Lab 1 – Create a Basic ASP.NET MVC Application

Introduction

The goal of this lab is to provide an introduction to the ASP.NET MVC framework and create a simple application allowing the user to view and edit data. After completing these exercises you will have a better understanding of the Model View Controller (MVC) pattern and hopefully a greater appreciation for the things which it enables.

Model-View-Controller pattern according to Wikipedia.com:

> *Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The central component, the model, consists of application data, business rules, logic and functions. A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the controller, accepts input and converts it to commands for the model or view.*
>
> Source: Wikipedia http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

Why Do People Care?

For many this may be the most important question to answer. Possibly more important than any specific feature that ASP.NET includes, the thing that may matter most is the ability it gives you to take full control over the things that matter to you most. Maybe it's fine grained controller over the HTML that is rendered, or an architecture that exhibits a clean separation of concerns.

The MVC pattern has grown in popularity because of some of the key benefits it provides including:

- Provides separation of concerns (SoC).
- Enables fine grained control over the content being rendered.
- Supports the statelessness of the web and HTTP Verbs (GET, POST, PUT, DELETE…)
- Integrates with common web CSS and JavaScript frameworks.
- Makes code easier to test. (Which also potentially means easier to maintain, change, understand)

For capabilities to matter these need to be things you are concerned with… and if these things are not on your radar (For whatever reason) then you may wish to move along, but if these matter lets dig in!

To complete this exercise you need to have Visual Studio 2013 installed on your system. A free version can be downloaded here: http://www.visualstudio.com/downloads/download-visual-studio-vs

For additional ASP.NET MVC resources please visit the HTTP://ASP.NET/MVC website and check out my blog HTTP://www.RobZelt.com/Blog where I will be posting follow up information.

Step 1 - Creating a new project

Open Visual Studio 2013 and create a new project by using the File-> New Project menu option or by clicking on new project on the start page. In Visual Studio 2013 the steps to create web applications have been simplified. We start by selecting an ASP.NET Web Application. Name the project “MvcDemo”.

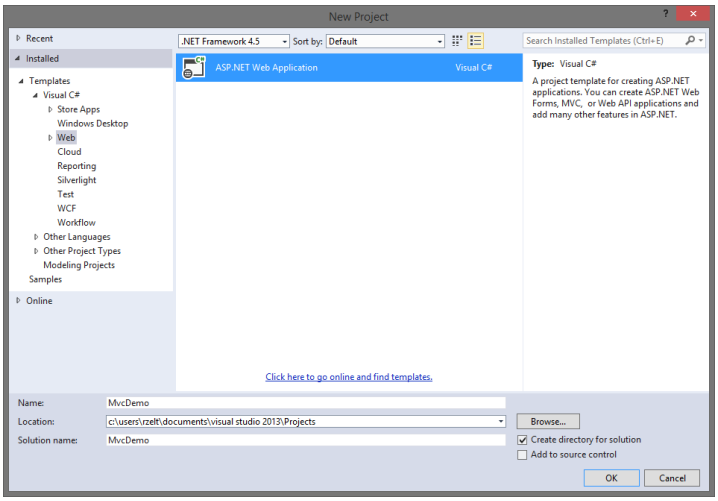

Figure 1

Select the Empty project template and check off MVC under the “Add folders and core references” section. The new web templates make it very easy to create One ASP.NET application that contains a combination of MVC, Web API, as well as Web Forms if desired.

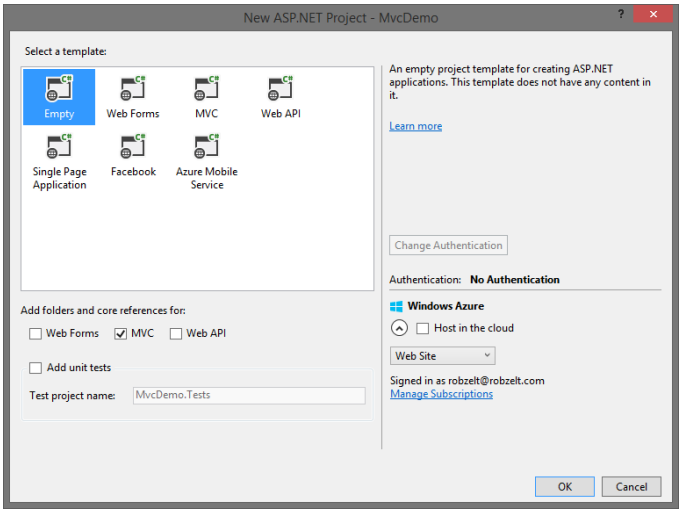

Figure 2

A new solution is created for us with our MvcDemo web project. The “Empty” template that we selected includes just the basic references and configuration needed to get us started.

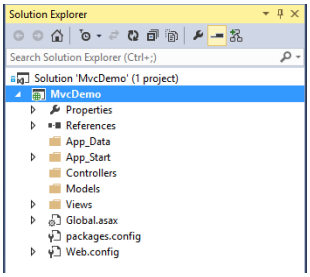

Figure 3

Step 2 - Adding the default controller HomeController

In order to begin handling http requests, we need to create a controller. By default ASP.Net MVC is configured to look for a controller called HomeController. To create this Right-Click on Controllers folder and select Add-> Controller from the menus.

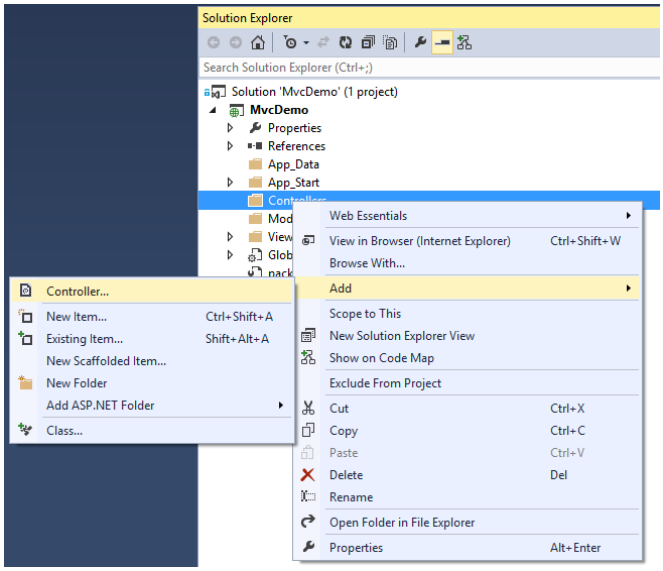

Figure 4

Visual Studio provides a number of default templates that can be used to create a new controller. Initially we are going to select MVC 5 Controller – Empty. We will later take a look at some of the others as we move along.

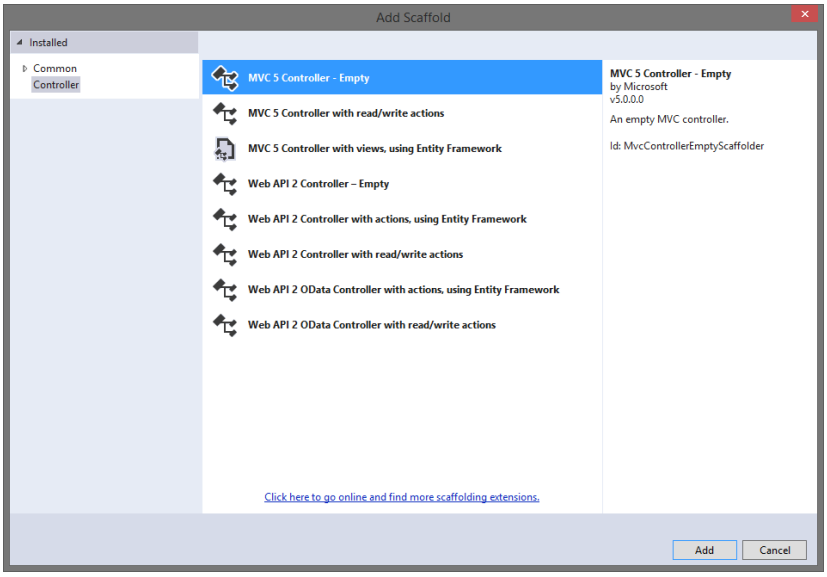

Figure 5

Name the controller HomeController so that it can be properly discovered.

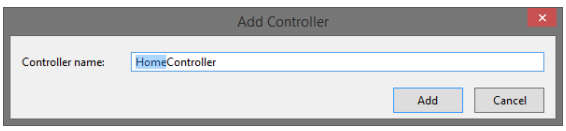

Figure 6

Once the controller is created it will contain a single method called index which will be execeuted when a http get request is made to either “/” or “/Home/Index”. If we try to view this in a web browser right now we will see the following. (Select Debug->Start Debugging from the menu or by pressing F5)

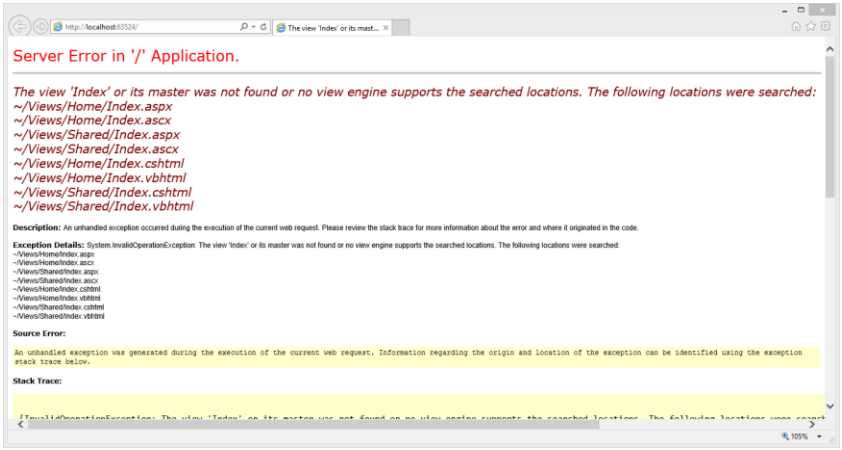

Figure 7

We see an error because no view hase been defined in our application for this request. The error message shows the default locations that ASP.Net looks in to find a vew file. Let’s go ahead and create an Index view for this controller action by right clicking on the “View()” portion of “return View()” in the index result.

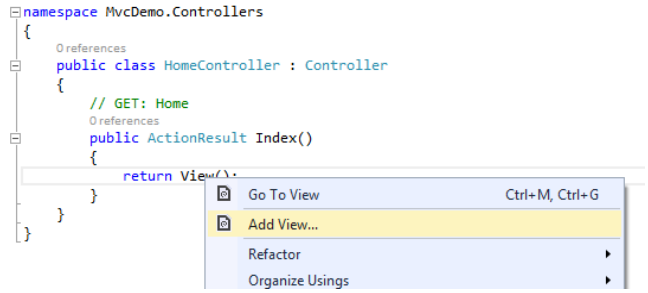

Figure 8

Visual Studio will provide us with the Add View wizard and allow us to select a template and select a number of options. To get started create a view named “Index” using the “Empty (without model)” template selecting, leaving all of the other options at their defaults.

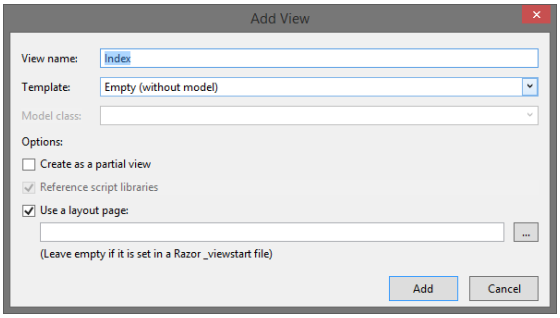

Figure 9

Once this completes you will see a new “Home” folder created under the Views which contains the new view Index.cshtml.

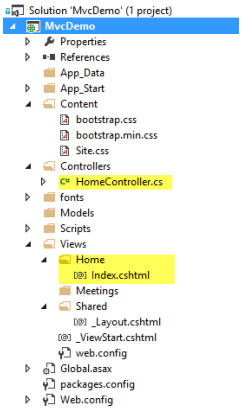

Figure 10

Hit F5 to build and run the project

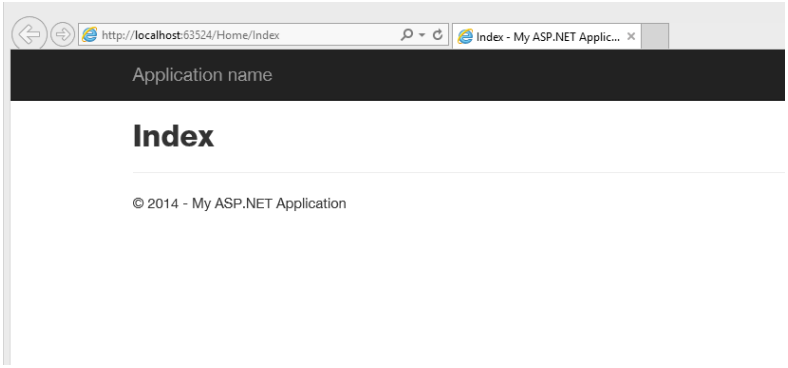

Figure 11

Feel free to take a look at the Index.cshtml file to see the html that it contains.

Step 3 - Adding the MeetingsController and returning a list of data

To begin working with some data we need to create another controller that we will use. Once again right-click on the Controllers folder and select “Add-> Controller”

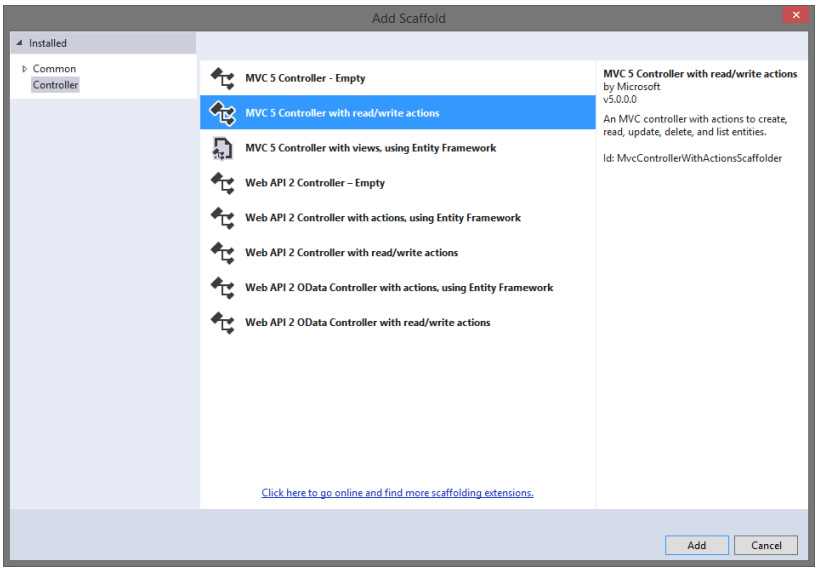

Figure 12

This time we will create an “MVC 5 controller with read/write actions” and name it “MeetingsController”

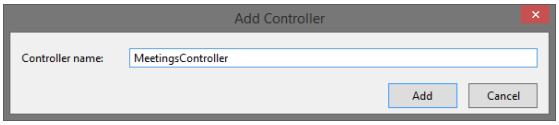

Figure 13

You will notice that the code contained in this controller is considerably more than in our previous example. It includes the same default Index method, but also Details, Create, Edit, and Delete methods,

Figure 14

In order to proceed to the next steps, we need to create a basic model to go along with our controller and views. Right-click on the models folder and select Add-> Class from the menu.

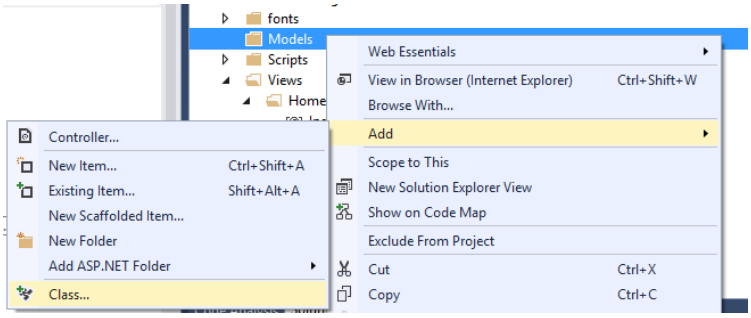

Figure 15

Give the class a filename of Meeting.cs.

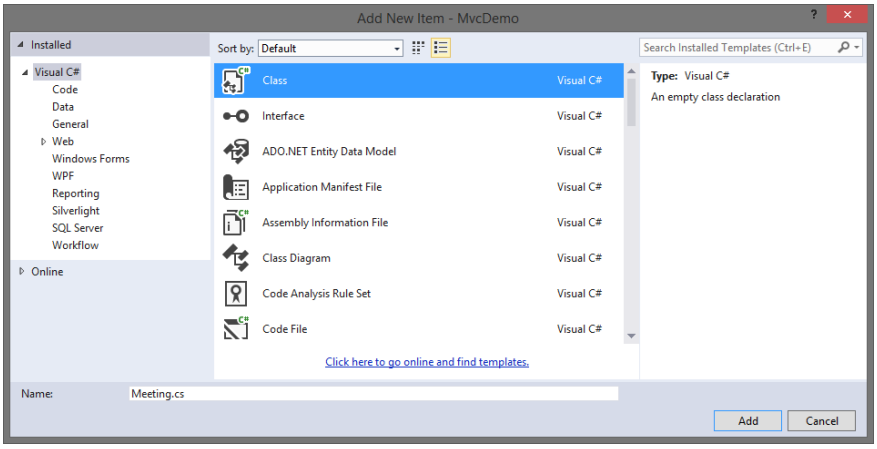

Figure 16

Add the following code to the class to create a basic meeting model for us to use.

```
public class Meeting
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime MeetingDate { get; set; }
}
```

In order to simplify data access for this lab we have created a simple in memory repository called MockMeetingRepository.cs. In Fowler's PoEAA, the Repository pattern is described as "Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects." In simple terms, it gives us a simple way to load data without needing to worry about the underlying data access technologies at this time.

You can download the file from http://robzelt.com/MockMeetingRepository.txt and add it into the models folder and rename to "MockMeetingRepository.cs". Once you have added it into the models folder of your project add the following code within the MeetingsController class to instantiate the repository as needed.

```
private MockMeetingRepository meetingRepository;


 public MeetingsController()
 {
     meetingRepository = new MockMeetingRepository();
 }
```

Then in the Index method add the code shown here too load a list of meetings and pass the model through to the view.

```
public ActionResult Index()
 {
     List<Meeting> meetings = meetingRepository.GetAll();
     return View(meetings);
 }
```

And finally lets add a new view using the List template by once again right clicking on the Return View(meetings) code.

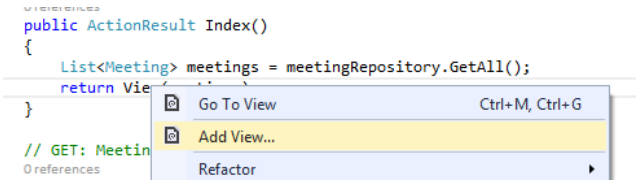

Figure 17

This time we want to select the “List” Template as well as specifying our Meeting class as the Model Class.

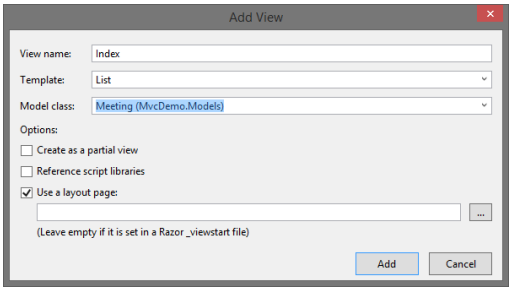

Figure 18

Once done, go ahead and run the app again, (F5 or Debug-> Start debugging)

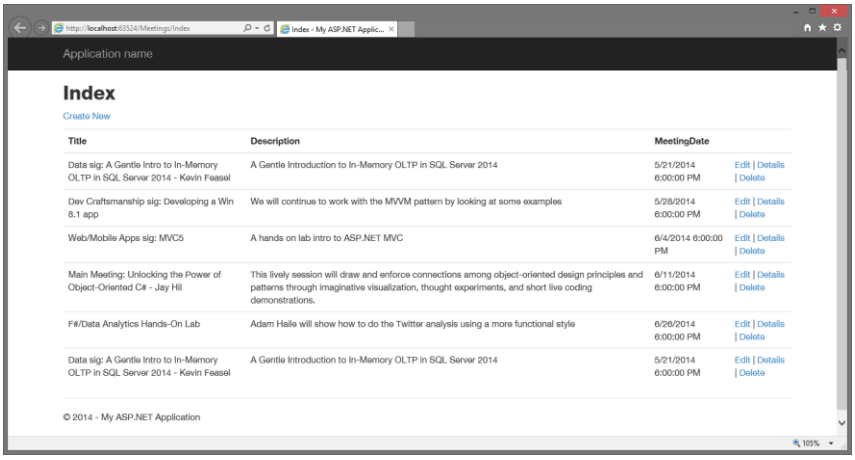

Figure 19

Take a closer look at the view that was generated… You will see that it displays the data as well as links to other pages.

Step 4 - Adding a details view

Now let’s add the functionality for the Details view…

```
public ActionResult Details(int id)
 {
     Meeting meeting = meetingRepository.Get(id);
     return View(meeting);
 }
```

After that create the view by right clicking….

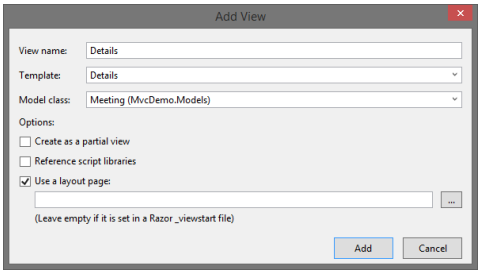

Figure 180

Run the application once again and select the Details Link from the listing page.

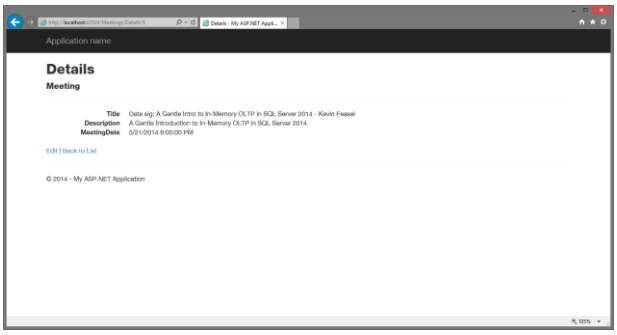

Figure 21

Let's now take a look at Edit. You will notice that there are two Edit methods in the controller, the second of which is decorated with an [HttpPost]attribute.

In the first method we need to populate the model with a meeting instance and pass it through to the view

```
public ActionResult Edit(int id)
 {
     Meeting meeting = meetingRepository.Get(id);
     return View(meeting);
 }
```

After that is in place, go ahead and create an Edit view by selecting the Edit template and once again specifying Meeting as the model class.

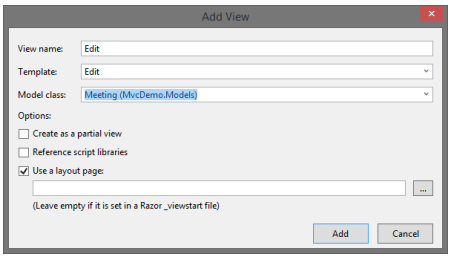

Figure 22

This view template provides us with a form and appropriate fields to edit the meeting model

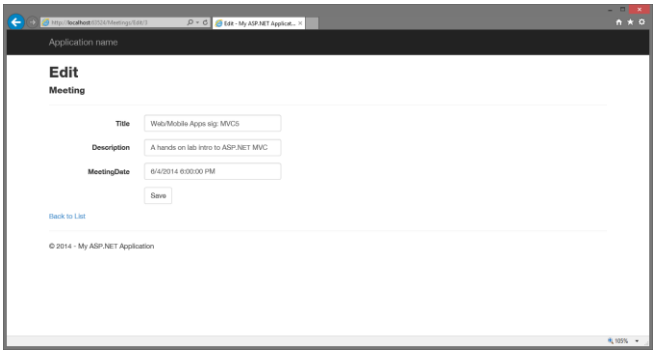

Figure 23

To handle the updated data, we first need to deal with the post method that will be called. By Default the controller action created by the template is looking for an int and a FormCollection object.

```
// POST: Meetings/Edit/5
[HttpPost]
0 references
public ActionResult Edit(int id, FormCollection collection)
{
```

However we want to take advantage of model binding…

Model Binders

A model binder in MVC provides a simple way to map posted form values to a .NET Framework type and pass the type to an action method as a parameter. Binders also give you control over the deserialization of types that are passed to action methods. Model binders are like type converters, because they can convert HTTP requests into objects that are passed to an action method. However, they also have information about the current controller context.

A model binder lets you associate a class that implements the IModelBinder interface using an action-method parameter or using a type. The IModelBinder interface contains a GetValue method that the framework calls in order to retrieve the value of a specified parameter or type. The DefaultModelBinder class works with most .NET Framework types, including arrays and IList, ICollection, and IDictionary objects.

Source: MSDN http://msdn.microsoft.com/en-us/library/dd410405(v=vs.100).aspx

Change the action to accept a Meeting object instead of the FormCollection:

```
// POST: Meetings/Edit/5
 [HttpPost]
 public ActionResult Edit(int id, Meeting meeting)
 {
     try
     {
         meetingRepository.Update(meeting);
         return RedirectToAction("Index");
     }
     catch
     {
         return View(meeting);
     }
 }
```

Now let's enable the user to create a new entry. Again we'll start by created a view using the appropriate "Create" template

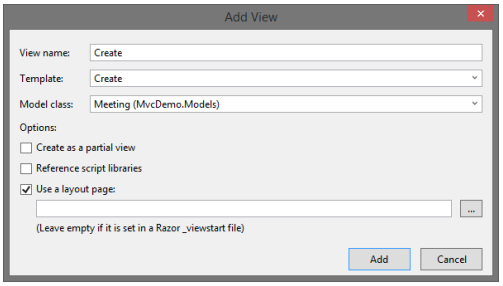

Figure 24

This provides us with an empty form to display to the user. The next step is to again deal with the post containing the data.

```
// POST: Meetings/Create
[HttpPost]
0 references
public ActionResult Create(FormCollection collection)
{
    try
    {
        // TODO: Add insert logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

Figure 195

One again the HttpPost method contains a FormCollection which we want to change to a Meeting object.

```
// POST: Meetings/Create
        [HttpPost]
        public ActionResult Create(Meeting meeting)
        {
            try
            {
                meetingRepository.Add(meeting);
                return RedirectToAction("Index");
            }
            catch
            {
                return View(meeting);
            }
        }
```

With that in place the last remaining item is to deal with the delete. Start by right clicking on the return View() code in the Delete method and create a view using the Delete template.

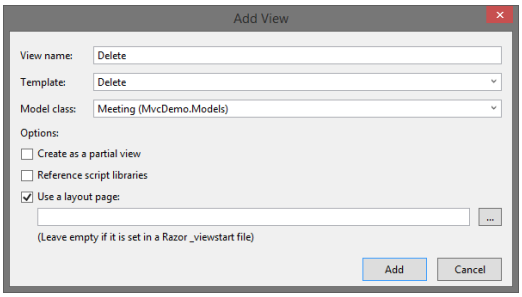

Figure 26

And update the code in the two delete methods as shown

```
        // GET: Meetings/Delete/5
        public ActionResult Delete(int id)
        {
            Meeting meeting = meetingRepository.Get(id);
            return View();
        }

        // POST: Meetings/Delete/5
        [HttpPost]
        public ActionResult Delete(int id, Meeting meeting)
        {
            try
            {
                meetingRepository.Remove(id);
                return RedirectToAction("Index");
            }
            catch
            {
                return View(meeting);
            }
        }
```

Congratulations! You should now have a fully functioning ASP.NET application that allows you to create, read, update, and delete data. Take some time to get familiar with things and learn our way around. We will use this base application and extend it in future labs.

The completed files for this lab are available at https://github.com/robzelt/IntroMvcDemo . Please send any comments or feedback to me at robzelt@robzelt.com.