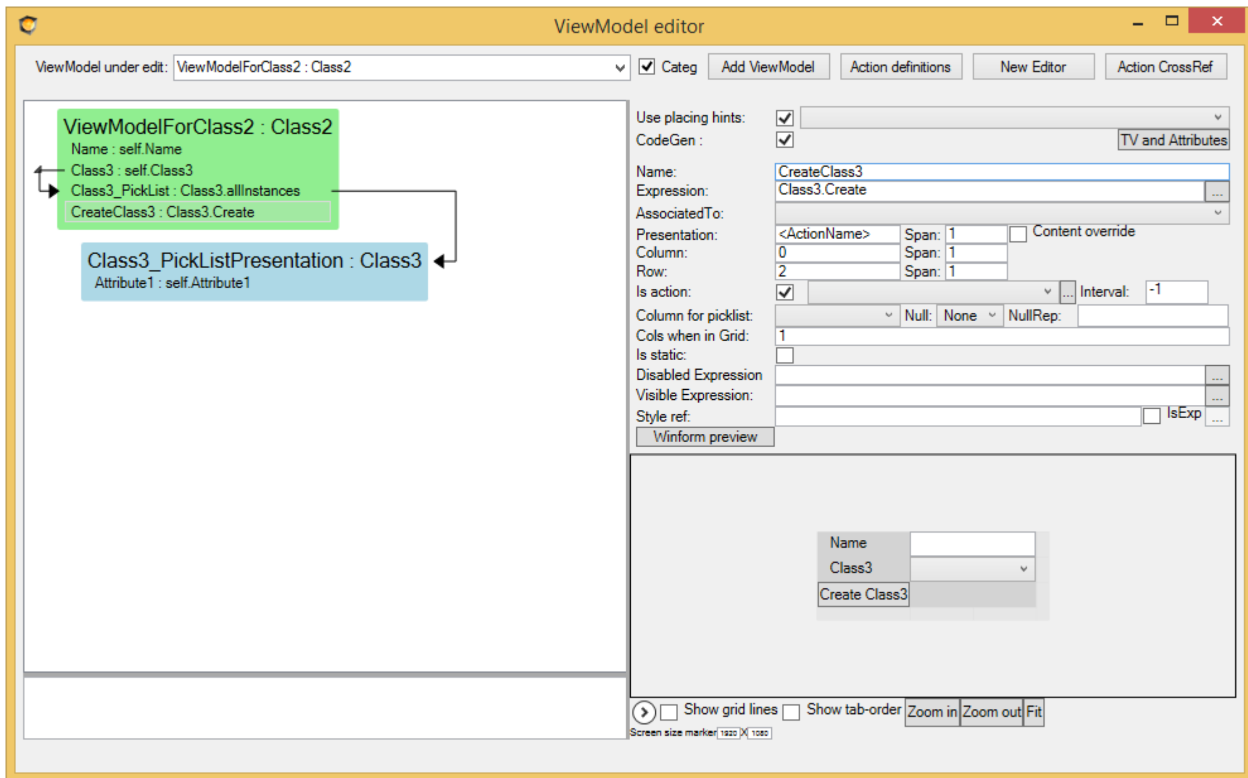
Create

Back to List

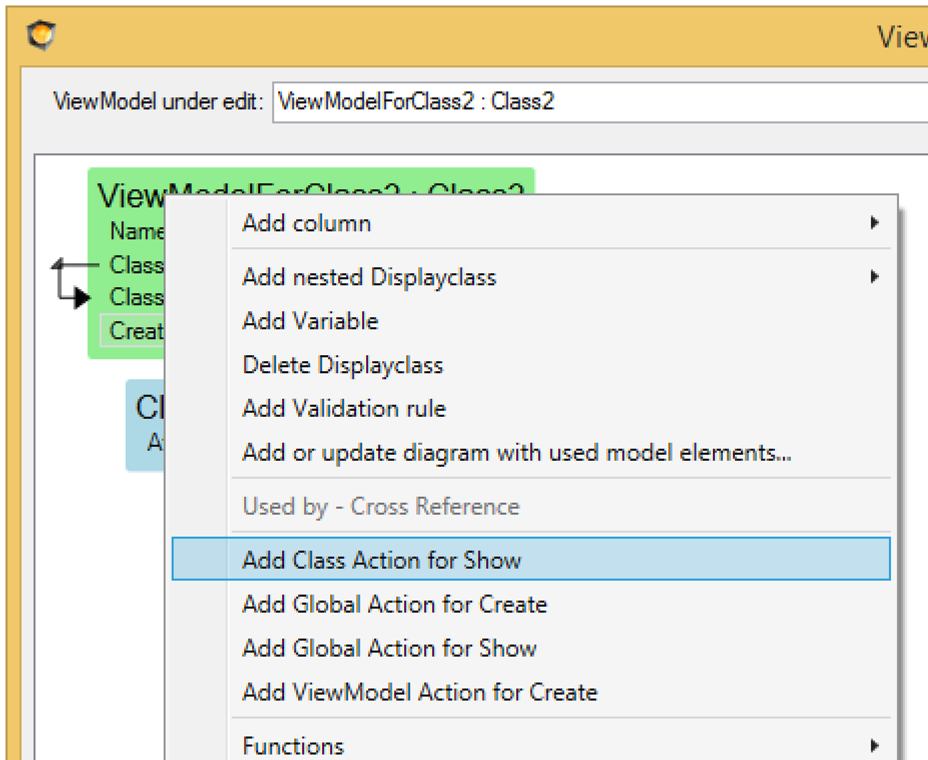
100 %

Clicking a row (in wecpof for MVC the first navigating action available on an action is used as the action to execute when clicking a grid row):

So let us add a viewmodel for class2 and an action to show it:



Making use of the shortcut to create an action:



The action is found here:

Actions-editor... ActionDefinitions

Show global actions ☑
Show Class actions ☑
Show ViewModel actions ☑ Filter Opt Out ☐
Filter
Filter on Class
Filter on ViewModel
Show All Actions

All Actions

Action	Action Type
----ShowClass1	On class Class1 shows ViewModelWithUIHints
----ViewViewModelForClass2	On class Class2 shows ViewModelForClass2
001-File---Exit	Main menu executes
001-File---Save	Main menu executes
001-File---Refresh	Main menu executes

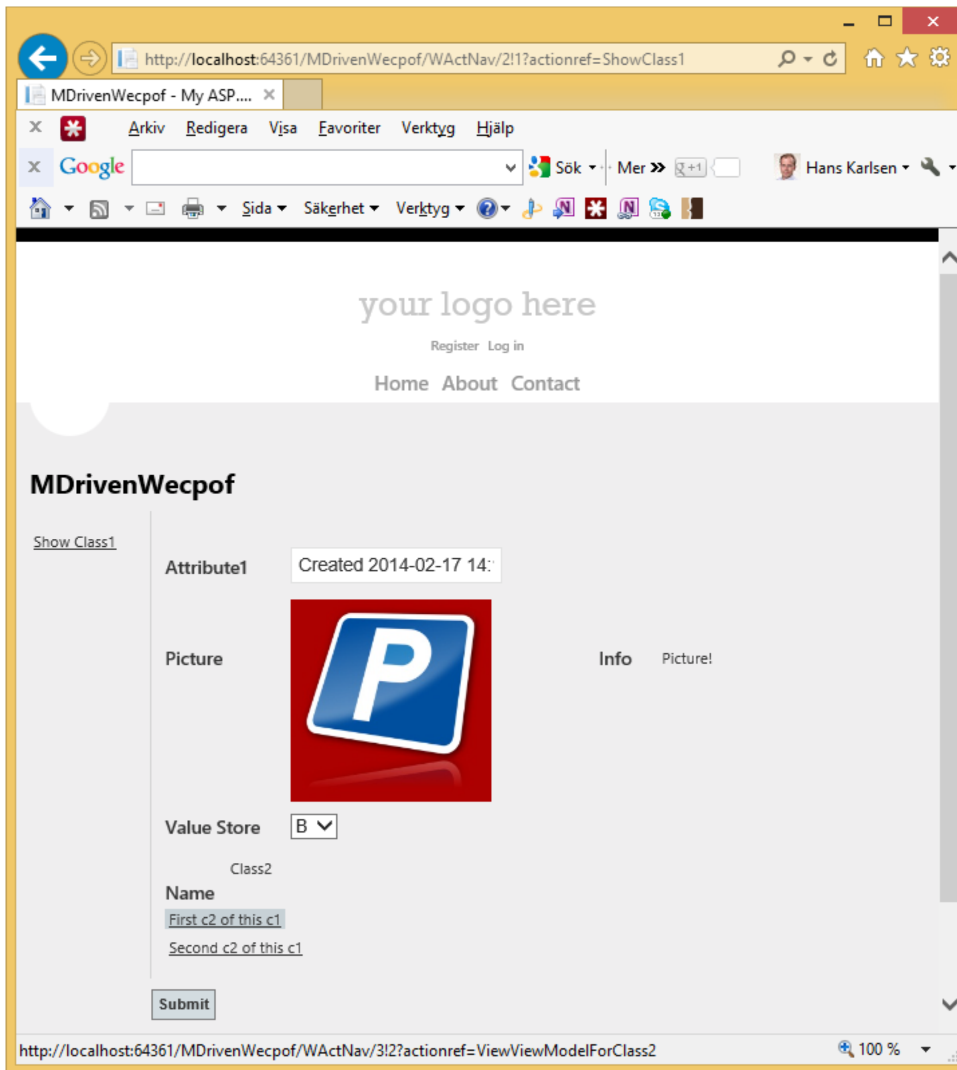
Name and Context

Name	ViewViewModelForClass2
Action only in ViewModel	
Only In This	
Action on Class	Class2

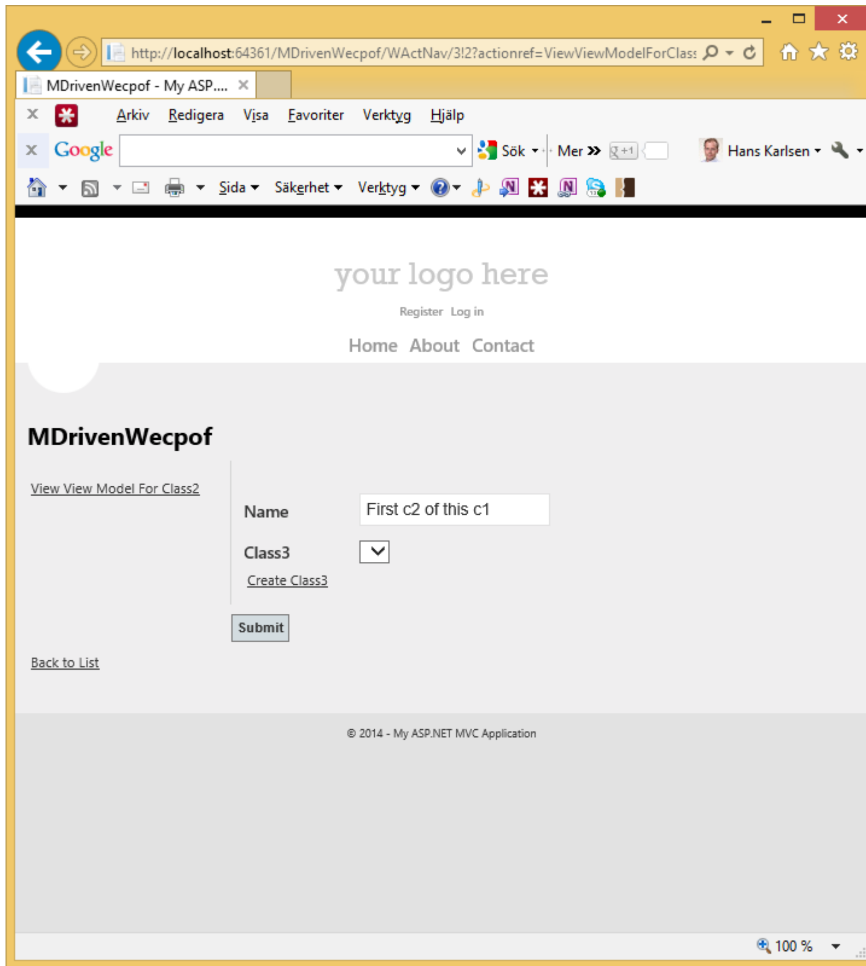
Action shows in 2 pl

Update code, run:

Clicking in on a Class1, notice the Class2's has been turned into links:

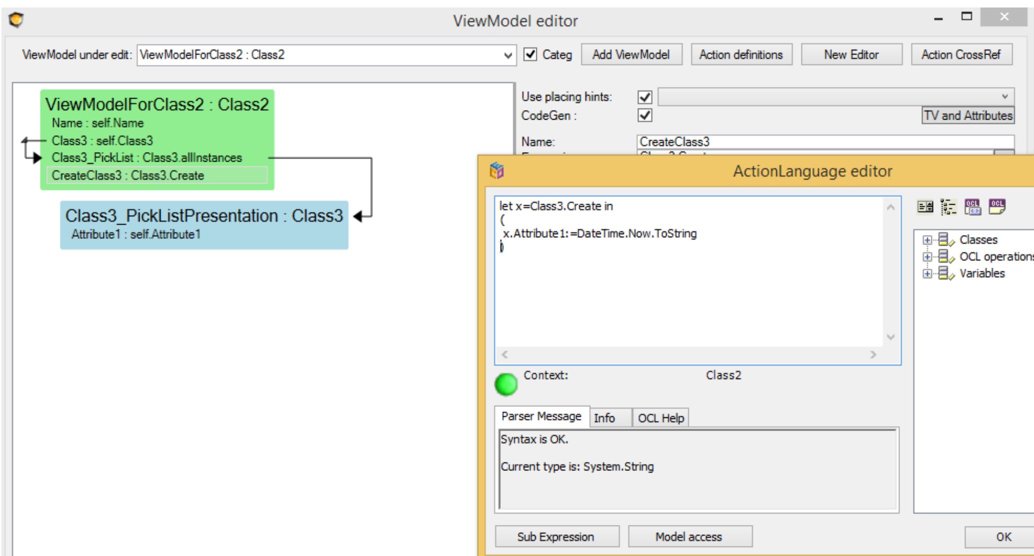


And voila:



Pressing the “Create Class3” link:

I can see that it creates a Class3 – but the Attribute1 of this class is not set by the action. Will fix:



Lets add a list of all class3 as well to the viewmodel:

ViewModel editor

ViewModel under edit: ViewModelForClass2 : Class2 | Categ | Add ViewModel | Action definitions | New Editor | Action CrossRef

ViewModelForClass2 : Class2
Name : self.Name
Class3 : self.Class3
Class3_PickList : Class3.allInstances
CreateClass3 : let x=Class3.Create in
(
x.Attribute1:=DateTime.Now.ToString
)
AllClass3 : Class3.AllInstances

Class3_PickListPresentation : Class3
Attribute1 : self.Attribute1

AllClass3 : Class3
Attribute1 : self.Attribute1

Use placing hints:
CodeGen :
TV and Attributes
Name: AllClass3
Expression: Class3.AllInstances
AssociatedTo: AllClass3
Presentation: <Name> Span: 1 Content override
Column: 0 Span: 2
Row: 3 Span: 5
Is action: Interval: -1
Column for picklist: Null: None NullRep:
Cols when in Grid: 1
Is static:
Readonly Expression
Visible Expression:
Style ref: IsExp
Winform preview

Name
Class3
Create Class3
All Class3
Attribut

Show grid lines | Show tab-order | Zoom in | Zoom out | Fit
Screen size marker 1920 X 1080

While I am at it – might as well add a viewmodel and an action to show Class3…:

ViewModel editor

ViewModel under edit: ShowClass3 : Class3 | Categ | Add ViewModel | Action definitions | New Editor | Action CrossRef

ShowClass3 : Class3
Attribute1 : self.Attribute1
UsedByTheseClass2 : self.Class2

Class2 : Class2
Name : self.Name

Use placing hints:
CodeGen :
Name: ShowClass3
Class: Class3
Column Count: 0
Row Count: 0
Column Width: 50
Row Height: 20
Taborder: DisplayOrderYBeforeX
Variables:
Validation rules:
Constraints:
OptOut Constraints:
Readonly Expression:
Requires root object:
Act As For Actions:
Winform preview

Attribute1
Used By These Class2
Name

Show grid lines | Show tab-order | Zoom in | Zoom out | Fit
Screen size marker 1920 X 1080

Update code, run:

ViewModelForClass2

View View Model For Class2

Name First c2 of this c1

Class3 2014-02-19 21:32:41

Create Class3

All Class3

Attribute1

Empty

Empty

Empty

Empty

2014-02-19 21:32:31

2014-02-19 21:32:41

2014-02-19 21:32:46

2014-02-19 21:45:33

2014-02-19 21:58:43

Submit

http://localhost:64361/MDrivenWecpof/WActNav/4!26?actionref=ViewShowClass3

The Create Class3 works ok… Click into one of the class3's:

ShowClass3

View Show Class3

Attribute1 2014-02-19 21:32:41

Used By These Class2

Name

First c2 of this c1

Submit

Back to List

© 2014 - My ASP.NET MVC Application

And that works as well… Notice that we have a list of the Class2's that use this Class3 – these are also links – because we defined a ClassAction on Class2 to show the ViewModelForClass2 – so we can navigate back… Easy. Fast. NeverFails.

Once I have wecpof set up this way I do not need to write more mundane code – I can define actions and viewmodels in the model and the MVC app will follow.

When you combine this with the advanced modelling abilities from CapableObjects – you can do a lot – fast – with high quality. And still you mix in custom jobs wherever you need in your app – it is just with CapableObjects you can finally spend 90% of the time on the 10% cool stuff. Challenged? I promise you: it is fun, fast, new – and we have just started… Join us!

Hans Karlsen M.sc, CTO