1

High-Speed Beginner Ramp Up

This chapter covers

* Explanation of the MVC pattern
* Dissecting the default application template
* Creating your first ASP.NET MVC 2 Project
* Creating Controllers & Actions
* Handling User Input
* Working with the View
* Simple View Models

This chapter is intended to provide you with a quick, high level overview of the framework. We'll create a very basic sample application, collect user input, and display some web pages. But first, let me introduce you to your new friend…

1.1 Welcome to ASP.NET MVC

ASP.NET MVC is a new web application framework from Microsoft. It was first unveiled in November of 2007 and has since seen more than 10 releases and 2 major versions. With the high number of releases, this framework has received quite a bit of feedback and is much more stable than some other new frameworks from Microsoft such as Workflow Foundation. The name stands for Model-View-Controller, a pattern that is becoming increasingly popular with web development frameworks.

ASP.NET MVC is both an alternative and compliment to Web Forms, which means you won't be dealing with Pages and Controls, postback or ViewState, or any complicated event lifecycle. Instead, you'll be defining controller, actions, and views. The underlying ASP.NET platform is the same, however, so things like HttpHandlers & HttpModules still apply, and you can mix MVC and Web Forms pages in the same application.

We'll cover all of the major features of the framework throughout this book, but quickly some of the benefits are:

* Full Control over HTML
* Full Control over URLs
* Better Separation of Concerns
* Extensibility
* Testability

As you read the chapters in this book, these items will become increasingly apparent. For now, we will briefly look at the underlying pattern the framework is based on. Why MVC? Where did it come from?

1.2 The MVC Pattern

The MVC pattern, or Model-View-Controller, is an adaptation of a pattern generated from the Smalltalk community in the 1970s by Trygve Reenskaug. It was popularized for use on the web with the advent of Ruby on Rails in 2003.

The components of MVC are very straightforward:

The Model - the "thing" that your software is built around. If you were building a blog, your models might be Post and Comment. In some contexts, this might simply refer to a view-specific model, which you'll learn about in the next chapter.

The View - a visual representation of a model, given some context. It's usually the resulting HTML that the framework renders to the browser. For example the HTML representing the blog post.

The Controller - a mediator. The controller processes input, acts upon the model, and decides what to do, render a view, redirect somewhere else, etc. The controller might pull the most recent comments for a blog post and send them to a view.

To see how these components interact with each other, take a look at Figure 1.1.

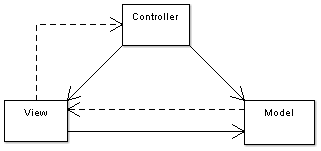


Figure 1.1 - A simple diagram depicting the relationship between the Model, View, and Controller. The solid lines indicate a direct association, and the dashed lines indicate an indirect association (graphic and description used with permission from Wikipedia).

Now that you have a rudimentary overview of the ASP.NET MVC Framework and the MVC pattern in general, you are armed to create your first project.

1.3 Creating Your First ASP.NET MVC 2 Project

The rest of this book assumes that you have ASP.NET MVC 2 installed, either on Visual Studio 2008 or on Visual Studio 2010. If you do not, check the Appendix for instructions on how to obtain it.

Fire up Visual Studio, and go to File 🡪 New Project. We will create a web application with some guestbook features. After selecting New Project, you are presented with the dialog pictured in Figure 1.2.

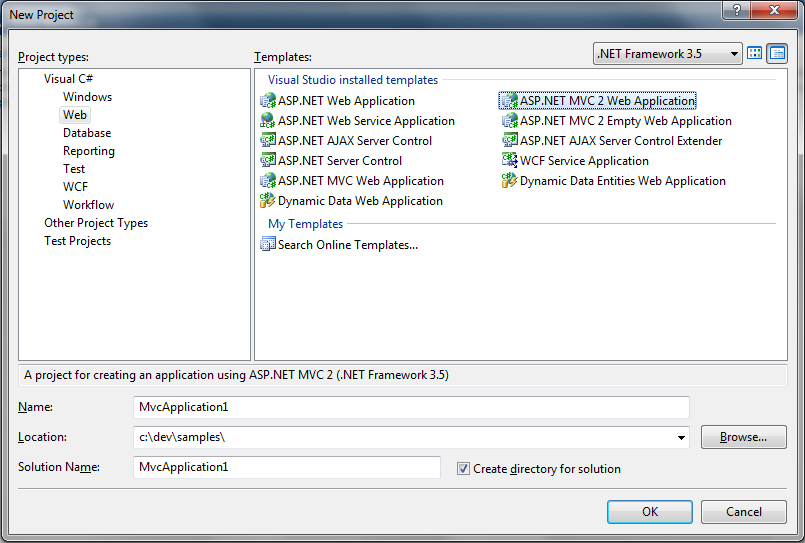


Figure 1.2 - The New Project Dialog. Notice the ASP.NET MVC 2 Project Templates

On the left pane, under Project types, select Web. For the Template, select ASP.NET MVC 2 Web Application. Give the application a name and location and click OK.

You are greeted with a dialog box (Figure 1.3) that asks you if you want to create a Unit Test project. Normally we'd recommend creating a unit test project because most non-trivial projects need automated tests; however, to keep this chapter focused, we'll select No for now.

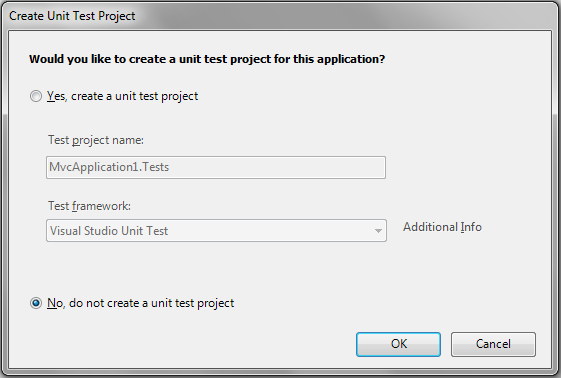


Figure 1.3 - Visual Studio prompts you to create a Unit Test project. For now, select No.

Your project is ready to go. Visual Studio created a number of folders for you. Let's examine them and see what their purpose is:

Content/ Static files such as CSS and images

Controllers/ Your application's controller classes

Models/ Your application's models

Scripts/ JavaScript files

Views/ Your application's views

Take a look at the folder structure for a minute. You'll work with this structure for all of your ASP.NET MVC projects, so everything will eventually look familiar.

The application that Visual Studio has given us is a working sample of the ASP.NET MVC Framework. That means we can just run it (CTRL-F5) and see how it works. Go ahead and do that now.

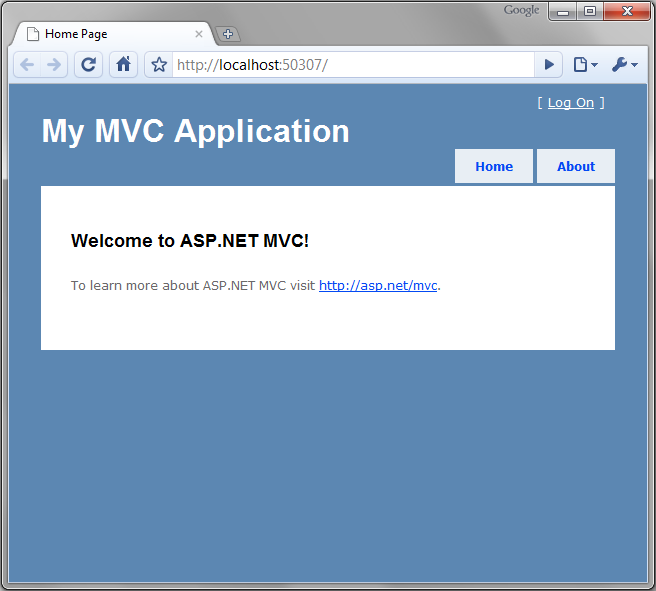


Figure 1.4 - The default ASP.NET MVC project template is fully functional

Your browser should be opened and you should be looking at a page that looks like Figure 1.4. Notice that the URL is simply <http://localhost:(port)/>. There is no path specified. Let's examine how this view was rendered.

First, we made a request to / (the root of the site), so we must check the routes to see how the application responds to URLs. Routes are a way for you to customize the URLs that users use when interacting with your site. You'll learn about routing in depth in Chapter 16 but we'll cover what you need to know to get started.

Routes are (by default) defined in the Global.asax. Open up this file and you should see the code shown in Listing 1.1.

Listing 1.1 - Route definitions

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Default", (#1)

"{controller}/{action}/{id}", (#2)

new { controller = "Home", action = "Index",

id = UrlParameter.Optional } (#3)

);

}

#1 - Route name

#2 - URL parameters

#3 - Default values

(Cueballs in Listing and Text)

Notice that two entries are defined. The first is an IgnoreRoute, and that basically tells the framework to not worry about anything matching the path. In this case, it means don't try to process any paths containing the .axd file extension, such as Trace.axd. The second entry, MapRoute, is what defines how URLs are processed. This built-in route will suffice for a while, but later on you will want to add more routes to the application. Just like how previous versions of ASP.NET decided the URL for your based on the directory structure and the Web Form file name, like Default.aspx, ASP.NET MVC projects come with a default URL structure, and applications that don't require custom URL schemes will do just fine with the defaults.

Each route has a name (#1), a URL definition (#2), and optional default values (#3). Our first request for / doesn't have any of these URL pieces, so we look to the defaults. The default values are:

controller 🡪 "Home"

action 🡪 "Index"

id 🡪 (optional)

A Note About routing

The route with the template, {controller}/{action}/{id} is a generic one and can be used for many, many different web requests. Tokens are denoted by the inclusion of curly braces "{}", and the word enclosed in braces matches a value the MVC Framework understands. The most common values that we'll be interested in are controller and action. The controller route value is a special value that the System.Web.Mvc.MvcHandler class passes to the controller factory in order to instantiate a controller.. This is also the route we’ll be using for the rest of the chapter, so we’ll be content with a URL in the form of http://site.org/controllername/ actionname. The basic route handler is an instance of IRouteHandler named MvcRouteHandler. We have complete control and could provide our own implementation of IRouteHandler if we wished, but we’ll save that for a later chapter.

So we know now that the controller is "Home" and the action is "Index." Take a look in the Controllers folder and you will see a class called HomeController. By convention, all controller classes end with the word Controller. Open up this class and you'll see your first controller class (Listing 1.2).

Listing 1.2 - The HomeController

[HandleError]

public class HomeController : Controller #1

{

public ActionResult Index() #2

{

ViewData["Message"] = "Welcome to ASP.NET MVC!";

return View();

}

public ActionResult About()

{

return View();

}

}

Cueballs in text

So what defines a controller in ASP.NET MVC anyway? For a class to be considered a controller, it must:

* End with the word "Controller"
* Inherit from System.Web.Mvc.Controller (or implement IController) (#1)
* Contain public methods that return ActionResult (these are called actions) (#2)

We know that the Index action is going to be called. In this action method, we have these 2 statements:

ViewData["Message"] = "Welcome to ASP.NET MVC!";

return View();

The first adds a string into a dictionary called ViewData. This is one way of passing data over to the view. The second line returns the result of a method called View(). This is a helper method, defined in the Controller base class. It simply returns a new ViewResult object. ViewResult is just one of the many ActionResult derivatives that you can return from actions.

This ViewResult tells the framework to render a view. You have the option of providing a name for the view, however if you don't - like in our case - it will just use the same name of the action.

So where is this view located? We learned a few minutes ago that the default project structure contains a Views folder. By convention, views are located in a subfolder corresponding the controller name. The name of the action (again by convention) is the same as the name of the view.

Inside of the Views folder you'll find a folder for each controller in the application, along with a special one named Shared. Open up the Home folder (since we're dealing with HomeController, and open the Index.aspx file. It should look like Listing 1.4.

Listing 1.4 - The Index.aspx view

<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="indexTitle" ContentPlaceHolderID="TitleContent" runat="server">

Home Page

</asp:Content>

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

<h2><%= Html.Encode(ViewData["Message"]) %></h2> (#1)

<p>

To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.

</p>

</asp:Content>

#1 - Use viewdata from Controller.

This view uses a master page, which is similar to what you would see in an ASP.NET Web Forms project. If you're curious, you can find this in /Views/Shared/Site.Master, however for now we can just focus on the view.

Our view will simply render the data provided by the controller. It should not contain any complex logic. Keeping the view simple makes it easy to read and maintain, especially since we'll be mixing code with HTML. In Listing 1.4, you can see that we simply output a message inside a code block denoted by <%= %> tags.

To illustrate working with the ASP.NET MVC Framework we will add some guestbook features to this application. The first step is adding a new controller.

1.4 Creating Controllers & Actions

To add a new controller to our site, we can right click on the Controllers folder. Visual Studio presents us with a context menu containing an item "Add Controller". You can see this in Figure 1.5.

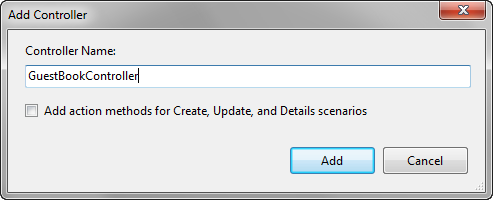


Figure 1.5 - The Add Controller dialog in Visual Studio

For the name, type "GuestBookController." For now, don't check the checkbox as we want to write our own actions. A class will be created for you that looks like Listing 1.5.

Listing 1.5 - Creating your first controller

public class GuestBookController : Controller

{

//

// GET: /GuestBook/

public ActionResult Index() #1

{

return View();

}

}

Cueballs in text

Notice that an initial action method Index is created for you (#1). For this action, we don't need to do anything except to render a view. Let's do that now.

1.4 Creating views

To create a view, right-click on the action method name and select "Add View" (shown in Figure 1.6.

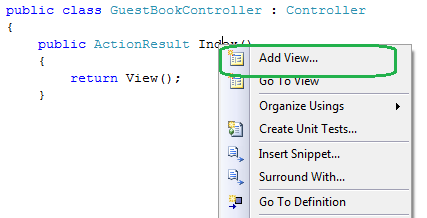


Figure 1.6 - Right click on an action to create a view.

You'll see a dialog box asking you for some information about the view (shown in Figure 1.7). The view name (by default) is the same name as the action, so verify that it says "Index." You can ignore the other options for now and click Add.

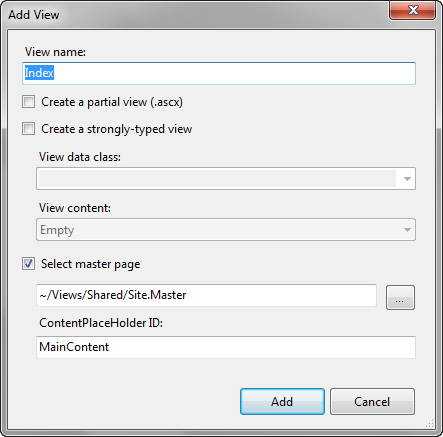


Figure 1.7 - The Add View dialog

Visual Studio will automatically create the appropriate folder and place the Index.aspx file in it. Open up this file and modify it so that it looks like Listing 1.6.

Listing 1.6 - The GuestBook Index view

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

Index #1

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<h2>Guest Book</h2>

<p>Please sign the Guest Book!</p>

<form method="post" action="/GuestBook/Sign"> #2

<fieldset>

<legend>Guest Book</legend>

<%= Html.Label("Name") %> |#3

<%= Html.TextBox("Name") %> |

|

<%= Html.Label("Email") %> |

<%= Html.TextBox("Email") %> |

|

<%= Html.Label("Comments") %> |

<%= Html.TextArea("Comments", new { rows=6, cols=30 }) %> |

<div>

<input type="submit" value="Sign" />

</div>

</fieldset>

</form>

</asp:Content>

(#1) Content controls to change the title of the page

(#2) This action does not exist (yet)

(#3) These are HTML Helpers

Cueballs in text

Using Content controls you can have sections of content that are placed in different areas on your page. The Master Page defines the various ContentPlaceHolders you can use. (#1) shows how you can change the title of the page without having to hard code it in the Master Page.

Our view has some form fields, so we need a <form> tag. Unlike Web Forms, ASP.NET MVC does not create any implicit forms for you. We create a simple form that posts to the URL "/GuestBook/Sign" (#2). This action doesn't exist yet, but we will create it in just a minute.

Inside the form we have some HTML helpers that generate form controls for us (#3). For now, just know that these output the HTML required for each element, but have some friendly functionality to deal with validation errors & automatic binding of data.

Before we run the application, we can add a couple of CSS entries to make the form look decent. Open up the /Content/Site.css file and add the following code somewhere in the file:

fieldset label

{

display: block;

}

fieldset input

{

display: block;

margin-bottom: 5px;

}

We're ready to run the application. Go ahead and press CTRL+F5 and see the site open. Navigate to [http://localhost:{port}/GuestBook](http://localhost:%7bport%7d/GuestBook). You should see the page in Figure 1.7.

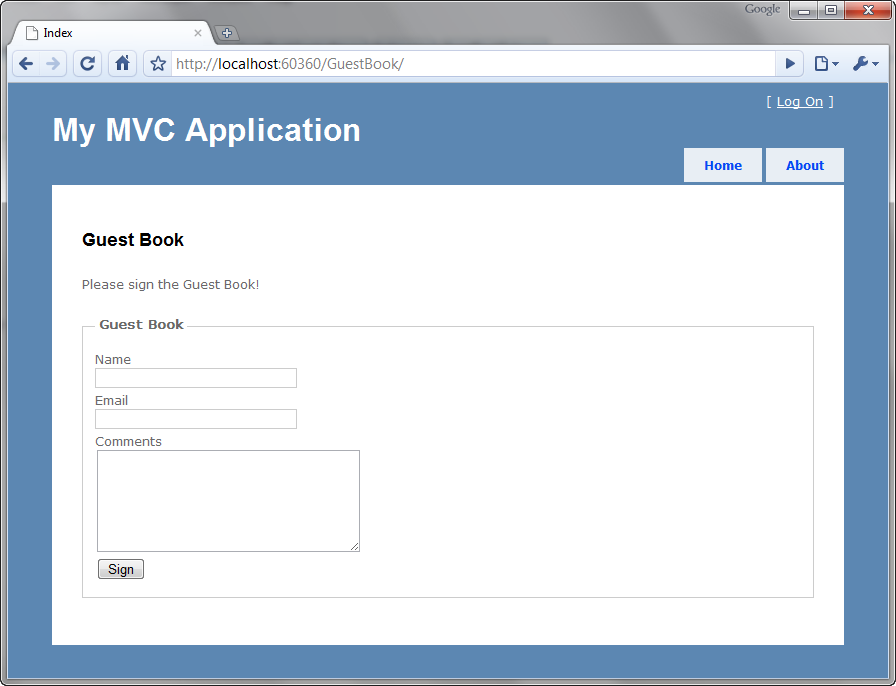


Figure 1.7 - Our GuestBook view

Notice that we only supplied "GuestBook" in the URL. The Index part was implied. How did this happen? Remember the routing rule from before? The default action is defined as "Index", which is what is happening here.

If you tried to fill out the form, you'll quickly find out that a 404 error occurs. This is because we haven't written the action that the form posts to yet! We'll do that next.

Open up GuestBookController and write the action in listing 1.7

Listing 1.7 - An action to respond to the form post

public ActionResult Sign(string name, string email, string comments) #1

{

//do something with the values, such as send an email

ViewData["name"] = name; |#2

ViewData["email"] = email; |

ViewData["comments"] = comments; |

return View("ThankYou"); #3

}

#1 Form values become method arguments

#2 Putting data into the ViewData dictionary

#3 Returning a specific view

In this action, you can see that the arguments match the names of our form values. This is intentional because the ASP.NET MVC Framework will automatically convert values from posted form values, query string values, and other places.

We want to access this data on the view (so that we can present the entry to the user). To do this, we utilize a feature called ViewData. This is a dictionary object (which means you put objects in a data structure which are referenced by a key.

Finally we return a specific view, called "ThankYou". You don't necessarily have to choose a view name that matches the action name, though in many cases that is most desirable. We'll create this view now (listing 1.8).

Listing 1.8 - The ThankYou.aspx view

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

ThankYou

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<h2>Thank You!</h2>

<p>Thank you for signing the guest book! You entered:</p>

Name: <%= ViewData["name"] %><br />

Email: <%= ViewData["email"] %><br />

Comments: <i><%= ViewData["comments"] %></i>

</asp:Content>

In the view we access the data that was provided by the controller. Notice how we use code blocks <%= %> to output the values.

Now we're done with our feature. If we run the application one more time and fill out some values (figure 1.8) then we should be taken to a new page that shows us what we submitted (figure 1.9).

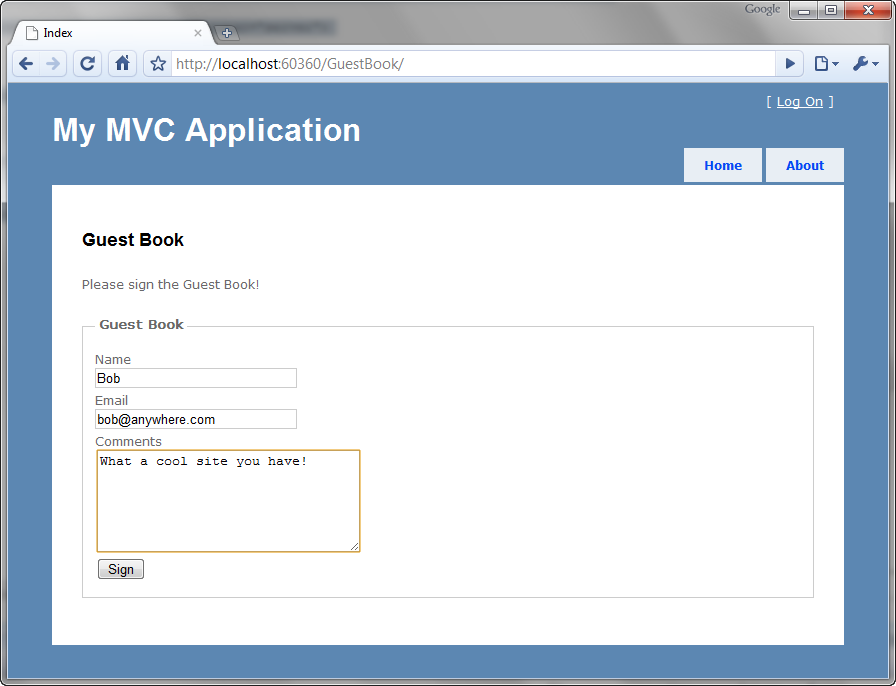


Figure 1.8 - Submitting the Guest Book form.

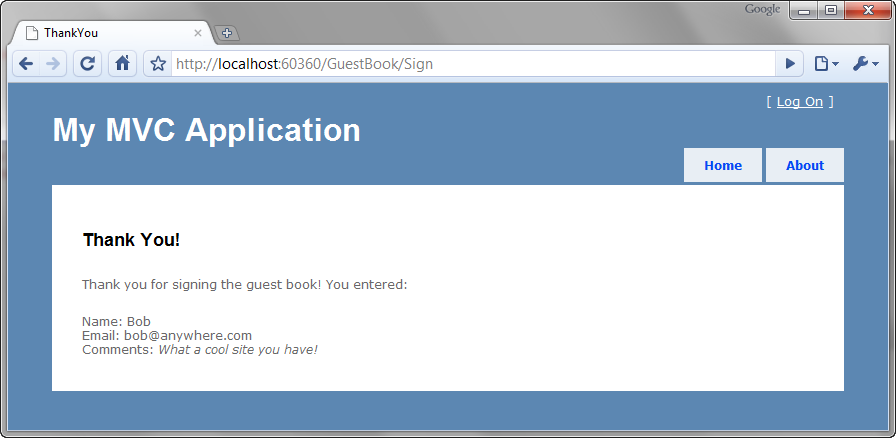


Figure 1.9 - Our data is displayed back to us.

Our first application is complete. While it is functional, it contains a number of problems that your authors consider bad practices. Namely:

* The URL in the form tag is hard-coded. If we change our URL structure then this will break..
* No Model. We dealt purely with primitive values.
* Using ViewData["foo"] utilizes magic strings and relies on casting to do anything meaningful with the data
* The URL still says "Sign" even though we rendered the ThankYou view. This is because we didn't redirect to a success page, we simply rendered one. This causes the next unfortunate aspect of the site…
* If you refresh this page, it will prompt you to submit the data again. If the user obliges, then two records will be posted with the same data.

This example is complete (and probably representative of many examples you'd find online) however it demonstrates some real problems that should not be present in a real application. This book is about practical ASP.NET MVC development practices that we would recommend. Let's take the remainder of this chapter to clean up some of these shortcomings.

1.5 Improving our Application

On the Index.aspx view, we have a hand-written form tag. This in itself is not bad, however we have hard-coded the URL. Using different routing rules, our URLs could easily change and that would cause this form to break. Instead, let's leverage the framework to build our form tag for us. We can use Html.BeginForm to generate a form tag like this:

<% using(Html.BeginForm("Sign")) { %>

<!-- form fields here -->

<% } %>

Html.BeginForm is a special HTML helper. It doesn't directly return a string (where we'd have to use <%=). Instead it uses the IDisposable pattern to gracefully wrap the form contents in a <form> </form> set of tags. The first argument is the name of the action. You're free to use the alternative <% Html.BeginForm(); %> without the curly braces, however you'll have to write </form> yourself.

We can simplify this further by just making the action name the same as the action that was rendered (Index). In this case we can omit the argument to BeginForm. Listing 1.9 contains the change.

Listing 1.9 - Using the Html.BeginForm helper to generate a form tag

<h2>Sign the Guest Book!</h2>

<% using (Html.BeginForm()) {%> #1

<!-- snip -->

<% } %>

(#1) Action name 'Index' will be inferred.

The next step is to create a model. The model does not have to be any particular type of object or inherit from any special class. It can be any class at all. Let's create a model class that represents the data that the user will be posting back to the server (listing 1.10).

Listing 1.10 - Creating a View Model for our GuestBook application

public class GuestBookEntry

{

public string Name { get; set; }

public string Email { get; set; }

public string Comments { get; set; }

}

Notice that the class doesn't contain any logic, nor does it have any dependencies on other systems. It is simply a data container.

Next let's move our attention to the Sign action in our GuestBookController class. Earlier we decided to change the action name to Index to simplify the rendering of the form. It makes sense, actually, to have one action method respond to the HTTP GET request and another respond to the HTTP POST. In general, a GET request should not be allowed to alter the system. To enforce the POST-only nature of this action we can apply the [HttpPost] attribute to the action.

Now that we have a model object representing the form fields on the view, instead of taking separate parameters in the action, we can simply use our newly created model, GuestBookEntry. Listing 1.11 shows these changes.

Listing 1.11 - Accepting a complex object as an action parameter

public ActionResult Index()

{

return View();

}

[HttpPost]

public ActionResult Index(GuestBookEntry entry)

{

/\* snip \*/

}

You're probably wondering how it is possible to accept a complex object like that. The answer lies in the magic of model binding. You'll learn all about model binding later in chapter 14, but for now, just understand that the ASP.NET MVC Framework is smart enough to bind these objects where the property names match keys contained in the Request.Form collection as well as Request.QueryString.

One more advantage of having a strongly-typed model for use on the View is that we can utilize the strongly-typed view helpers and get rid of the magic strings we saw back in listing 1.6. We'll use what are called strongly-typed views to define a specific type for view data for a given view. This is accomplished by changing the Inherits directive of the view to include ViewPage<T> (rather than just ViewPage). Listing 1.12 shows this change.

Listing 1.12 - Changing the Index view to ViewPage<T> in order to benefit from strongly-typed view data

<%@ Page Title="" Language="C#"

MasterPageFile="~/Views/Shared/Site.Master"

Inherits="System.Web.Mvc.ViewPage<GuestBookEntry>" %> #1

#1 - Type names must be fully qualified

Now our Index view requires an instance of GuestBookEntry to be assigned to the view before rendering. We need to revisit our action to make sure this is provided. Listing 1.13 shows our original Index action modified to send a new instance of GuestBookEntry to the view.

Listing 1.13 - Providing the expected view data instance to the view

public ActionResult Index()

{

var model = new GuestBookEntry();

return View(model);

}

Instead of just rendering a view, we must provide an instance of GuestBookEntry. This makes perfect sense as we are indeed creating a new GuestBookEntry on the form.

Now we can use the strongly-typed view helpers, shown in Listing 1.14. Notice the lack of magic strings!

Listing 1.14 - Using strongly-typed view helpers instead of strings

<h2>Sign the Guest Book!</h2>

<% using (Html.BeginForm()) {%>

<fieldset>

<legend>Fields</legend>

<p>

<%= Html.LabelFor(model => model.Name) %>

<%= Html.TextBoxFor(model => model.Name) %>

</p>

<p>

<%= Html.LabelFor(model => model.Email) %>

<%= Html.TextBoxFor(model => model.Email) %>

</p>

<p>

<%= Html.LabelFor(model => model.Comments) %>

<%= Html.TextAreaFor(model => model.Comments) %>

</p>

<p>

<input type="submit" value="Create" />

</p>

</fieldset>

<% } %>

We have a couple more changes before we'll be done. Remember that we noticed that a refresh would cause the form data to be re-Posted and therefore we'd have duplicate entries in the guest book? To fix this we will leverage the Post-Redirect-Get pattern (or PRG). It's actually quite simple:

* Post some data to an action
* Redirect the user to a different action
* The user's browser issues a GET for the new action

Because the browser is issuing a GET as the last request, a refresh does no harm at all. It simply retrieves the page again.

Our controller can be augmented to implement this pattern, shown in listing 1.15. In order to satisfy rendering the data back to the user (since we are not redirecting them) we need to store the data somewhere temporarily. TempData is perfect for this. TempData is a collection that you can use to store data. It will be persisted in server Session memory for one round-trip.

Listing 1.15 - Implementing Post-Redirect-Get

public class GuestBookController : Controller

{

public ActionResult Index()

{

var model = new GuestBookEntry();

return View(model);

}

[HttpPost]

public ActionResult Index(GuestBookEntry entry)

{

TempData["entry"] = entry; #1

return RedirectToAction("ThankYou"); #2

}

public ActionResult ThankYou()

{

if(TempData["entry"] == null) |#3

{ |

return RedirectToAction("index"); |

} |

var model = (GuestBookEntry) TempData["entry"]; #4

return View(model);

}

}

#1 - Hang on to the submitted data so we can retrieve it upon redirect

#2 - Redirect the user to the ThankYou action

#3 - Make sure the user has the data, if not redirect them back to Index

#4 - Retrieve the data from TempData & pass it to the view

The only thing remaining is to modify the ThankYou view to be strongly-typed as well. This time we will do it with the Add View dialog box, so first delete the ThankYou.aspx file. Next we'll right click on the action method and choose Add View just like we did before. This time we will check the box to create a strongly-typed view. Look at Figure 1.10 to see what the options should look like, then press Add.

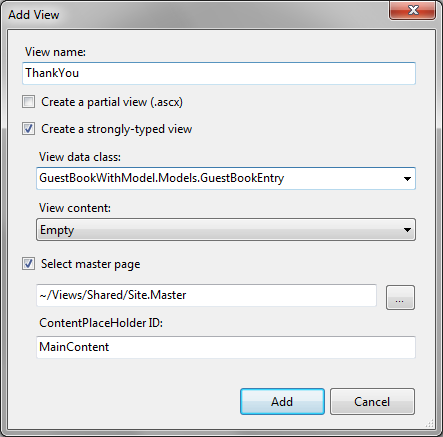


Figure 1.10 - Creating a strongly-typed view using the Add View dialog

Your model object might not show up at first, so make sure you've built the solution before opening this dialog. Also, your namespace might differ from the one shown here.

Inside the view, we'll utilize a quick helper called Html.DisplayForModel(). This relies on a neat feature called Templated Helpers that you will learn about in Chapter 3. For now, just enjoy the free functionality! Listing 1.16 shows the ThankYou view.

Listing 1.16 - The ThankYou view uses a helper to display our model object

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<GuestBookWithModel.Models.GuestBookEntry>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

ThankYou

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<h2>Thank You!</h2>

Thank you for signing our Guest Book. You entered: <br />

<%= Html.DisplayForModel() %>

</asp:Content>

Isn't that much easier? No need to enumerate all of the properties if you want to simply output the whole thing.

We have made a number of changes to make this application a little bit nicer. We addressed each one of the problems listed back in section 1.4 and we now have a fully functional guest book application. Go ahead and run it. Notice how your URL says "ThankYou" when you've signed the guest book. Also notice that when you refresh, the system handles it gracefully and brings you back to the Index view.

1.6 Summary

We covered a lot of material in this chapter. Congratulations on making it through. You are now well-positioned to dive into each subtopic in more depth. Now that you have the big picture, you see that programming pages with the MVC pattern is quite a bit different from programming with Web Forms. You have seen that the first difference is the added simplicity.

In this chapter you learned how to create a project, add controllers and views, work with models and strongly-typed view data, and the PRG pattern. You learned how to deal with user input, leveraged model binding, and how to use TempData to stash data off for a single round-trip to access it later. Phew!

The rest of the book will contain much more focused chapters, in order to give you a broad understanding of each concept in the book. Let's begin this journey with an in-depth look at The Presentation Model. Both controllers and views depend on the shape of the presentation model used; therefore, a firm understanding in this area will serve you will. Read on.