CHAPTER 14

Overview of MVC Projects

I am going to provide some additional context before diving into MVC Framework features. This chapter gives an overview of the structure and nature of an ASP.NET MVC application, including the default project structure and naming conventions, some of which are optional and some of which are hard-coded into the way that the MVC Framework operates.

Working with Visual Studio MVC Projects

When you create a new ASP.NET project, Visual Studio gives you a number of choices as to the initial content you want in the project. The idea is to ease the learning process for new developers and apply some time-saving best practices for common features and tasks. This support continues with templates used to create controllers and views that are created with template code for listing data objects, editing model properties and so on.

With Visual Studio 2013 and MVC 5, Microsoft has updated the templates and *scaffolding*, as it is known, to blur the boundaries between different kinds of ASP.NET project and to provide a wider range of project templates and code configurations.

Part 1 of this book will have left you in no doubt that I am not a fan of this kind of approach to cookie cutter project or code. The intent is good, but the execution is always underwhelming. One of the characteristics I like most about ASP.NET and the MVC Framework is just how much flexibility I have in tailoring the platform to suit my development style, and the projects, classes and views that Visual Studio creates and populates make me feel constrained to work in someone else's style. I also find the content and configuration too generic and too bland to be useful. In Chapter 10, I mentioned that one of the dangers of using responsive design to target mobile devices is a kind of averaging that ends up compromising the experience for all devices, and something similar has happened to the Visual Studio templates. Microsoft can't possibly know what kind of application you need to create and so they cover all the bases, but in such a drab and generalized way that I end up just ripping out the default content anyway.

My advice (given to anyone who makes the mistake of asking) is to start with an empty project and add the folders, files, and packages that you need. Not only will you learn more about the way that the MVC Framework works, but you will have complete control over what your application contains.

But my preferences should not color your development experience. You may find the templates and scaffolding more useful than I do, especially if you are new to ASP.NET development and you have not yet developed a distinctive personal development style that suits you. You may also find the project templates a useful resource and a source of ideas, although you should be cautious about adding any functionality to an application before you completely understand how it works.

Creating the Project

When you first create a new MVC Framework project, you have two basic starting points to choose from: the `Empty` template and the `MVC` template. The names are a little misleading, because you can add the basic folders and assemblies required for the MVC Framework to any project template by checking the `MVC` option in the `Add Folders`

and Core References section of the New ASP.NET Project dialog window, as shown in Figure 14-1. For the MVC option, this option is checked for you.

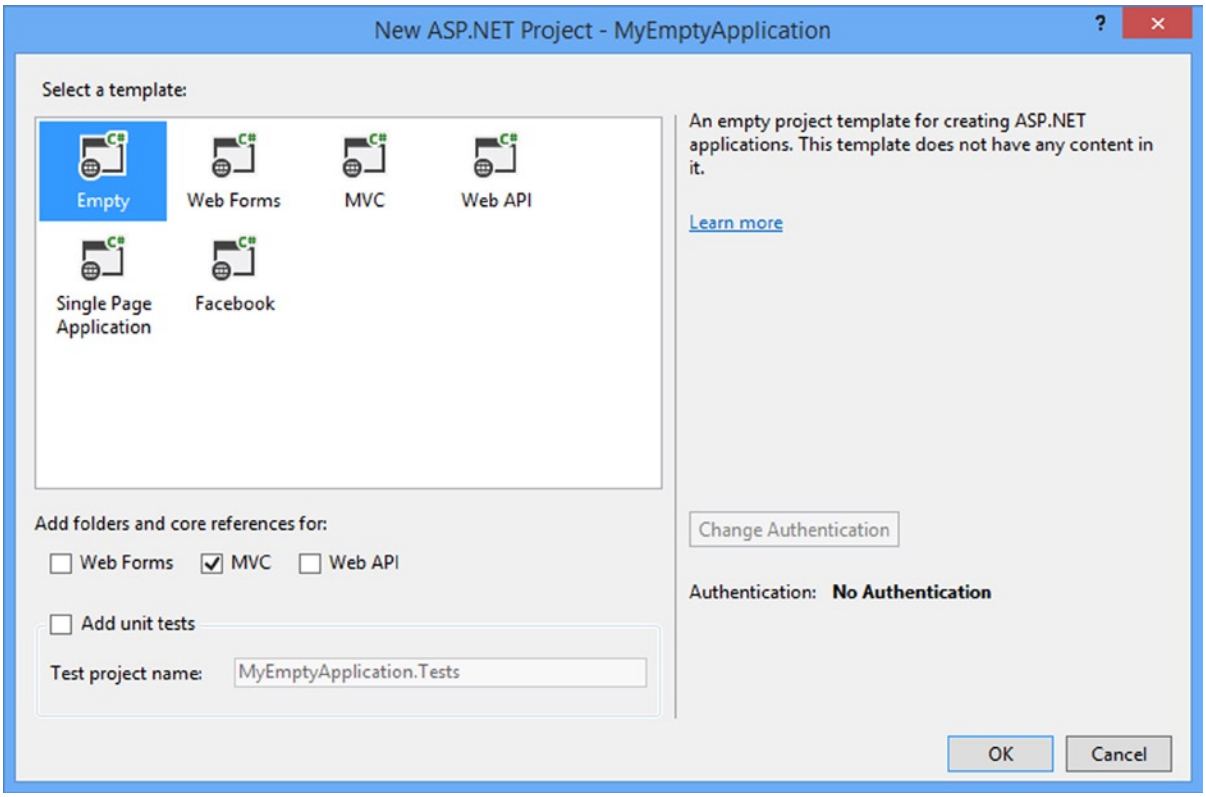

Figure 14-1. *Selecting the project type, folders and assemblies for a new project*

The real difference is the additional content that the MVC project template adds to new projects, which provides a ready-made starting point that includes some default controllers and views, a security configuration, some popular JavaScript and CSS packages (such as jQuery and Bootstrap) and a layout that uses Bootstrap to provide a theme for the application content. The Empty project option just contains the basic references required for an MVC framework and the barebones folder structure. There is a fair amount of content added by the MVC template and you can see the differences in Figure 14-2, which shows the contents of two newly created projects. The one on the left was created with the Empty template with the MVC folders and references option checked. The others show the content of a project that was created with the MVC template, and to be able to show the files on the page, I had to focus the Solution Explorer on different folders because a single listing was too long for a printed page.

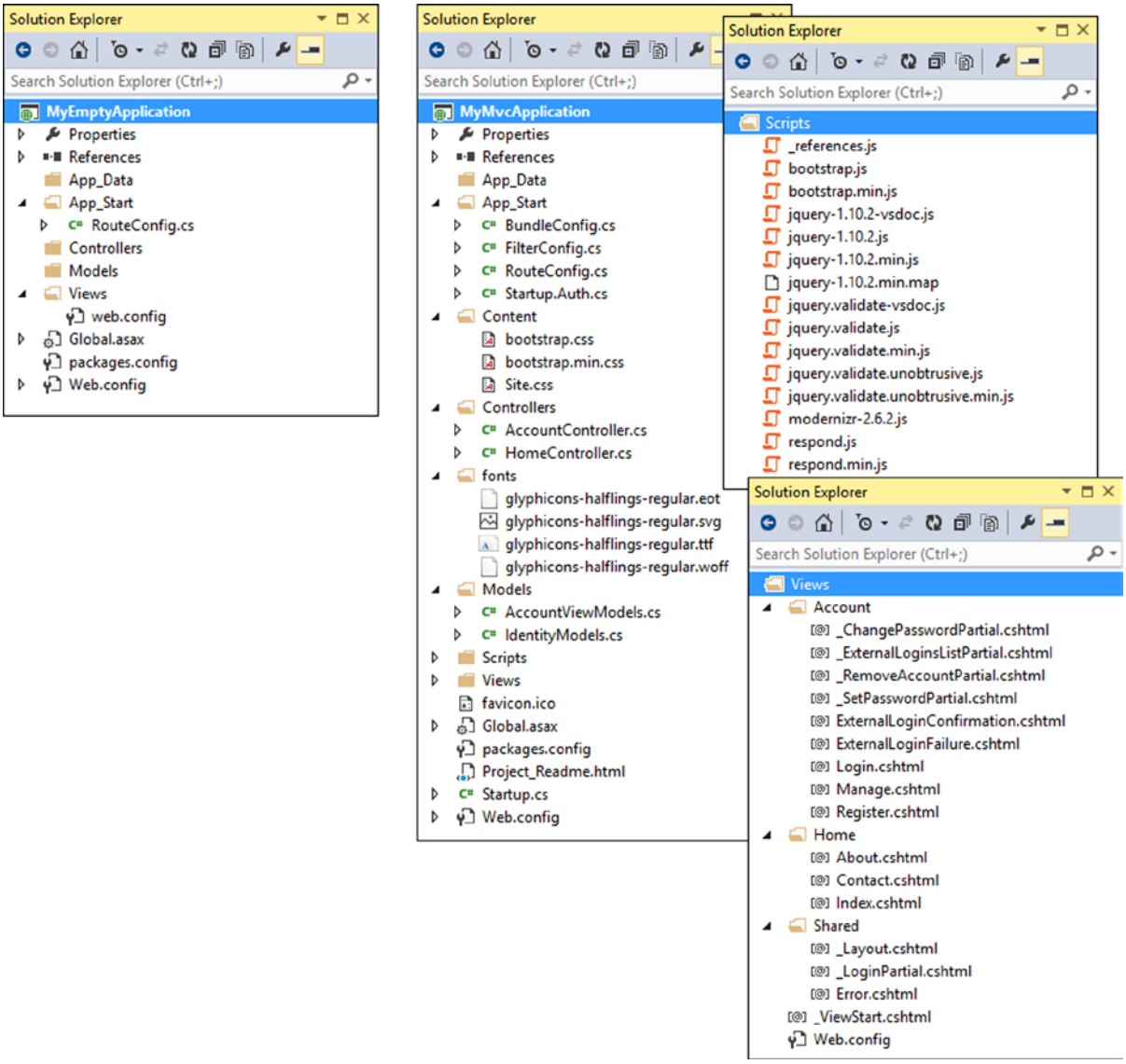

Figure 14-2. *The default content added to a project by the Empty and MVC templates*

The extra files that the MVC project adds look worse than they really are. Some are related to authentication and others are JavaScript and CSS files for which there are regular and minified versions. (I describe how these can be used in Chapter 26.)

■ **Tip** Visual Studio assembles a project created with the MVC template using NuGet packages, which means that you can see which packages are used by selecting `Manage NuGet Packages for Solution` from the `Visual Studio Tools ➤ Library Package Manager` menu. It also means that you can add the same packages to any project, including one created with the `Empty` template (and this is what I was doing for the examples in Part 1 of this book).

Whichever template you choose, you will notice that the resulting projects have similar folder structures. Some of the items in an MVC project have special roles, which are hard-coded into ASP.NET or the MVC Framework. Others are subject to naming conventions. I have described each of the core files and folders in Table 14-1, some of which are not present in projects by default but which I introduce in later chapters.

Table 14-1. *Summary of MVC Project Items*

Folder or File	Description	Notes
`/App_Data`	This folder is where you put private data, such as XML files or databases if you are using SQL Server Express, SQLite, or other file-based repositories.	IIS will not serve the contents of this folder.
`/App_Start`	This folder contains some core configuration settings for your project, including the definition of routes and filters and content bundles.	I describe routes in Chapters 15 and 16, filters in Chapter 18 and content bundles in Chapter 26.
`/Areas`	Areas are a way of partitioning a large application into smaller pieces.	I describe areas in Chapter 15.
`/bin`	The compiled assembly for your MVC application is placed here, along with any referenced assemblies that are not in the GAC.	You won't see the `bin` directory in the `Solution Explorer` window unless you click the `Show All Files` button. Since these are binary files generated on compilation, you should not normally store them in source control.
`/Content`	This is where you put static content such as CSS files and images.	This is a convention but not required. You can put your static content anywhere that suits you.
`/Controllers`	This is where you put your controller classes.	This is a convention. You can put your controller classes anywhere you like, because they are all compiled into the same assembly.
`/Models`	This is where you put your view model and domain model classes, although all but the simplest applications benefit from defining the domain model in a dedicated project, as I demonstrated for SportsStore.	This is a convention. You can define your model classes anywhere in the project or in a separate project.
`/Scripts`	This directory is intended to hold the JavaScript libraries for your application.	This is a convention. You can put script files in any location, as they are just another type of static content. See Chapter 26 for more information about managing script files.

(*continued*)

Table 14-1. (*continued*)

Folder or File	Description	Notes
`/Views`	This directory holds views and partial views, usually grouped together in folders named after the controller with which they are associated.	The `/Views/web.config` file prevents IIS from serving the content of these directories. Views must be rendered through an action method.
`/Views/Shared`	This directory holds layouts and views which are not specific to a single controller.	
`/Views/Web.config`	This is *not* the configuration file for your application. It contains the configuration required to make views work with ASP.NET and prevents views from being served by IIS and the namespaces imported into views by default.	
`/Global.asax`	This is the global ASP.NET application class. Its code-behind class (`Global.asax.cs`) is the place to register routing configuration, as well as set up any code to run on application initialization or shutdown, or when unhandled exceptions occur.	The `Global.asax` file has the same role in an MVC application as it does in a Web Forms application.
`/Web.config`	This is the configuration file for your application.	The `Web.config` file has the same role in an MVC application as it does in a Web Forms application.

Understanding MVC Conventions

There are two kinds of conventions in an MVC project. The first kind is just suggestions as to how you might like to structure your project. For example, it is conventional to put your JavaScript files in the `Scripts` folder. This is where other MVC developers would expect to find them, and where NuGet packages will install them. But you are free to rename the `Scripts` folder, or remove it entirely and put your scripts somewhere else. That would not prevent the MVC Framework from running your application as long as the `script` elements in your views refer to the location you settle on.

The other kind of convention arises from the principle of *convention over configuration*, which was one of the main selling points that made Ruby on Rails so popular. Convention over configuration means that you don't need to explicitly configure associations between controllers and their views, for example. You just follow a certain naming convention for your files, and everything just works. There is less flexibility in changing your project structure when dealing with this kind of convention. The following sections explain the conventions that are used in place of configuration.

■ **Tip** All of the conventions can be changed if you are using a custom view engine (which I cover in Chapter 20), but this is not a step to be taken lightly and, for the most part, these are the conventions you will be dealing with in MVC projects.

Following Conventions for Controller Classes

Controller classes must have names that end with `Controller`, such as `ProductController`, `AdminController`, and `HomeController`. When referencing a controller from elsewhere in the project, such as when using an HTML helper method, you specify the first part of the name (such as `Product`), and the MVC Framework automatically appends `Controller` to the name and starts looking for the controller class.

■ **Tip** You can change this behavior by creating your own implementation of the `IControllerFactory` interface, which I describe in Chapter 19.

Following Conventions for Views

Views and partial views go into the folder `/Views/`*`Controllername`*. For example, a view associated with the `ProductController` class would go in the `/Views/Product` folder.

■ **Tip** Notice that I omit the `Controller` part of the class from the `Views` folder: `/Views/Product`, *not* `/Views/ProductController`. This may seem counterintuitive at first, but it quickly becomes second nature.

The MVC Framework expects that the default view for an action method should be named after that method. For example, the default view associated with an action method called `List` should be called `List.cshtml`. Thus, for the `List` action method in the `ProductController` class, the default view is expected to be `/Views/Product/List.cshtml`. The default view is used when you return the result of calling the `View` method in an action method, like this:

```
...
return View();
...
```

You can specify a different view by name, like this:

```
...
return View("MyOtherView");
...
```

Notice that I do not include the file name extension or the path to the view. When looking for a view, the MVC Framework looks in the folder named after the controller and then in the `/Views/Shared` folder. This means that I can put views that will be used by more than one controller in the `/Views/Shared` folder and the framework will find them.

Following Conventions for Layouts

The naming convention for layouts is to prefix the file with an underscore (_) character, and layout files are placed in the `/Views/Shared` folder. This layout is applied to all views by default through the `/Views/_ViewStart.cshtml` file. If you do not want the default layout applied to views, you can change the settings in `_ViewStart.cshtml` (or delete the file entirely) to specify another layout in the view, like this:

```
@{
    Layout = "~/Views/Shared/_MyLayout.cshtml";
}
```

Or you can disable any layout for a given view, like this:

```
@{
    Layout = null;
}
```

Debugging MVC Applications

You debug an ASP.NET MVC application in exactly the same way as you debug an ASP.NET Web Forms application. The Visual Studio debugger is a powerful and flexible tool, with many features and uses. I can only scratch the surface in this book, but in the sections that follow I show you how to set up the debugger and perform different debugging activities on your MVC project.

Preparing the Example Project

To demonstrate using the debugger, I have created a new MVC project using the `MVC` project template, just so you can see how the default content and configuration is set up and the effect of the default theme that is applied to views. I called the new project `DebuggingDemo`, as shown in Figure 14-3. I have chosen the `Individual User Accounts` authentication option, which sets up a basic user security system.

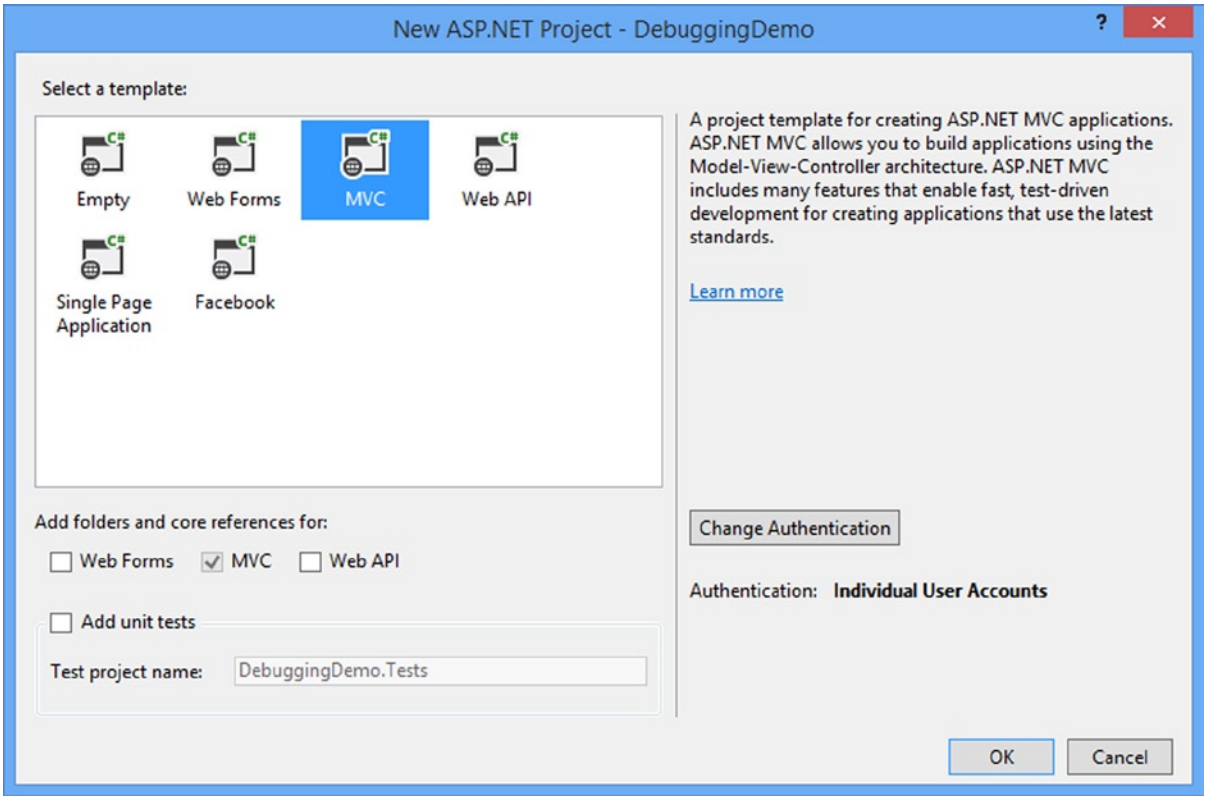

Figure 14-3. *Creating a new project using the MVC project template*

Click the OK button and Visual Studio will create the project and add the default packages, files, and folders that the MVC template includes. You can see how the files and settings added to the project are applied by starting the application, as shown in Figure 14-4.

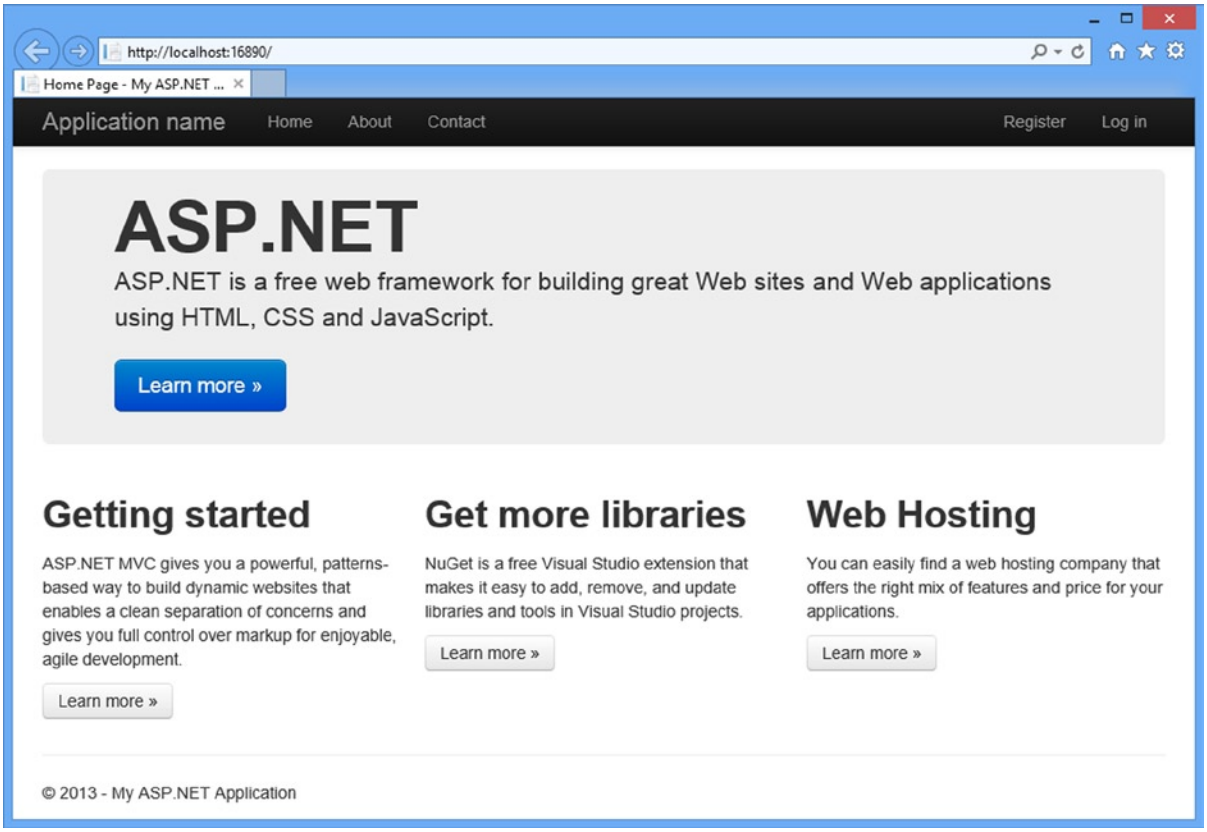

Figure 14-4. *The effect of the additions made by the MVC project template*

There are some placeholder elements for the name of the application and branding, and some pointers to the MVC documents, NuGet, and some hosting options. The navigation bar at the top of the screen is the same kind I used for the SportsStore application and the layout has some responsive features. Change the width of the window to see the effect.

Creating the Controller

Visual Studio creates a `Home` controller as part of the initial project content, but I am going to replace the code that Visual Studio added with that shown in Listing 14-1.

Listing 14-1. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
```

```
            int secondVal = 5;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

Creating the View

Visual Studio also created the `Views/Home/Index.cshtml` view file as part of the project setup. I don't need the default content and have to replace it with the markup shown in Listing 14-2.

Listing 14-2. The Contents of the Index.cshtml File

```
@model int

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <title>Index</title>
</head>
<body>
    <h2 class="message">@ViewData["Message"]</h2>
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

The last preparatory step I need to take is to add a style to the `/Content/Site.css` file and change one of the existing ones, as shown in Listing 14-3. The `Site.css` file is created by Visual Studio as part of the `MVC` project template and is the default location for application CSS styles. (I added a `link` element to the view in Listing 14-2 that imports this file into the `Index.cshtml` view.)

Listing 14-3. Adding a Style to the /Content/Site.css File

```
body { padding-top: 5px; padding-bottom: 5px; }
.field-validation-error { color: #b94a48; }
.field-validation-valid { display: none; }
input.input-validation-error { border: 1px solid #b94a48; }
input[type="checkbox"].input-validation-error { border: 0 none; }
```

```
.validation-summary-errors { color: #b94a48; }
.validation-summary-valid { display: none; }
.no-color { background-color: white; border-style:none; }
.message { font-size: 20pt; text-decoration: underline;}
```

Launching the Visual Studio Debugger

Visual Studio prepares new projects for debugging automatically, but it is useful to understand how to change the configuration. The important setting is in the `Web.config` file in the root project folder and can be found in the `system.web` element, as shown in Listing 14-4.

Listing 14-4. The Debug Attribute in the Web.config File

```
...
<system.web>
   <compilation debug="true" targetFramework="4.5.1" />
   <httpRuntime targetFramework="4.5.1" />
</system.web>
...
```

A lot of compilation in an MVC Framework project is done when the application is running in IIS, and so you need to ensure that the `debug` attribute on the compilation attribute is set to `true` during the development process. This ensures that the debugger is able to operate on the classes files produced through on-demand compilation

---

■ **Caution** Do not deploy your application to a production server without setting the `debug` value to `false`. If you are using Visual Studio to deploy your application (as I did in Chapter 13), then the setting will be changed automatically when you select the Release configuration for the project.

In addition to the `Web.config` file, I want to ensure that Visual Studio includes debugging information in the class files that it creates. This isn't critical, but it can cause problems if the different debug settings are not in sync. Ensure that the Debug configuration is selected in the Visual Studio toolbar, as shown in Figure 14-5.

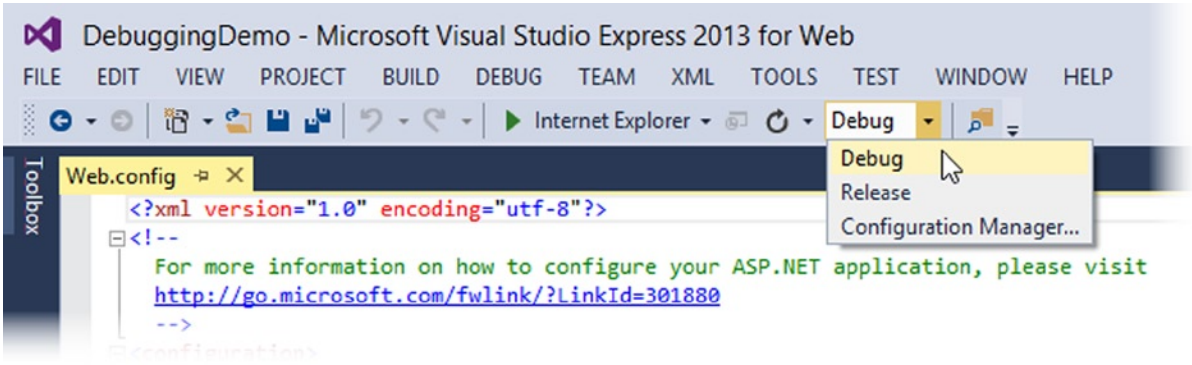

Figure 14-5. *Selecting the Debug configuration*

To debug an MVC Framework application, select `Start Debugging` from the Visual Studio `Debug` menu or click on the green arrow in the Visual Studio toolbar (which you can see in Figure 14-5, next to the name of the browser that will be used to display the app—Internet Explorer in this case).

If the `debug` attribute in the `Web.config` file is set to `false` when you start the debugger, then Visual Studio will display the dialog shown in Figure 14-6. Select the option which allows Visual Studio to edit the `Web.config` file and click the `OK` button and the debugger will start.

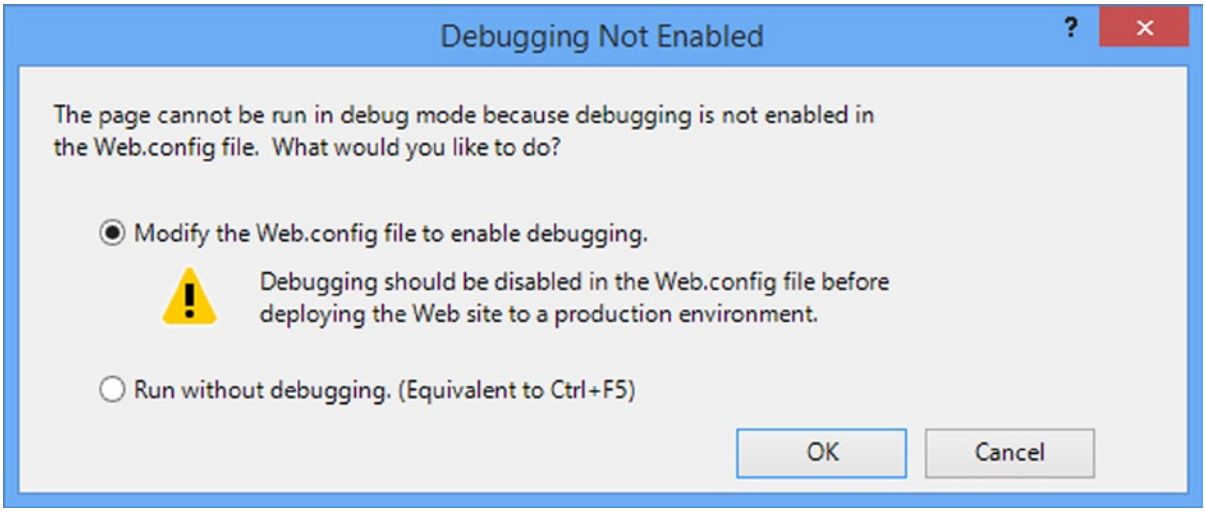

Figure 14-6. *The dialog that Visual Studio displays when the Web.config File disables debugging*

At this point, your application will be displayed in a new browser window, as shown in Figure 14-7.

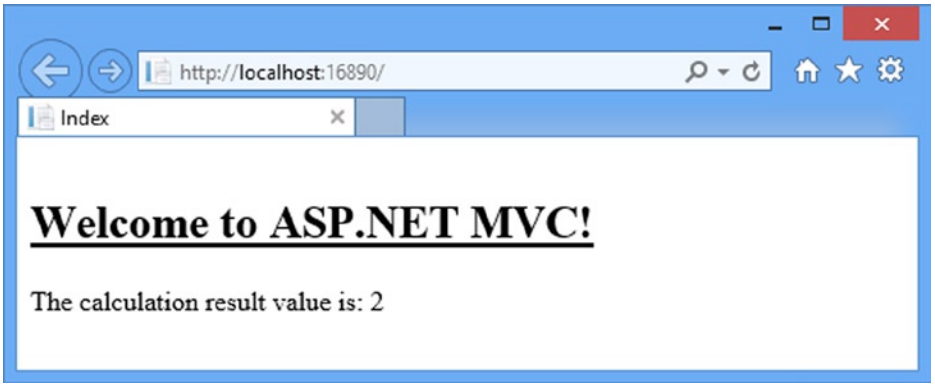

Figure 14-7. *Running the debugger*

The debugger will be attached to your application, but you will not notice any difference until the debugger breaks. (I explain what this means in the next section.) To stop the debugger, select `Stop Debugging` from the Visual Studio `Debug` menu or close the browser window.

Causing the Visual Studio Debugger to Break

An application that is running with the debugger attached will behave normally until a *break* occurs, at which point the execution of the application is halted and control is turned over to the debugger. While in this state, you can inspect and control the state of the application. Breaks occur for two main reasons: when a breakpoint is reached and when an unhandled exception arises. You will see examples of both in the following sections.

Using Breakpoints

A *breakpoint* is an instruction that tells the debugger to halt execution of the application and hand control to the programmer. You can inspect the state of the application and see what is happening and, optionally, resume execution again.

To create a breakpoint, right-click a code statement and select `Breakpoint` ➤ `Insert Breakpoint` from the pop-up menu. As a demonstration, apply a breakpoint to the first statement in the `Index` action method of the `Home` controller and you will see a red dot appear in the margin of the text editor, as shown in Figure 14-8.

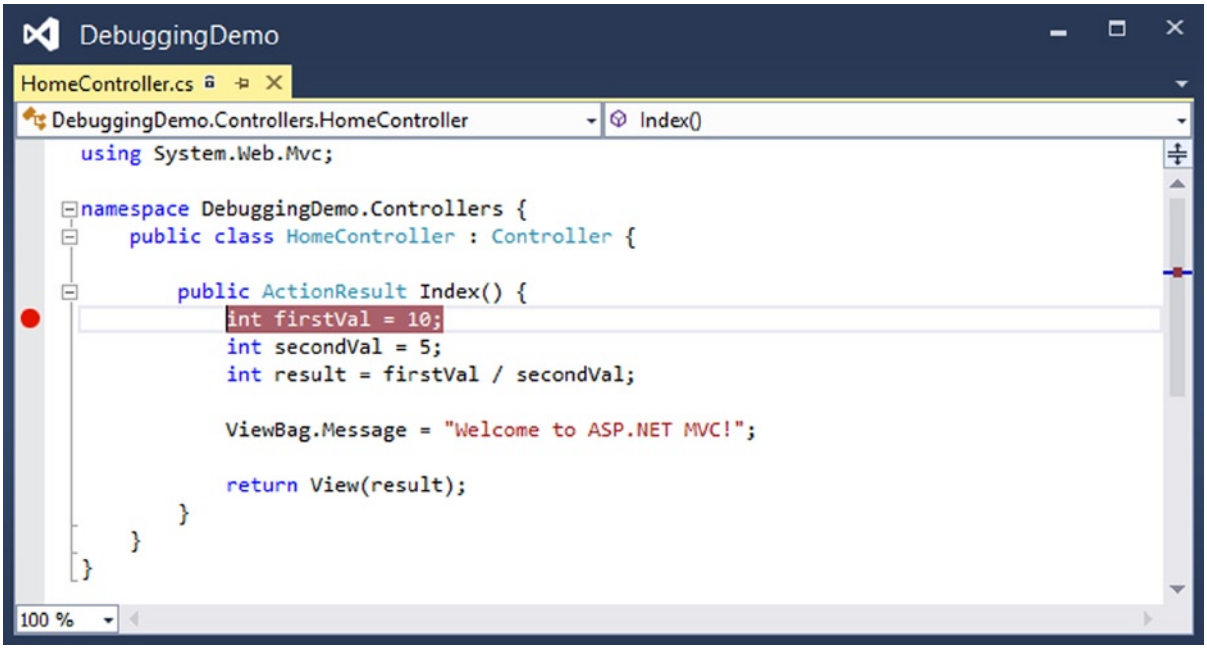

Figure 14-8. *Applying a breakpoint to the first statement in the Index action method*

To see the effect of the breakpoint, start the debugger by selecting `Start Debugging` from the Visual Studio `Debug` menu. The application will run until the statement to which the breakpoint has been applied is reached, at which point the debugger will break, halting execution of the application and transferring control. Visual Studio highlights the point at which the execution has been stopped with yellow highlights, as shown in Figure 14-9.

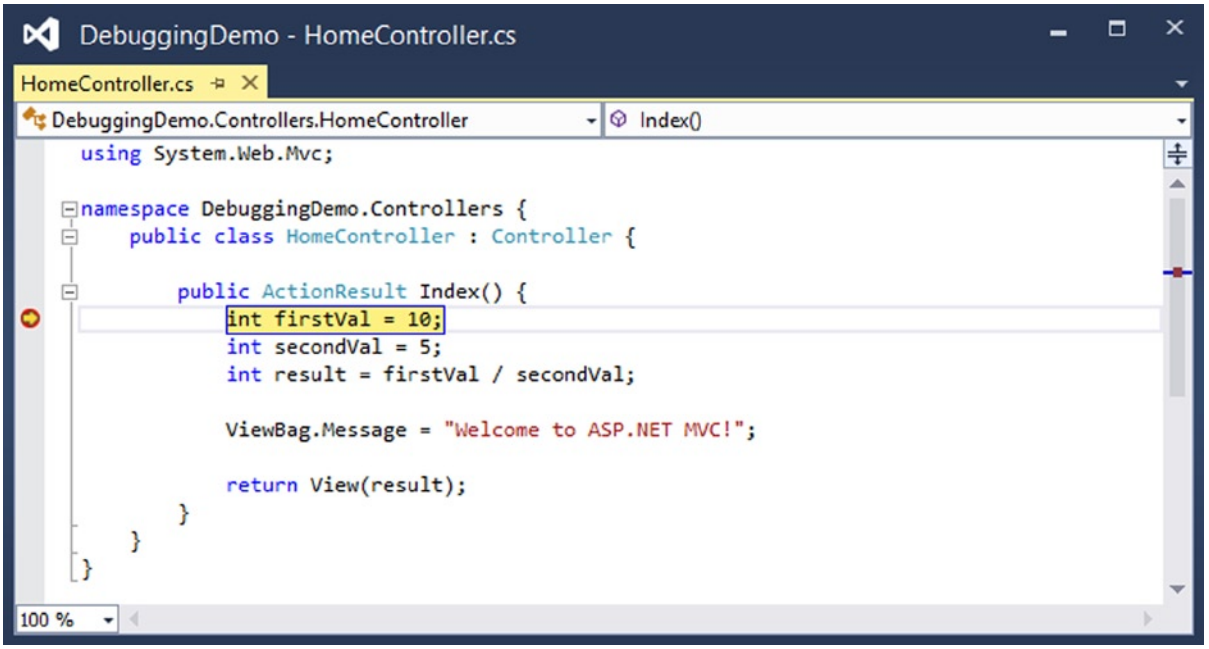

Figure 14-9. *Hitting a breakpoint*

■ **Note** A breakpoint is triggered only when the statement it is associated with is executed. My example breakpoint was reached as soon as the application was started because it is inside the action method that is called when a request for the default URL is received. If you place a breakpoint inside another action method, you must use the browser to request a URL associated with that method. This can mean working with the application in the way a user would or navigating directly to the URL in the browser window.

Once you have control of the application's execution, you can move to the next statement, follow execution into other methods and generally explore the state of your application. You can do this using the toolbar buttons or using the items in the Visual Studio Debug menu. In addition to giving you control of the execution of the app, Visual Studio provides you with a lot of useful information about the state of your app. In fact, there is so much information that I only have room to show you the basics.

Viewing Data Values in the Code Editor

The most common use for breakpoints is to track down bugs in your code. Before you can fix a bug, you have to figure out what is going on and one of the most useful features that Visual Studio provides is the ability to view and monitor the values of variables right in the code editor.

As an example, start the app using the debugger and wait until the breakpoint I added in the previous section is reached. When the debugger breaks, move the mouse pointer over the statement that defines the `result` variable. You will see a small pop-up which shows you the current value, as illustrated by Figure 14-10. It can be hard to make out the pop-up, so I have shown a magnified version in the figure.

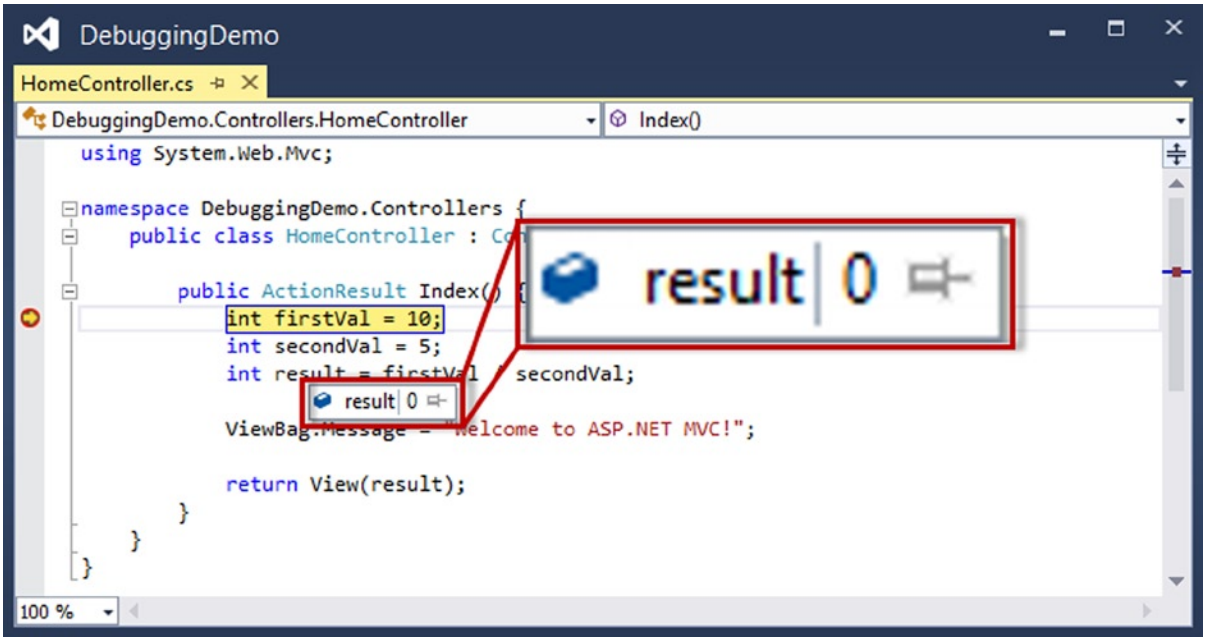

Figure 14-10. *Displaying the value of a variable in the Visual Studio code editor*

The execution of the statements in the `Index` action method has not reached the point where a value has been assigned to the `result` variable, so Visual Studio shows the default value, which is `0` for the `int` type. Select the `Step Over` menu item in the Visual Studio `Debug` menu (or press `F10`) to advance the point of execution to the statement which defines the `ViewBag.Message` property and hold your mouse over the `result` variable again. The debugger executed the statement that assigns a value to the `result` variable, and you can see the effect in Figure 14-11.

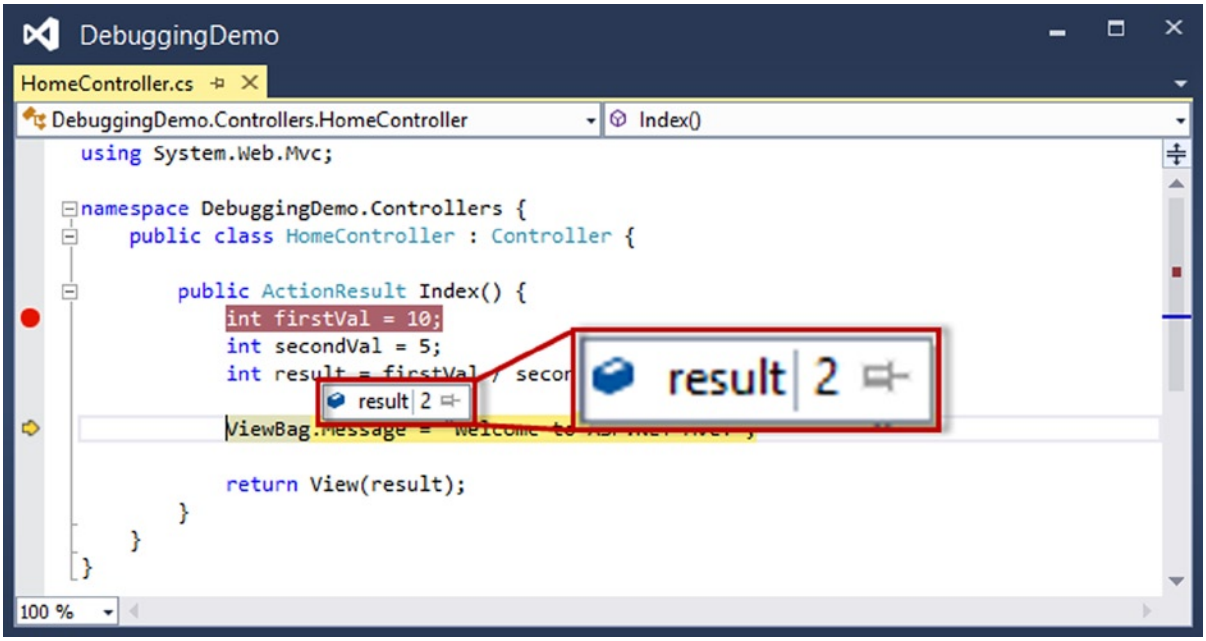

Figure 14-11. *The effect of assigning a value of a variable*

I use this feature when I start the process of tracking down a bug because it gives an immediate insight into what is going on inside the app. I find it especially useful for spotting `null` values, which indicate that a variable has not been assigned a value (a cause of many bugs early in the development process, in my experience).

You will notice that there is a pushpin icon to the right of the value in the pop-up. If you click on this, the pop-up becomes permanent and will indicate when the value of the variable changes. This allows you to monitor one or more variables and see when they change and what their new values are.

Viewing Application State in the Debugger Windows

Visual Studio provides a number of different windows that you can use to get information about your app while the execution has been halted following a breakpoint. A complete list of the windows available is shown on the `Debug` ➤ `Windows` menu, but two of the most useful are the `Locals` and `Call Stack` windows. The `Locals` window automatically displays the value of all of the variables in the current scope, as Figure 14-12 illustrates. This gives you an all-in-one view of the variables, which are likely to be of interest.

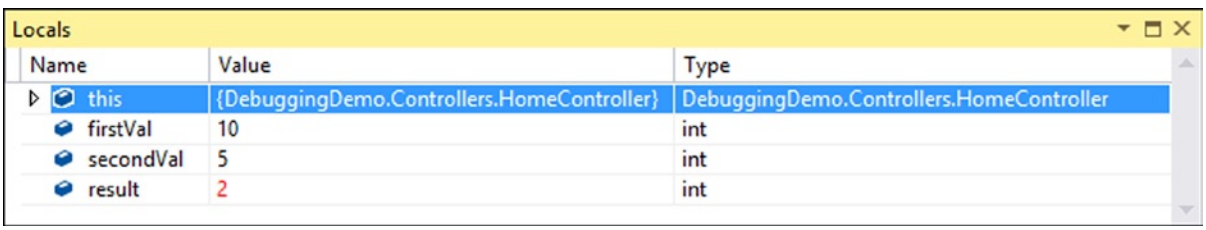

Figure 14-12. *The Locals window*

Variables whose values were changed by the previously executed statement are shown in red. In the figure, the `result` variable is red because I just executed the statement that assigns a value.

■ **Tip** The set of variables shown in the `Locals` window changes as you navigate through the application, but if you want to keep an eye on a variable globally, then right-click on one of the items shown in the `Locals` window and select the `Add Watch` option. The items in the `Watch` window don't change as you execute statements in the app, providing you with a fixed point of reference.

The `Call Stack` window shows you the sequence of calls that have led to the current statement being executed. This can be helpful if you are trying to figure out odd behavior because you can unwind the call stack and explore the circumstances that led to the breakpoint being triggered. (I have not shown you the `Call Stack` window in a figure because the example app doesn't have enough call depth to provide a useful insight. But I recommend you explore this and the other Visual Studio windows to get more of an idea of what information the debugger is able to provide.)

■ **Tip** You can add breakpoints to views. This can be helpful for inspecting the values of view model properties, for example. You add a breakpoint to a view just as I did in the code file: right-click the Razor statement that you are interested in and select `Breakpoint` ➤ `Insert Breakpoint`.

Breaking on Exceptions

Unhandled exceptions are a fact of development. One of the reasons that I unit and integration test my projects is to minimize the likelihood that such an exception will occur in production. As an aid to finding and fixing unhandled exceptions, the Visual Studio debugger will break automatically when it encounters one.

■ **Note** Only *unhandled* exceptions cause the debugger to break. An exception becomes *handled* if you catch and handle it in a `try...catch` block. Handled exceptions can be a useful programming tool. They are used to represent the scenario where a method was unable to complete its task and needs to notify its caller. Unhandled exceptions are bad, because they represent an *unforeseen* condition and because they generally drop the user into an error page.

To demonstrate breaking on an exception, I have made a small change to the `Index` action method in the `Home` controller, as shown in Listing 14-5.

Listing 14-5. Adding a Statement That Will Cause an Exception in the HomeController.cs File

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
```

```
            int secondVal = 0;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

I changed the value of the `secondVal` variable to be `0`, which will cause an exception in the statement that divides `firstVal` by `secondVal`.

■ **Note** I also removed the breakpoint from the Index action method by right-clicking on the breakpoint icon in the margin and selecting `Delete Breakpoint` from the pop-up menu.

When you start the debugger, the application will run until the exception is thrown, at which point the exception helper pop-up will appear, as shown in Figure 14-13.

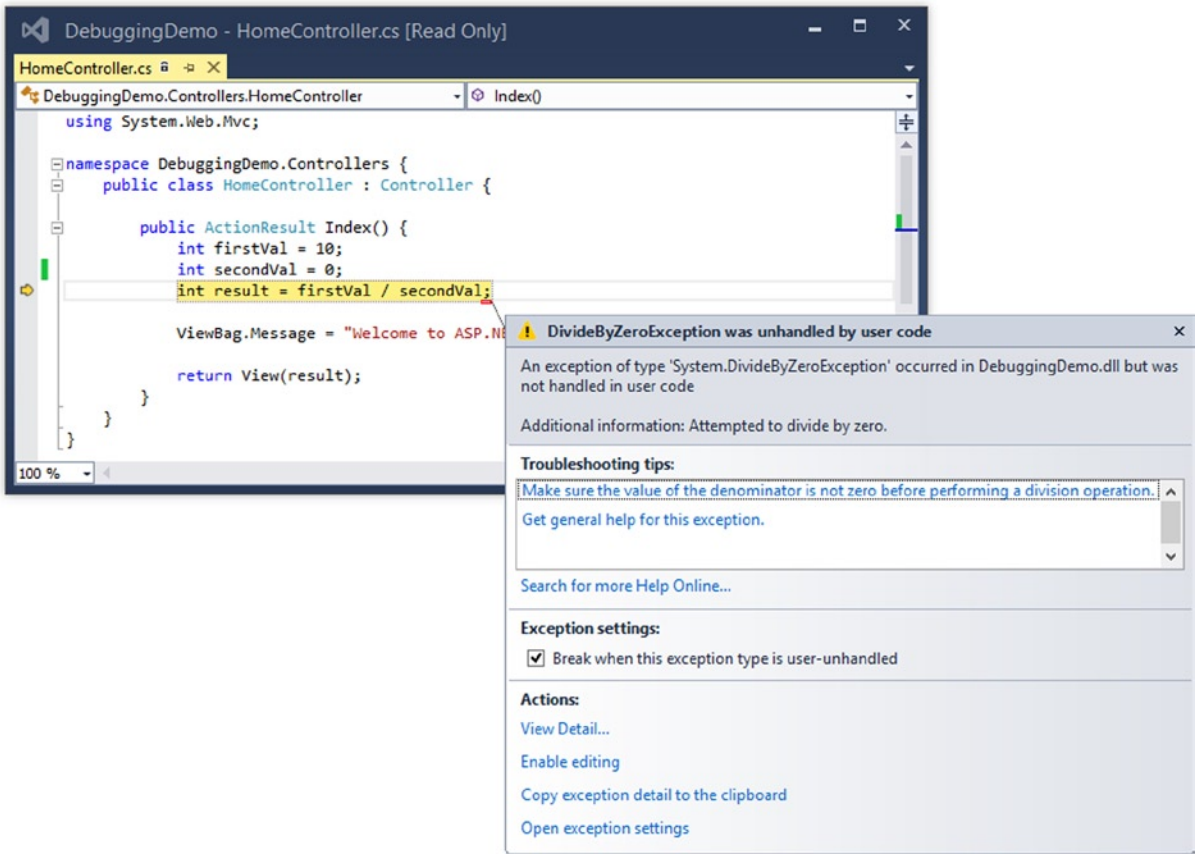

Figure 14-13. *The exception helper*

The exception helper gives you details of the exception. When the debugger breaks on an exception, you can inspect the application state and control execution, just as when a breakpoint is hit.

Using Edit and Continue

An interesting Visual Studio debugging feature is called *Edit and Continue*. When the debugger breaks, you can edit your code and then continue debugging. Visual Studio recompiles your application and re-creates the state of your application at the moment of the debugger break.

Enabling Edit and Continue

I need to make sure that Edit and Continue is enabled in two places:

- In the `Edit and Continue` section of the `Debugging` options (select `Options` from the Visual Studio `Tools` menu), make sure that `Enable Edit and Continue` is checked, as shown in Figure 14-14.

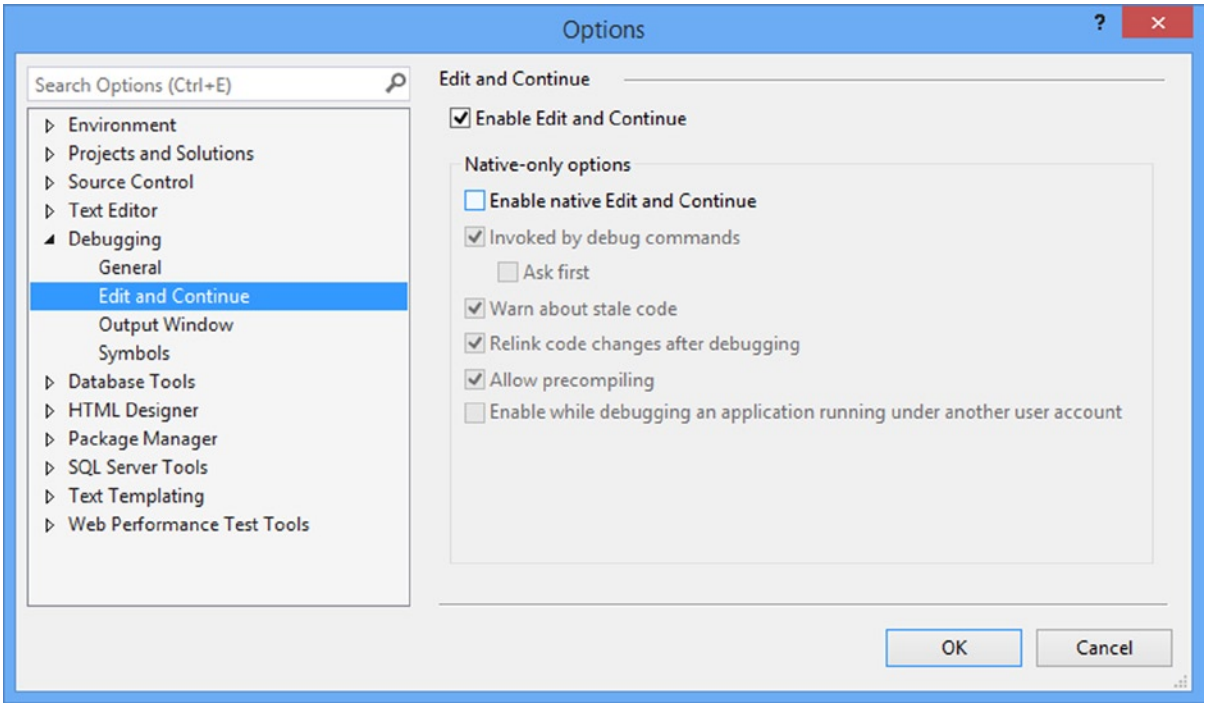

Figure 14-14. *Enabling Edit and Continue in the Options dialog box*

- In the project properties (select `DebuggingDemo Properties` from the Visual Studio `Project` menu), click the `Web` section and ensure that `Enable Edit and Continue` is checked, as shown in Figure 14-15.

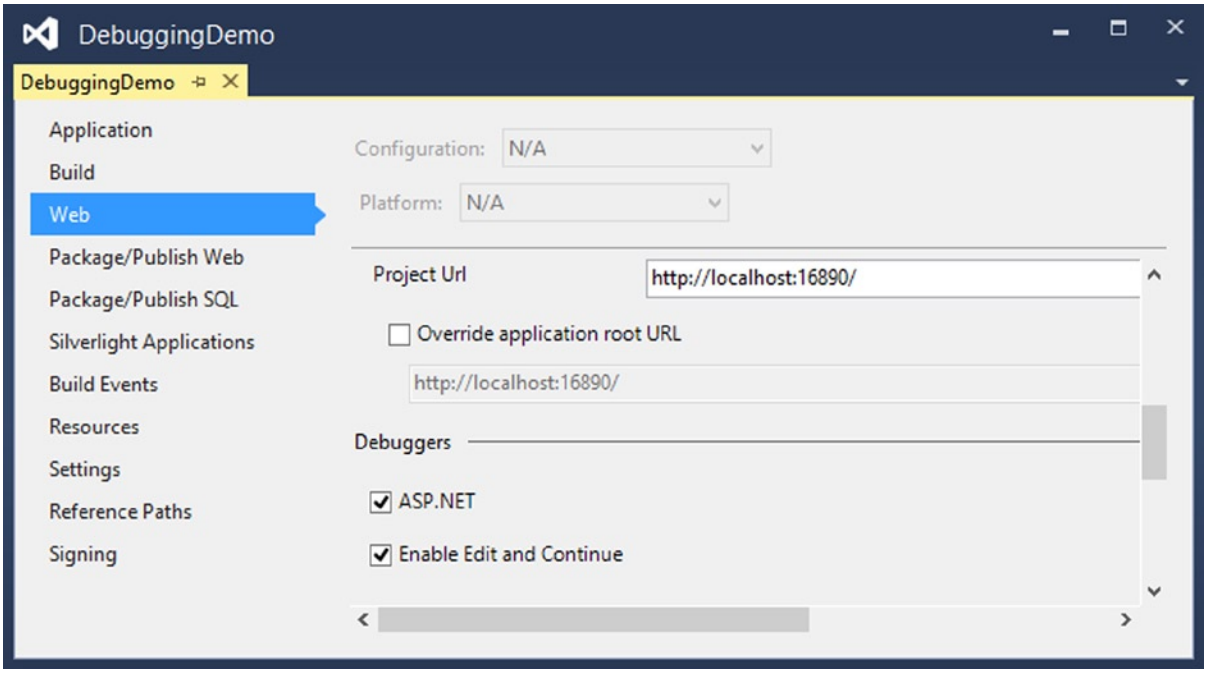

Figure 14-15. *Enabling Edit and Continue in the project properties*

Modifying the Project

The Edit and Continue feature is somewhat picky. There are some conditions under which it cannot work. One such condition is present in the `Index` action method of the `HomeController` class: the use of dynamic objects. To work around this, I have commented out the line that uses the view bag in the `HomeController.cs` class, as shown in Listing 14-6.

Listing 14-6. Removing the ViewBag Call from the Index Method in the HomeController.cs File

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 0;
            int result = firstVal / secondVal;

            // This statement has been commented out
            //ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
    }
}
```

I need to make a corresponding change in the Index.cshtml view, as shown in Listing 14-7.

Listing 14-7. Removing the ViewBag Call from the Index.cshtml File

```
@model int

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <title>Index</title>
</head>
<body>
    <!-- This element has been commented out -->
    <!--<h2 class="message">@ViewData["Message"]</h2>-->
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

Editing and Continuing

I am now ready for a demonstration of the Edit and Continue feature. Begin by selecting Start Debugging from the Visual Studio Debug menu. The application will be started with the debugger attached and run until it reaches the line where I perform the calculation in the Index method. The value of the second parameter is zero, which causes an exception to be thrown. At this point, the debugger halts execution, and the exception helper pops up (just like the one shown in Figure 14-13).

Click the Enable editing link in the exception helper window. In the code editor, change the expression that calculates the value for the result variable, as follows:

```
...
int result = firstVal / 2;
...
```

I have removed the reference to the secondVal variable and replaced it with a numeric literal value of 2. Now select Continue from the Visual Studio Debug menu to resume execution of the application. The new value is used to generate the value for the result variable, producing the output shown in Figure 14-16.

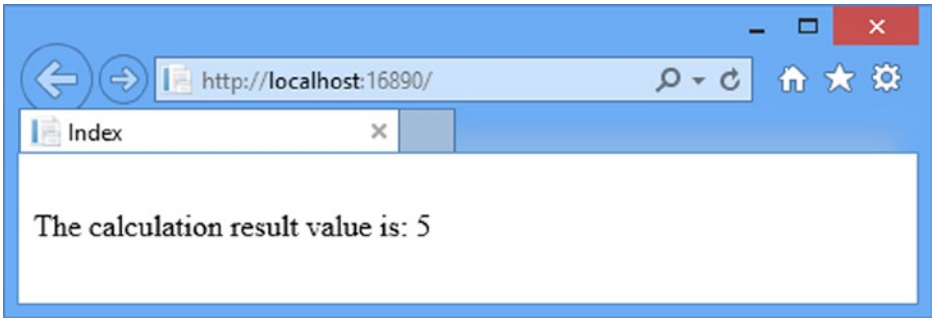

Figure 14-16. *The effect of correcting a bug using the Edit and Continue feature*

Take a moment to reflect on what happened here. I started the application with a bug in it: an attempt to divide a value by zero. The debugger detected the exception and stopped executing the program. I edited the code and then told the debugger to continue the execution. At this point, Visual Studio recompiled the application, restored its state and continued execution as normal using the new value. The browser received the rendered result, which reflected the new data value. Without Edit and Continue, I would have needed to stop the application, make a change, compile the application, and restart the debugger. I would then use the browser to repeat the steps that I took up to the moment of the debugger break. It is avoiding this last step that can be the most important. Complex bugs may require many steps through the application to re-create, and the ability to test potential fixes without needing to repeat those steps over and over can save the programmer's time and sanity.

Using Browser Link

Visual Studio 2013 includes a feature called *browser link* that allows you to view the application in multiple browsers simultaneously and have them all reload when you make a change. This feature is most useful once an application has stabilized and you are doing fit and finish work on the HTML and CSS that your views generate. (I'll explain why shortly.)

To use browser link, click on the small down arrow next to the selected browser on the Visual Studio toolbar and select `Browse With` from the menu, as shown in Figure 14-17.

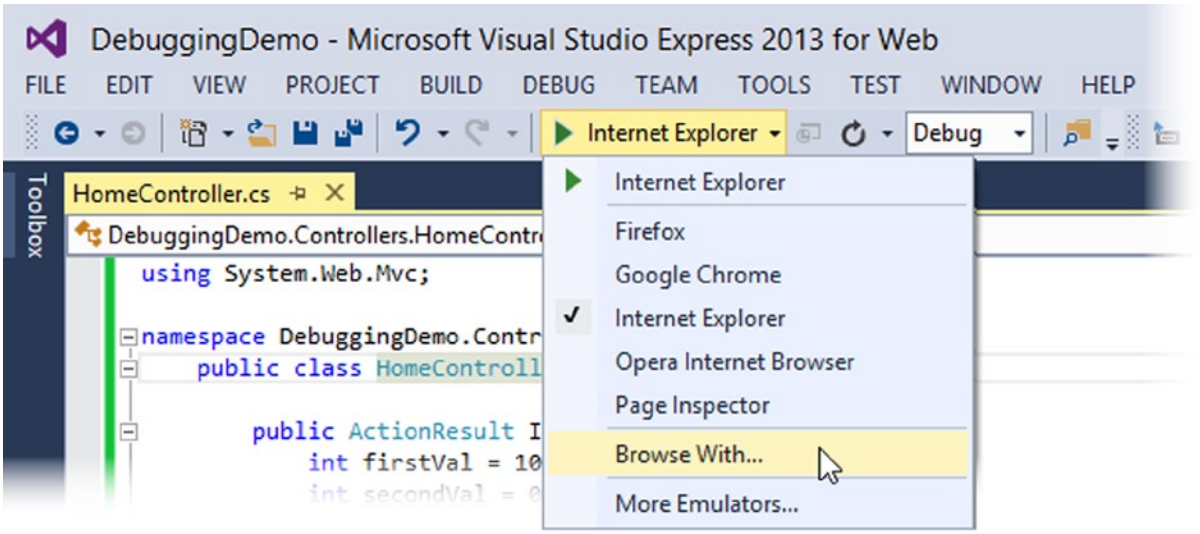

Figure 14-17. *Preparing to select the browsers used for the browser link feature*

The `Browse With` dialog window will appear. Hold the control key and select the browsers that you want to use. In Figure 14-18, you can see that I have chosen Internet Explorer and Chrome. You can also use this dialog to add new browsers (although Visual Studio is pretty good at detecting most of the mainstream ones).

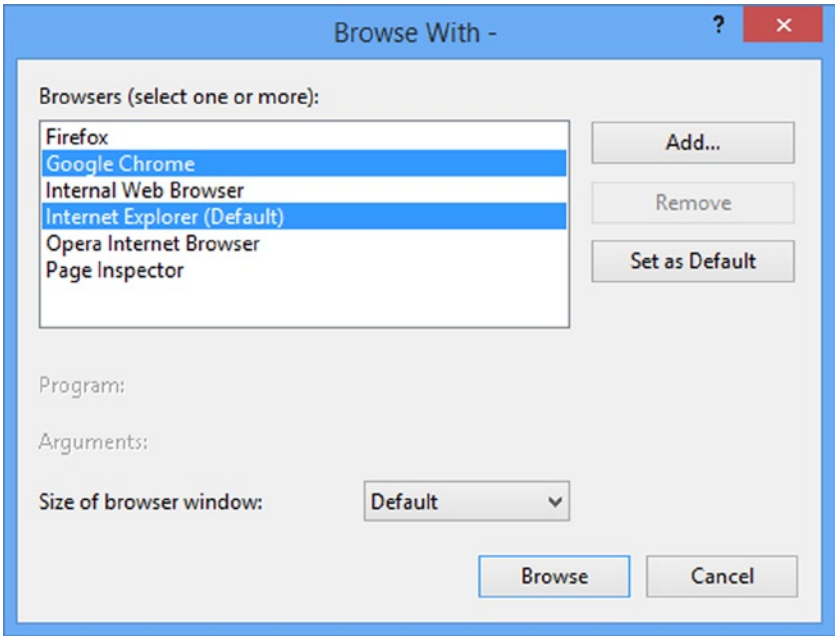

Figure 14-18. *Selecting multiple browsers*

Click the `Browse` button and Visual Studio will open the browsers you have selected and have each of them load the project URL. You can edit the code and views in the application and then update all of the browser windows by selecting Refresh Linked Browsers from the Visual Studio toolbar, as shown in Figure 14-19. The application will be compiled automatically so that you can see changes.

Figure 14-19. *Refreshing linked browsers*

This feature works by including some JavaScript in the HTML sent to the browser and it can be a nice way of developing iteratively. The reason that I recommend it only for working on views is that they are less likely to cause IIS to send HTTP error messages to the browser, which is what happens when there is an error in the code. The JavaScript code isn't added to error responses, which means that the link between Visual Studio and the browsers is lost. You have to start over using the `Browse With` menu. The browser link feature is a good idea, but the use of JavaScript is a problem. I use a similar tool called LiveReload (`http://livereload.com`) for my non-ASP.NET development and it provides a better approach because it uses browser plugins that are not affected by HTTP error messages. The value of Visual Studio browser link will be limited until Microsoft takes a similar approach.

Summary

In this chapter, I have shown you the structure of a Visual Studio MVC project and how the various parts fit together. I also touched on one of the most important characteristics of the MVC Framework: convention. These are topics that I will return to again and again in the chapters that follow, as I dig deeper into how the MVC Framework operates.