MVC :: Understanding Models, Views, and Controllers

This tutorial provides you with a high-level overview of ASP.NET MVC models, views, and controllers. In other words, it explains the 'M', 'V', and 'C' in ASP.NET MVC.

After reading this tutorial, you should understand how the different parts of an ASP.NET MVC application work together. You should also understand how the architecture of an ASP.NET MVC application differs from an ASP.NET Web Forms application or Active Server Pages application.

The Sample ASP.NET MVC Application

The default Visual Studio template for creating ASP.NET MVC Web Applications includes an extremely simple sample application that can be used to understand the different parts of an ASP.NET MVC application. We take advantage of this simple application in this tutorial.

You create a new ASP.NET MVC application with the MVC template by launching Visual Studio 2008 and selecting the menu option File, New Project (see Figure 1). In the New Project dialog, select your favorite programming language under Project Types (Visual Basic or C#) and select **ASP.NET MVC Web Application** under Templates. Click the OK button.

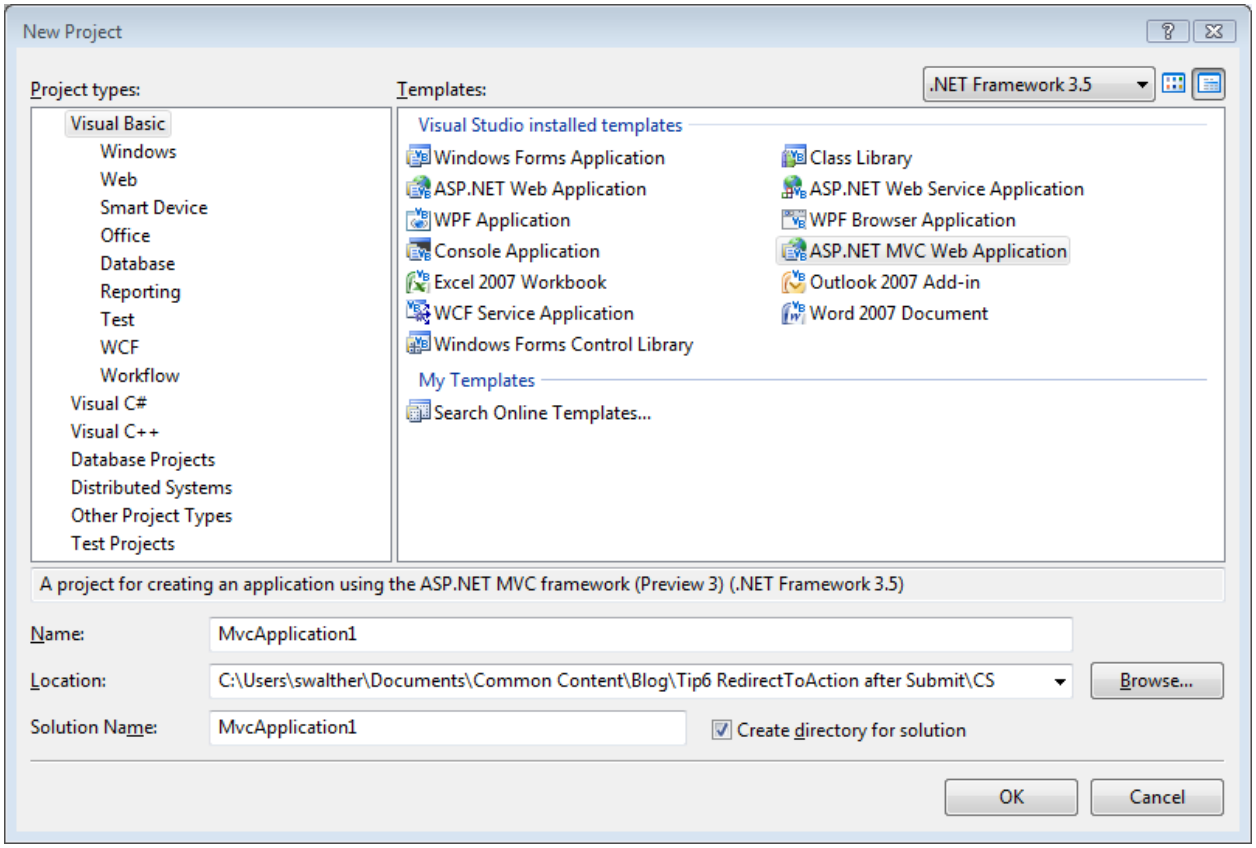

Figure 1 – New Project Dialog

When you create a new ASP.NET MVC application, the Create Unit Test Project dialog appears (see Figure 2). This dialog enables you to create a separate project in your solution for testing your ASP.NET MVC application. Select the option **No, do not create a unit test project** and click the **OK** button.

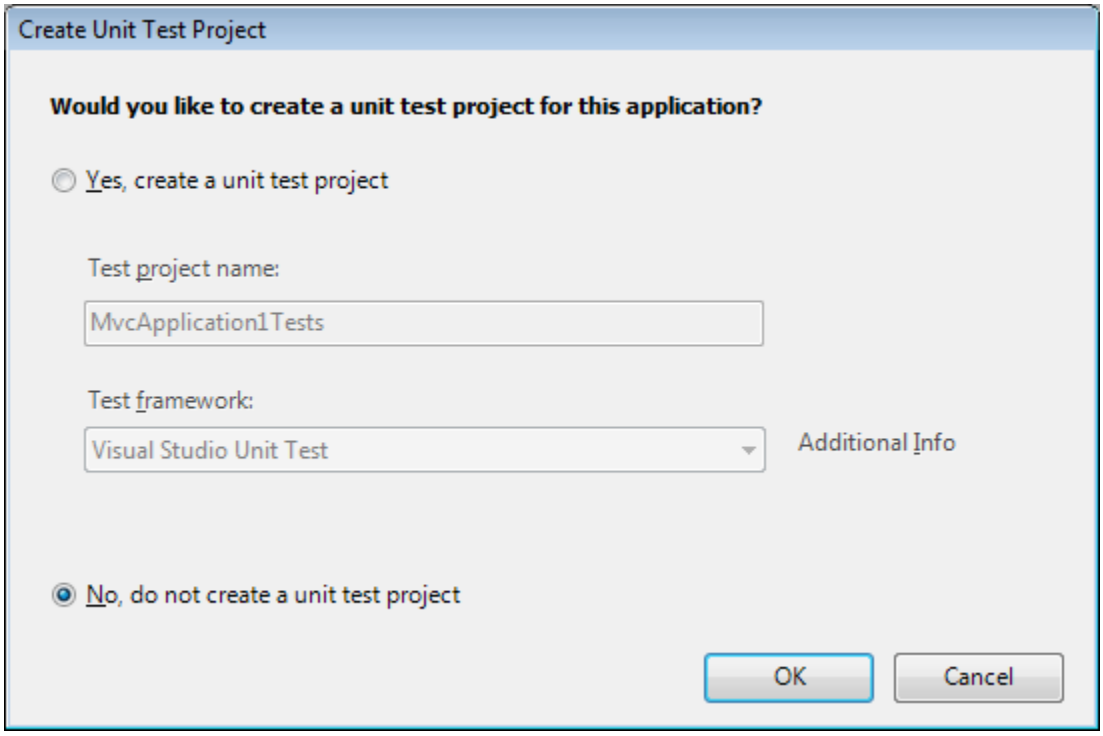

Figure 2 – Create Unit Test Dialog

After the new ASP.NET MVC application is created. You will see several folders and files in the Solution Explorer window. In particular, you'll see three folders named Models, Views, and Controllers. As you might guess from the folder names, these folders contain the files for implementing models, views, and controllers.

If you expand the Controllers folder, you should see a file named HomeController.cs. If you expand the Views folder, you should see two subfolders named Home and Shared. If you expand the Home folder, you'll see two additional files named `About.aspx` and `Home.aspx` (see Figure 3). These files make up the sample application included with the default ASP.NET MVC template.

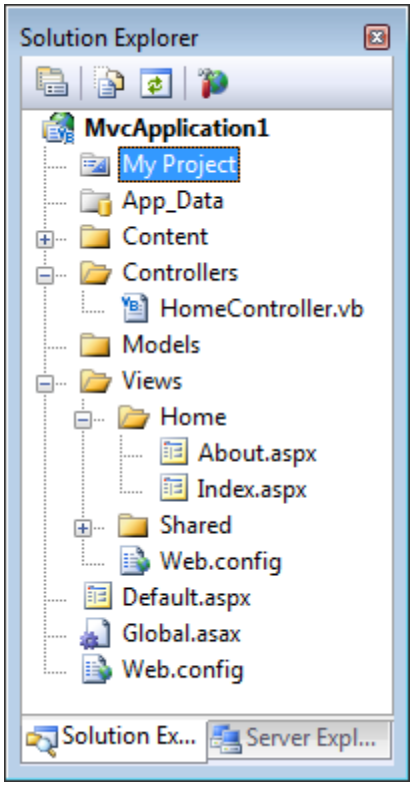

Figure 3 – The Solution Explorer Window

You can run the sample application by selecting the menu option **Debug, Start Debugging**. Alternatively, you can press the F5 key.

When you first run an ASP.NET application, the dialog in Figure 4 appears that recommends that you enable debug mode. Click the OK button and the application will run.

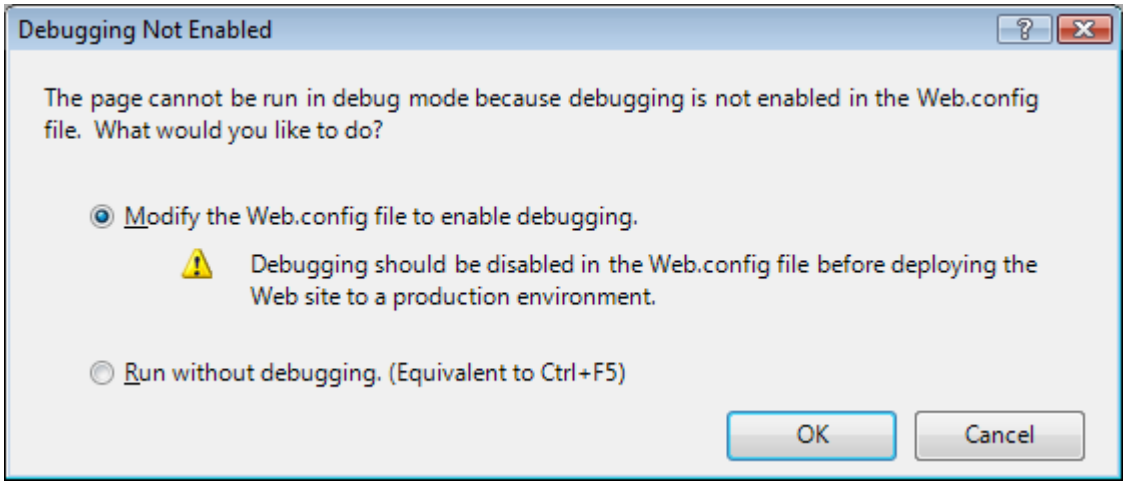

Figure 4 – Debugging Not Enabled dialog

When you run an ASP.NET MVC application, Visual Studio launches the application in your web browser. The sample application consists of only two pages: the Index page and the About page. When the application first starts, the Index page appears (see Figure 5). You can navigate to the About page by clicking the menu link at the top right of the application.

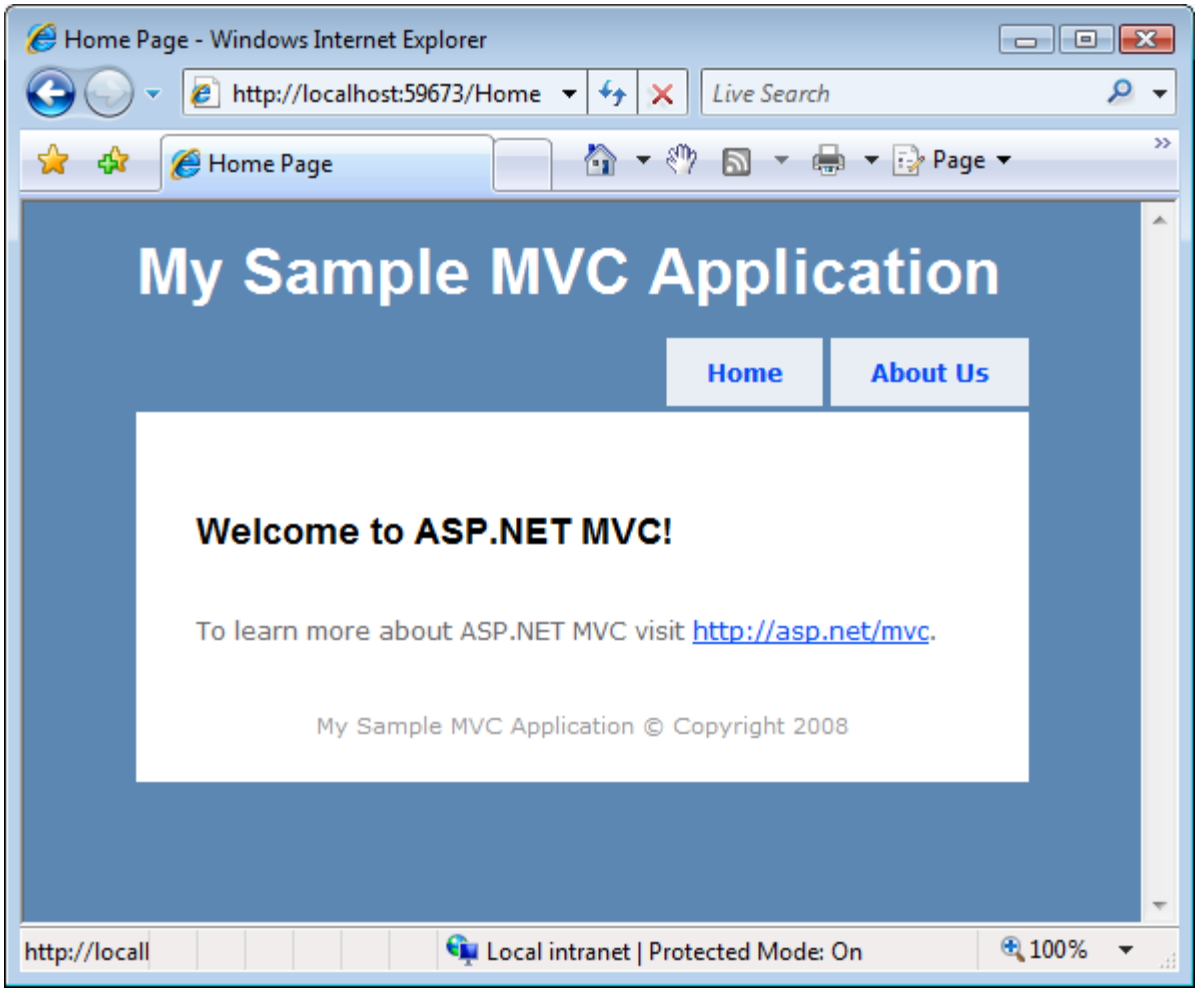

Figure 5 – The Index Page

Notice the URLs in the address bar of your browser. When you click the Home menu link, the URL in the browser address bar changes to **/Home**. When you click the About menu link, the URL in the browser address bar changes to **/About**.

If you close the browser window and return to Visual Studio, you won't be able to find a file named Home or a file named About. The files don't exist. How is this possible?

A URL Does Not Equal a Page

When you build a traditional ASP.NET Web Forms application or an Active Server Pages application, there is a one-to-one correspondence between a URL and a page. If you request a page named SomePage.aspx from the server, then there had better be a page on disk named SomePage.aspx. If the SomePage.aspx file does not exist, you get an ugly **404 – Page Not Found** error.

When building an ASP.NET MVC application, in contrast, there is no correspondence between the URL that you type into your browser's address bar and the files that you find in your application. In an ASP.NET MVC application, a URL corresponds to a controller action instead of a page on disk.

In a traditional ASP.NET or ASP application, browser requests are mapped to pages. In an ASP.NET MVC application, browser requests are mapped to controller actions. An ASP.NET Web Forms application is content-centric. An ASP.NET MVC application, in contrast, is application logic centric.

Understanding URL Routing

A browser request gets mapped to a controller action through a feature of ASP.NET MVC called *URL Routing*. URL Routing *routes* incoming requests to controller actions.

URL Routing uses a route table to handle incoming requests. This route table is created when your web application first starts. The route table is setup in the `Global.asax` file. The default MVC `Global.asax` file is contained in Listing 1.

Listing 1 – `Global.asax`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcApplication1
{
    public class GlobalApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index", id = ""
   }
            );
```

```
        }


        protected void Application_Start()
        {
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

When an ASP.NET application first starts, the `Application_Start()` method is called. In Listing 1, this method calls the `RegisterRoutes()` method and the `RegisterRoutes()` method creates the default route table.

The default route table consists of one route. This default route breaks all incoming requests into three segments (a URL segment is anything between forward slashes). The first segment is mapped to a controller name, the second segment is mapped to an action name, and the final segment is mapped to a parameter passed to the action named Id.

For example, consider the following URL:

/Product/Details/3

This URL is parsed into three parts like this:

Controller = ProductController

Action = Details

Id = 3

Notice that the suffix Controller is tacked on to the end of the Controller parameter. This is just a quirk of MVC.

The default route includes default values for all three segments. The default Controller is `HomeController`, the default Action is Index, and the default Id is an empty string. With these defaults in mind, consider how the following URL is parsed:

/Employee

This URL is parsed into three parameters like this:

Controller = EmployeeController

Action = Index

Id = “”

Finally, if you open an ASP.NET MVC Application without supplying any URL (for example, http://localhost) then the URL is parsed like this:

Controller = HomeController

Action = Index

Id = “”

The request is routed to the `Index()` action on the `HomeController` class.

Understanding Controllers

A controller is responsible for controlling the way that a user interacts with an MVC application. A controller determines what response to send back to a user when a user makes a browser request.

A controller is just a class (for example, a Visual Basic or C# class). The sample ASP.NET MVC application includes ones controller named `HomeController.cs` located in the Controllers folder. The contents of the `HomeController.cs` is reproduced in Listing 2.

Listing 2 – `HomeController.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcApplication1.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Title"] = "Home Page";
            ViewData["Message"] = "Welcome to ASP.NET MVC!";

            return View();
        }

        public ActionResult About()
        {
            ViewData["Title"] = "About Page";

            return View();
        }
    }
}
```

Notice that the HomeController has two methods named `Index()` and `About()`. These two methods correspond to the two actions exposed by the controller. The URL /Home/Index invokes the `HomeController.Index()` method and the URL /Home/About invokes the `HomeController.About()` method.

Any public method in a controller is exposed as a controller action. You need to be careful about this. This means that any public method contained in a controller can be invoked by anyone with access to the Internet by entering the right URL into a browser.

Understanding Views

The two controller actions exposed by the `HomeController` class, `Index()` and `About()`, both return a view. A view contains the HTML markup and content that is sent to the browser. A view is the equivalent of a page when working with an ASP.NET MVC application.

You must create your views in the right location. The `HomeController.Index()` action returns a view located at the following path:

\Views\Home\Index.aspx

The `HomeController.About()` action returns a view located at the following path:

\Views\Home\About.aspx

In general, if you want to return a view for a controller action, then you need to create a subfolder in the Views folder with the same name as your controller. Within the subfolder, you must create an .aspx file with the same name as the controller action.

The file in Listing 3 contains the `About.aspx` view.

Listing 3 – `About.aspx`

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
   AutoEventWireup="true" CodeBehind="About.aspx.cs"
   Inherits="MvcApplication1.Views.Home.About" %>


<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent"
   runat="server">

    <h2>About Us</h2>

    <p>

        TODO: Put <em>about</em> content here.

    </p>

</asp:Content>
```

If you ignore the first line in Listing 3, most of the rest of the view consists of standard HTML. You can modify the contents of the view by entering any HTML that you want here.

A view is very similar to a page in Active Server Pages or ASP.NET Web Forms. A view can contain HTML content and scripts. You can write the scripts in your favorite .NET

programming language (for example, C# or Visual Basic .NET). You use scripts to display dynamic content such as database data.

Understanding Models

We have discussed controllers and we have discussed views. The last topic that we need to discuss is models. What is an MVC model?

An MVC model contains all of your application logic that is not contained in a view or a controller. The model should contain all of your application business logic and database access logic. For example, if you are using LINQ to SQL to access your database, then you would create your LINQ to SQL classes (your dbml file) in the Models folder.

A view should contain only logic related to generating the user interface. A controller should only contain the bare minimum of logic required to return the right view or redirect the user to another action. Everything else should be contained in the model.

In general, you should strive for fat models and skinny controllers. Your controller methods should contain only a few lines of code. If a controller action gets too fat, then you should consider moving the logic out to a new class in the Models folder.

Summary

This tutorial provided you with a high level overview of the different parts of an ASP.NET MVC web application. You learned how URL Routing maps incoming browser requests to particular controller actions. You learned how controllers orchestrate how views are returned to the browser. Finally, you learned how models contain application business and database access logic.