



Your First MVC Application

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, you'll create a simple data-entry application using the ASP.NET MVC Framework. I take things a step at a time so you can see how an ASP.NET MVC application is constructed. To keep things simple, I will skip over some of the technical details for the moment. But don't worry. If you are new to MVC, you will find plenty to keep you interested. Where I use something without explaining it, I provide a reference to the chapter in which you can find all the details.

Preparing Visual Studio

Visual Studio Express contains all of the features you need to create, test and deploy an MVC Framework application, but some of those features are hidden away until you ask for them. To enable all of the features, select **Expert Settings** from the Visual Studio Tools ► Settings menu.

■ **Tip** For some reason, Microsoft has decided that the top-level menus in Visual Studio should be all in uppercase, which means that the menu I referred to is really **TOOLS**. I think this is rather like shouting and I will capitalize menu names as **Tools** throughout this book.

Creating a New ASP.NET MVC Project

I am going to start by creating a new MVC Framework project in Visual Studio. Select **New Project** from the **File** menu to open the **New Project** dialog. If you select the **Web** templates in the **Visual C#** section, you will see the **ASP.NET Web Application** project template. Select this project type, as shown in [Figure 2-1](#).

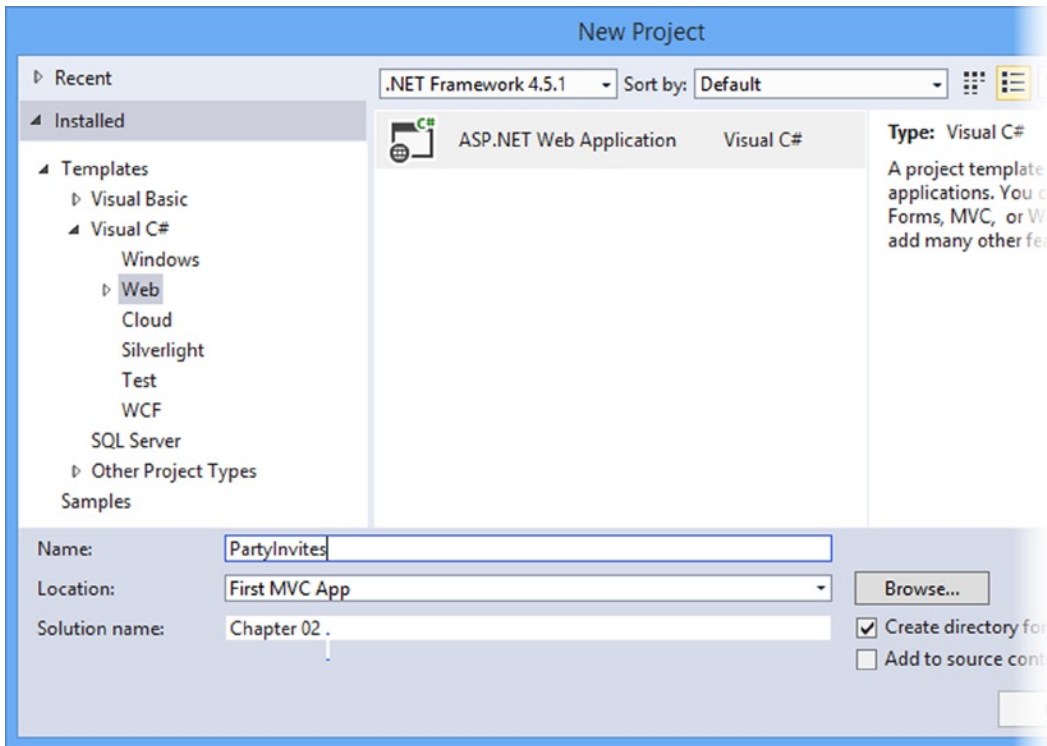


Figure 2-1. The Visual Studio ASP.NET Web Application project template

Set the name of the new project to **PartyInvites** and click the OK button to continue. You will see another dialog box, shown in Figure 2-2, which asks you to set the initial content for the ASP.NET project. This is part of the Microsoft initiative to better integrate the different parts of ASP.NET into a set of consistent tools and templates.

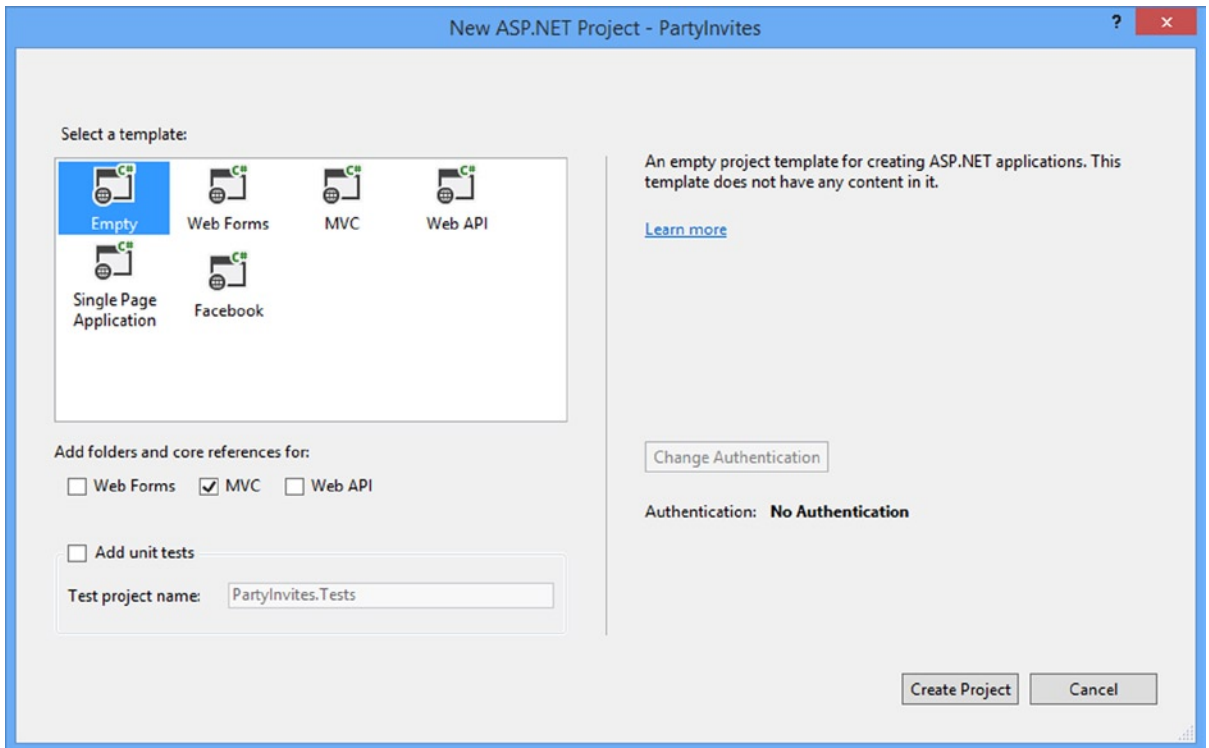


Figure 2-2. Selecting the initial project configuration

■ **Tip** Make sure you select version 4.5.1 of the .NET Framework at the top of the window. This is the latest version of .NET and is required for some of the advanced features that I describe in this book.

The templates create projects with different starting points and configurations for features such as authentication, navigation and visual themes. I am going to keep things simple: select the Empty option and check the MVC box in the *Add folders and core references* section, as shown in the figure. This will create a basic MVC project with minimal predefined content and will be the starting point that I use for all of the examples in this book. Click the OK button to create the new project.

■ **Note** The other project template options are intended to give you a more complete starting point for your ASP.NET projects. I don't like these templates because they encourage developers to treat some important features, such as authentication, as black boxes. My goal in this book is to give you the knowledge to understand and manage every aspect of your MVC application and, as a consequence, I use the Empty template for most of the examples in the book – the exception is in Chapter 14, where I show you the content that the MVC template adds to new projects.

Once Visual Studio creates the project, you will see a number of files and folders displayed in the **Solution Explorer** window, as shown in Figure 2-3. This is the default project structure for a new MVC project and you will soon understand the purpose of each of the files and folders that Visual Studio creates.

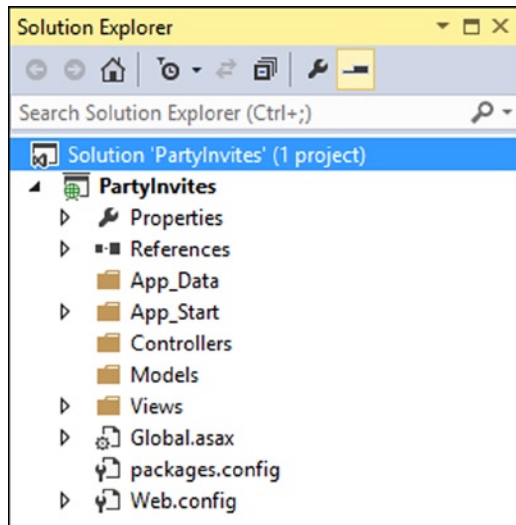


Figure 2-3. The initial file and folder structure of an MVC project

You can try to run the application now by selecting **Start Debugging** from the Debug menu (if it prompts you to enable debugging, just click the OK button). You can see the result in Figure 2-4. Because I started with the empty project template, the application does not contain anything to run, so the server generates a 404 Not Found Error.

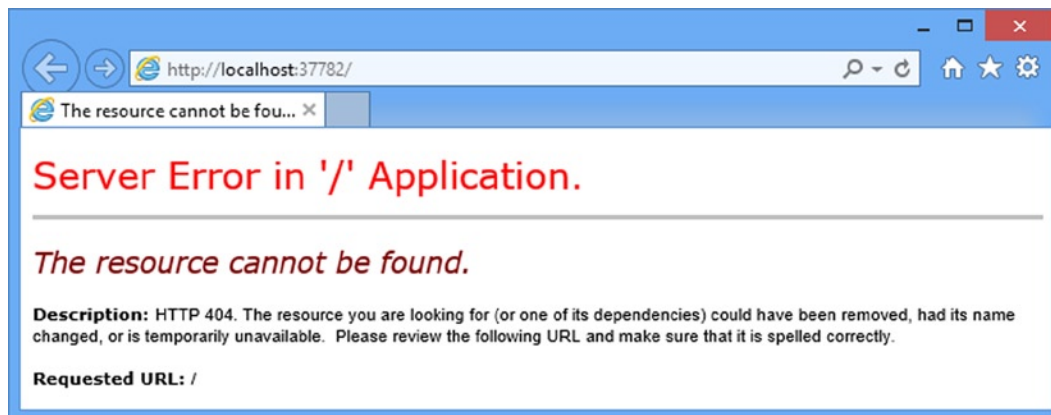


Figure 2-4. Trying to run an empty project

When you are finished, be sure to stop debugging by closing the browser window that shows the error, or by going back to Visual Studio and selecting **Stop Debugging** from the Debug menu.

As you have just seen, Visual Studio opens the browser to display the project. **The default browser is**, of course, Internet Explorer, but you can select any browser that you have installed by using the toolbar shown in Figure 2-5. As the figure shows, I have a range of browsers installed, which I find useful for testing web apps during development.

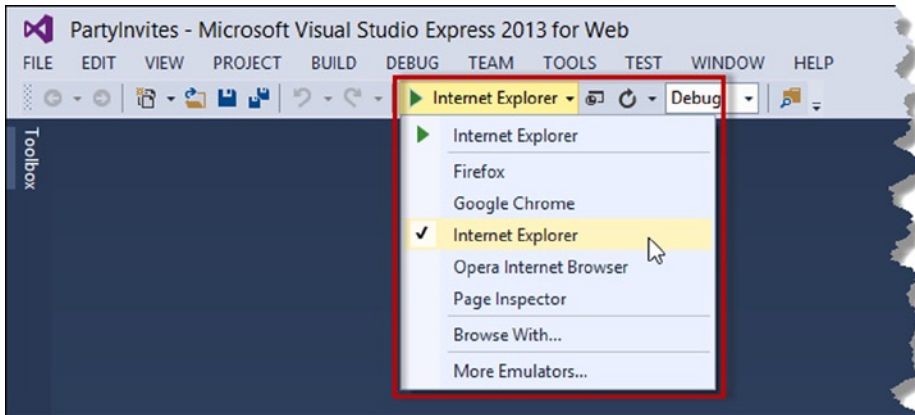


Figure 2-5. Changing the browser that Visual Studio uses to run the project

I will be using Internet Explorer 11 throughout this book, but that's just because I know that IE is so widely installed. Internet Explorer used to play fast and loose with web standards, but recent versions have been good at implementing the HTML5 standard. Google Chrome is also a good choice for development and I tend to use it for my own projects.

Adding the First Controller

In MVC architecture, incoming **requests are handled by controllers**. In ASP.NET MVC, controllers are **just C# classes** (usually inheriting from `System.Web.Mvc.Controller`, the framework's built-in controller base class).

Each public method in a controller is known as an *action method*, meaning you can invoke it from the Web via some URL to perform an action. The MVC convention is to put controllers in the Controllers folder, which Visual Studio created when it set up the project.

■ **Tip** You do not need to follow this or most other MVC conventions, but I recommend that you do—not least because it will help you make sense of the examples in this book.

To add a controller to the project, right-click the Controllers folder in the Visual Studio Solution Explorer window and choose Add and then Controller from the pop-up menus, as shown in Figure 2-6.

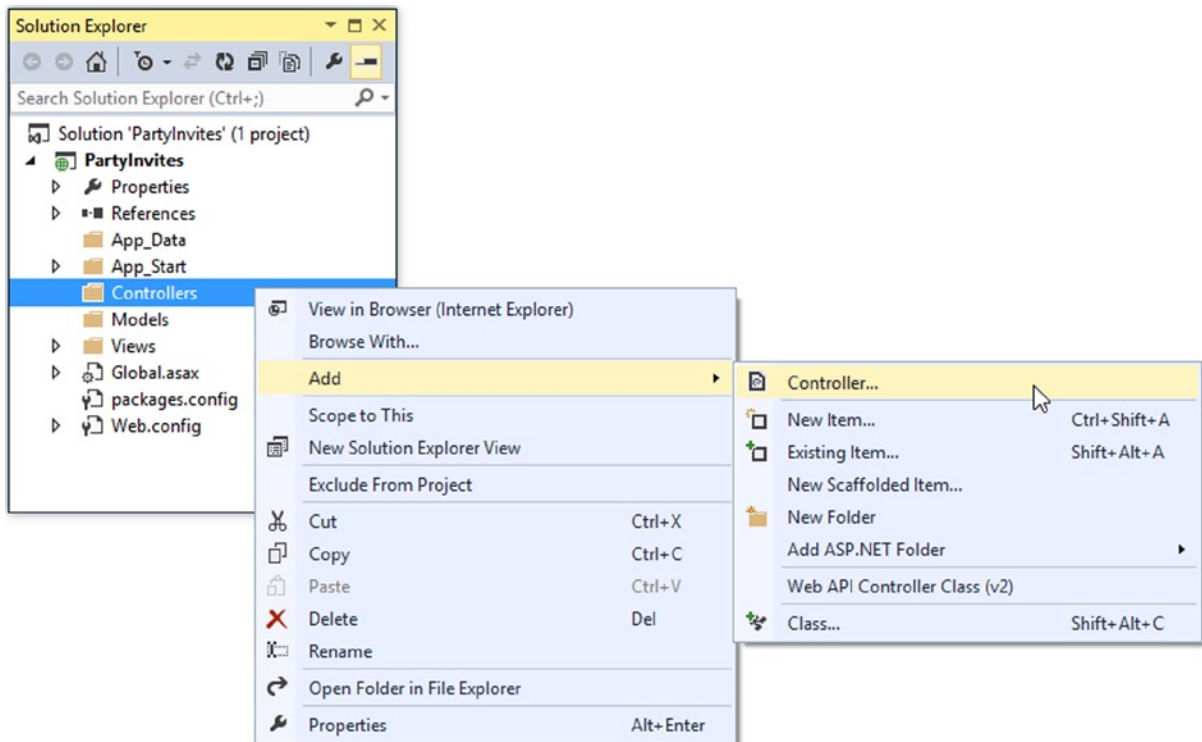


Figure 2-6. Adding a controller to the MVC project

When the Add Scaffold dialog appears, select the MVC 5 Controller – Empty option, as shown in Figure 2-7, and click the Add button.

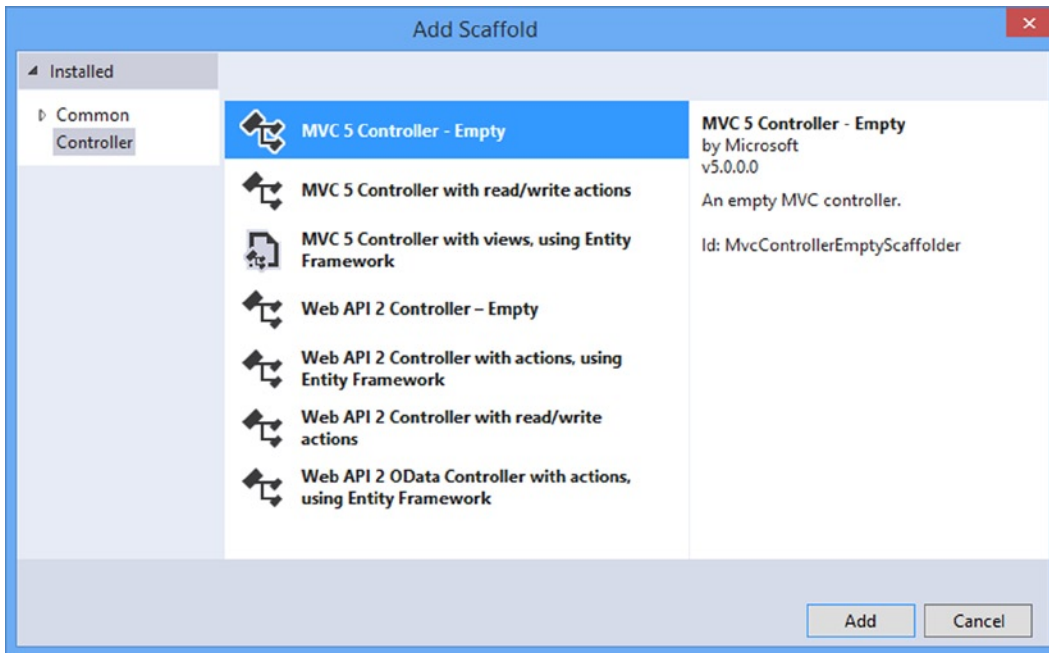


Figure 2-7. Selecting an empty controller from the Add Scaffold dialog

The Add Controller dialog will appear. Set the name to HomeController and click the Add button. There are several conventions represented in this name: names given to controllers should indicate their purpose; the default controller is called Home and controller names have the suffix Controller.

■ **Tip** If you have used earlier versions of Visual Studio to create MVC applications, then you will notice that the process is slightly different. Microsoft has changed the way that Visual Studio can populate a project with preconfigured classes and other items.

Visual Studio will create a new C# file in the Controllers folder called HomeController.cs and open it for editing. I have listed the default contents that Visual Studio puts into the class file in Listing 2-1. You can see that the class is called HomeController and it is derived from the Controller class, which is found in the System.Web.Mvc namespace.

Listing 2-1. The Default Contents of the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
```

```

        public ActionResult Index() {
            return View();
        }
    }
}

```

A good way of getting started with MVC is to make a couple of simple changes to the controller class. Edit the code in the `HomeController.cs` file so that it matches Listing 2-2. I have highlighted the statements that have changed so they are easier to see.

Listing 2-2. Modifying the `HomeController.cs` File

```

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public string Index() {
            return "Hello World";
        }
    }
}

```

These changes don't have a dramatic effect, but they make for a nice demonstration. I have changed the action method called `Index` so that it returns the string "Hello World". Run the project again by selecting **Start Debugging** from the Visual Studio Debug menu. The browser will display the result of the `Index` action method, as shown in Figure 2-8.

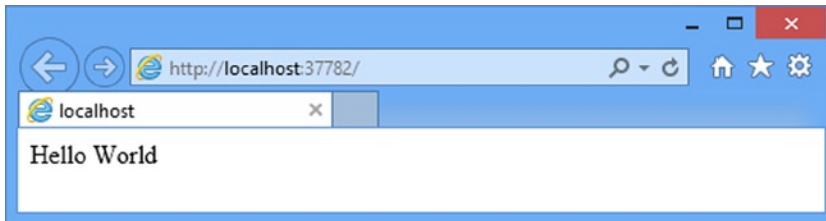


Figure 2-8. The output from the controller action method

■ **Tip** Notice that Visual Studio has directed the browser to port 37782. You will almost certainly see a different port number in the URL that your browser requests because Visual Studio allocates a random port when the project is created. If you look in the Windows task bar notification area, you will find an icon for IIS Express. This is a cut-down version of the full IIS application server which is included with Visual Studio and is used to deliver ASP.NET content and services during development. I'll show you how to deploy an MVC project into a production environment in Chapter 13.

Understanding Routes

As well as models, views, and controllers, MVC applications use the ASP.NET *routing system*, which decides how URLs map to controllers and actions. When Visual Studio creates the MVC project, it adds some default routes to get us started. You can request any of the following URLs, and they will be directed to the Index action on the HomeController:

- /
- /Home
- /Home/Index

So, when a browser requests <http://yoursite/> or <http://yoursite/Home>, it gets back the output from HomeController's Index method. You can try this yourself by changing the URL in the browser. At the moment, it will be <http://localhost:37782/>, except that the port part may be different. If you append /Home or /Home/Index to the URL and hit return, you will see the same Hello World result from the MVC application.

This is a good example of benefiting from following MVC conventions. In this case, the convention is that I will have a controller called HomeController and that it will be the starting point for my MVC application. The default routes that Visual Studio creates for a new project assume that I will follow this convention. And since I *did* follow the convention, I automatically got support for the URLs in the preceding list.

If I had *not* followed the convention, I would need to modify the routes to point to whatever controller I had created instead. For this simple example, the default configuration is all I need.

■ **Tip** You can see and edit your routing configuration by opening the RouteConfig.cs file in the App_Start folder. I explain what the entries in this file do in Chapters 16 and 17.

Rendering Web Pages

The output from the previous example wasn't HTML—it was just the string "Hello World". To produce an HTML response to a browser request, I need a *view*.

Creating and Rendering a View

The first thing I need to do is modify my Index action method, as shown in Listing 2-3.

Listing 2-3. Modifying the Controller to Render a View in the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ViewResult Index() {
            return View();
        }
    }
}
```

The changes in Listing 2-3 are shown in bold. When I return a `ViewResult` object from an action method, I am instructing MVC to *render* a view. I create the `ViewResult` by calling the `View` method with no parameters. This tells MVC to render the *default* view for the action.

If you run the application at this point, you can see the MVC Framework trying to find a default view to use, as shown in the error message displayed in Figure 2-9.

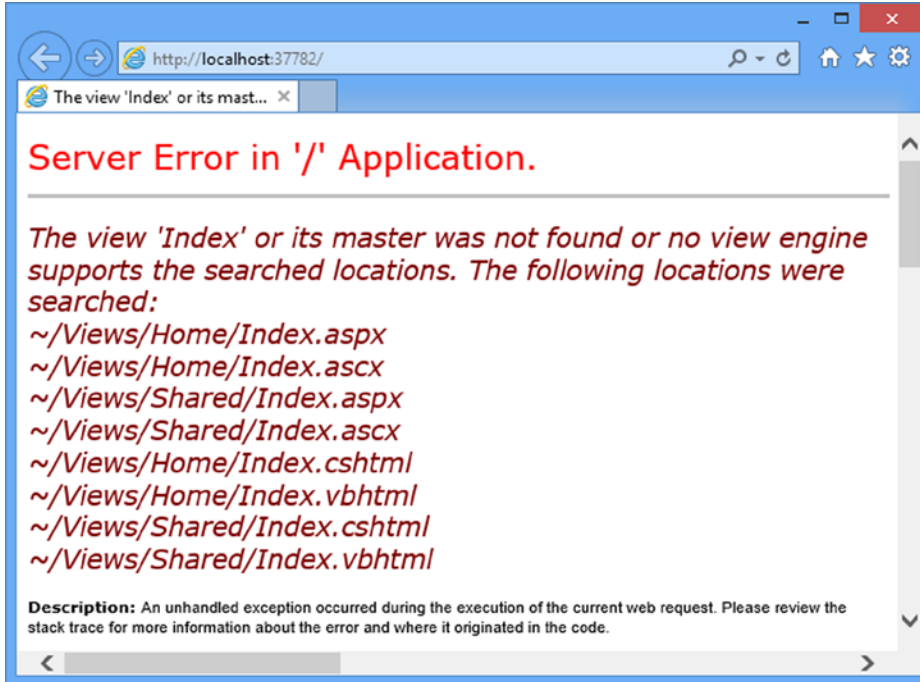


Figure 2-9. The MVC Framework trying to find a default view

This error message is quite helpful. It explains not only that MVC could not find a view for the action method, but it shows where it looked. This is another nice illustration of an MVC convention: views are associated with action methods by a naming convention. The action method is called `Index` and the controller is called `Home` and you can see from Figure 2-9 that MVC is trying to find different files in the `Views` folder that have that name.

The simplest way to create a view is to ask Visual Studio to do it for you. Right-click anywhere in the definition of the `Index` action method in code editor window for the `HomeController.cs` file and select `Add View` from the pop-up menu, as shown in Figure 2-10.

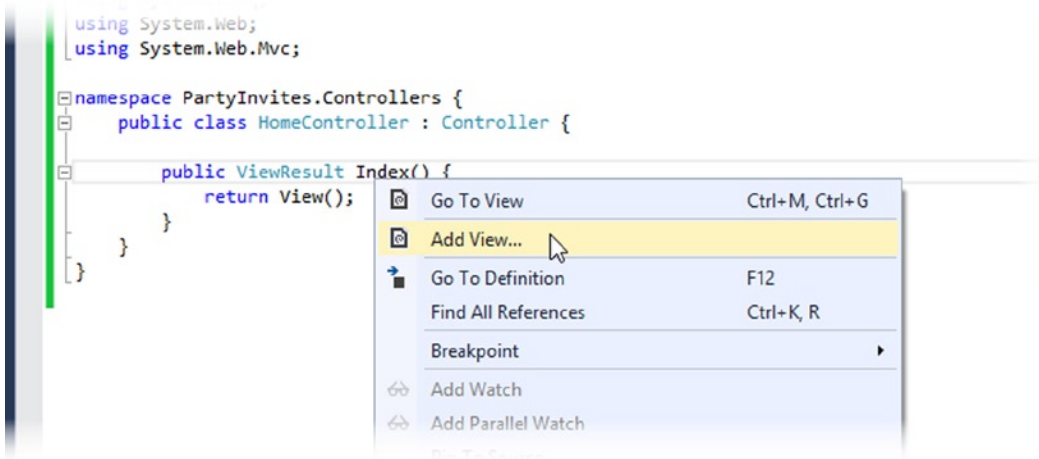


Figure 2-10. Asking Visual Studio to create a view for an action method

Visual Studio displays the Add View dialog, which allows you to configure the initial contents of the view file that will be created. Set View Name to Index (the name of the action method that the view will be associated with—another convention), set Template to Empty (without model), and leave the Create as a partial view and Use a layout page boxes unchecked, as shown in Figure 2-11. Don't worry about what all of these options mean at the moment—I'll explain all of the details in later chapters. Click the Add button to create the new view file.

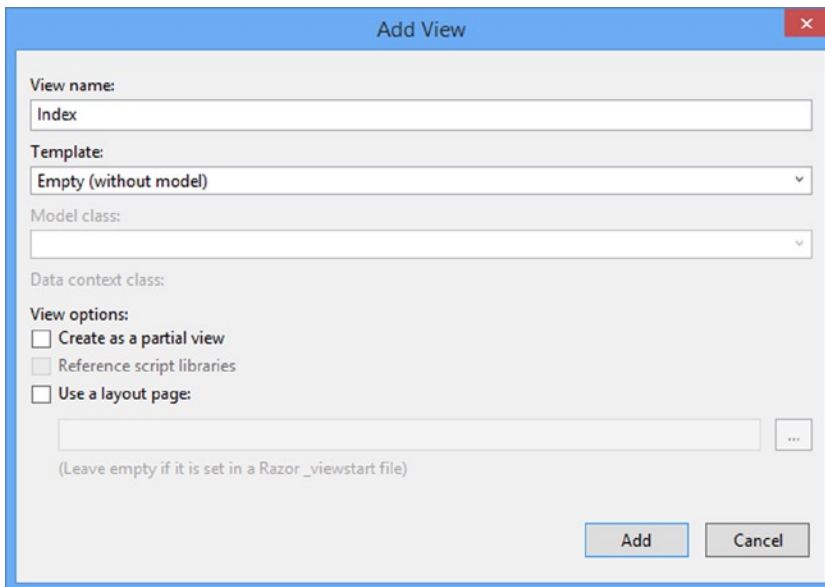


Figure 2-11. Configuring the initial contents of the view file

Visual Studio will create a file called `Index.cshtml` in the `Views/Home` folder. If this isn't the effect you achieve, then delete the file you created and try again. This is another MVC Framework convention: views are placed in the `Views` folder, organized in folders that correspond to the name of the controller they are associated with.

■ **Tip** The `.cshtml` file extension denotes a C# view that will be processed by Razor. Early versions of MVC relied on the ASPX view engine, for which view files have the `.aspx` extension.

The effect of the values I told you to enter into the `Add View` dialog tell Visual Studio to create the most basic view, the contents of which are shown in Listing 2-4.

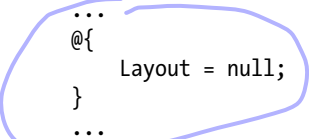
Listing 2-4. The Initial Contents of the `Index.cshtml` File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

Visual Studio opens the `Index.cshtml` file for editing. You'll see that this file contains mostly HTML. The exception is the part that looks like this:



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

This is an expression that will be interpreted by the Razor view engine, which processes the contents of views and generates HTML that is sent to the browser. This is a simple Razor expression and it tells Razor that I chose not to use a layout, which is like a template for the HTML that will be sent to the browser (and which I describe in Chapter 5). I am going to ignore Razor for the moment and come back to it later. Make the addition to the `Index.cshtml` file that is shown in bold in Listing 2-5.

Listing 2-5. Adding to the View HTML in the `Index.cshtml` File

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

```

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Hello World (from the view)
    </div>
</body>
</html>

```

The addition displays another simple message. Select **Start Debugging** from the **Debug** menu to run the application and test the view. You should see something similar to Figure 2-12.

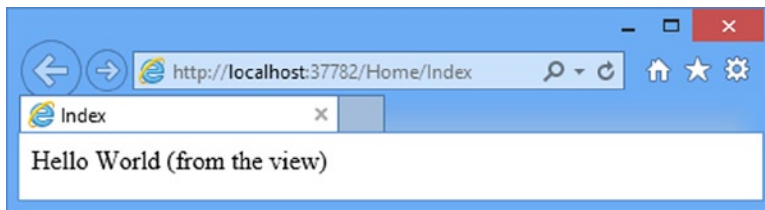


Figure 2-12. Testing the view

When I first edited the `Index` action method, it returned a string value. This meant that MVC did nothing except pass the string value as is to the browser. Now that the `Index` method returns a `ViewResult`, the MVC Framework renders a view and returns the HTML it produces. I didn't tell MVC which view should be used, so it used the naming convention to find one automatically. The convention is that the view has the name of the action method and is contained in a folder named after the controller: `/Views/Home/Index.cshtml`.

I can return other results from action methods besides strings and `ViewResult` objects. For example, if I return a `RedirectResult`, the browser will be redirected to another URL. If I return an `HttpUnauthorizedResult`, I force the user to log in. These objects are collectively known as *action results*, and they are all derived from the `ActionResult` class. The action result system lets us encapsulate and reuse common responses in actions. I'll tell you more about them and show more complex uses in Chapter 17.

Adding Dynamic Output

The whole point of a web application platform is to construct and display *dynamic* output. In MVC, it is the **controller's job to construct some data and pass it to the view, which is responsible for rendering it to HTML.**

One way to pass data from the controller to the view is by using the `ViewBag` object, which is a member of the Controller base class. **`ViewBag` is a dynamic object to which you can assign arbitrary properties,** making those values available in whatever view is subsequently rendered. Listing 2-6 demonstrates passing some simple dynamic data in this way in the `HomeController.cs` file.

Listing 2-6. Setting Some View Data in the HomeController.cs File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View();
        }
    }
}
```

I provide data for the view when I assign a value to the `ViewBag.Greeting` property. The `Greeting` property didn't exist until the moment I assigned the value—this allows me to pass data from the controller to the view in a free and fluid manner, without having to define classes ahead of time. I refer to the `ViewBag.Greeting` property again in the view to get the data value, as illustrated in Listing 2-7, which shows the corresponding change to the `Index.cshtml` file.

Listing 2-7. Retrieving a ViewBag Data Value in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
    </div>
</body>
</html>
```

The addition to Listing 2-7 is a **Razor expression**. When I call the `View` method in the controller's `Index` method, the MVC framework locates the `Index.cshtml` view file and asks the Razor view engine to parse the file's content. Razor looks for expressions like the one I added in the listing and processes them. In this example, processing the expression means inserting the value assigned to the `ViewBag.Greeting` property in the action method into the view.

There's nothing special about the property name `Greeting`; you could replace this with any property name and it would work the same, just as long as the name you use in the controller matches the name you use in the view. You can pass multiple data values from your controller to the view by assigning values to more than one property. You can see the effect of these changes by starting the project, as shown in Figure 2-13.

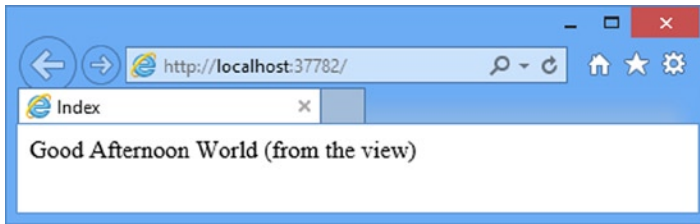


Figure 2-13. A dynamic response from MVC

Creating a Simple Data-Entry Application

In the rest of this chapter, I will explore more of the basic MVC features by building a simple data-entry application. I am going to pick up the pace in this section. My goal is to demonstrate MVC in action, so I will skip over some of the explanations as to how things work behind the scenes. But don't worry, I'll revisit these topics in depth in later chapters.

Setting the Scene

Imagine that a friend has decided to host a New Year's Eve party and that she has asked me to create a web app that allows her invitees to electronically RSVP. She has asked for four key features:

- A home page that shows information about the party
- A form that can be used to RSVP
- Validation for the RSVP form, which will display a thank-you page
- RSVPs e-mailed to the party host when complete

In the following sections, I will build up the MVC project I created at the start of the chapter and add these features. I can check the first item off the list by applying what I covered earlier and add some HTML to my existing view to give details of the party. Listing 2-8 shows the additions I made to the Views/Home/Index.cshtml file.

Listing 2-8. Displaying Details of the Party in the Index.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
        <p>We're going to have an exciting party.<br />
```

```

    (To do: sell it better. Add pictures or something.)
  </p>
</div>
</body>
</html>

```

I am on my way. If you run the application, you'll see the details of the party—well, the placeholder for the details, but you get the idea—as shown in Figure 2-14.

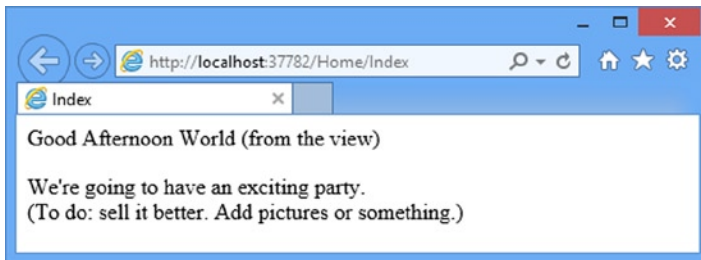


Figure 2-14. Adding to the view HTML

Designing a Data Model

In MVC, the *M* stands for *model*, and it is the most important part of the application. The model is the representation of the real-world objects, processes, and rules that define the subject, known as the *domain*, of the application. The model, often referred to as a *domain model*, contains the C# objects (known as *domain objects*) that make up the universe of the application and the methods that manipulate them. The views and controllers expose the domain to the clients in a consistent manner and a well-designed MVC application starts with a well-designed model, which is then the focal point as controllers and views are added.

I don't need a complex model for the PartyInvites application because it is such a simple application and I need to create just one domain class which I will call GuestResponse. This object will be responsible for storing, validating, and confirming an RSVP.

Adding a Model Class

The MVC convention is that the classes that make up a model are placed inside the Models folder, which Visual Studio created as part of the initial project setup. Right-click Models in the Solution Explorer window and select Add followed by Class from the pop-up menus. Set the file name to GuestResponse.cs and click the Add button to create the class. Edit the contents of the class to match Listing 2-9.

■ **Tip** If you don't have a Class menu item, then you probably left the Visual Studio debugger running. Visual Studio restricts the changes you can make to a project while it is running the application.

Listing 2-9. The GuestResponse Domain Class Defined in the GuestResponse.cs File

```
namespace PartyInvites.Models {
    public class GuestResponse {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

■ **Tip** You may have noticed that the `WillAttend` property is a *nullable* `bool`, which means that it can be `true`, `false`, or `null`. I explain the rationale for this in the Adding Validation section later in the chapter.

Linking Action Methods

One of my application goals is to include an RSVP form, so I need to add a link to it from my `Index.cshtml` view, as shown in Listing 2-10.

Listing 2-10. Adding a Link to the RSVP Form in the `Index.cshtml` File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
        <p>We're going to have an exciting party.<br />
        (To do: sell it better. Add pictures or something.)
        </p>
        @Html.ActionLink("RSVP Now", "RsvpForm")
    </div>
</body>
</html>
```

`Html.ActionLink` is an HTML helper method. The MVC Framework comes with a collection of built-in helper methods that are convenient for rendering HTML links, text inputs, checkboxes, selections, and other kinds of content. **The `ActionLink` method takes two parameters:** the first is the text to display in the link, and the second is the action to perform when the user clicks the link. I explain the complete set of HTML helper methods in Chapters 21-23. You can see the link that the helper creates by starting the project, as shown in Figure 2-15.

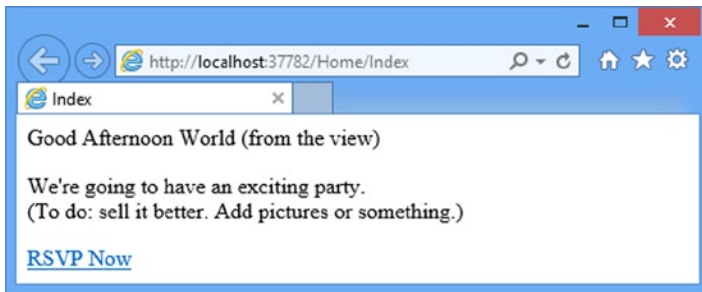


Figure 2-15. Adding a link to the view

If you roll your mouse over the link in the browser, you will see that the link points to <http://yourserver/Home/RsvpForm>. The `Html.ActionLink` method has inspected the application's URL routing configuration and determined that `/Home/RsvpForm` is the URL for an action called `RsvpForm` on a controller called `HomeController`.

■ **Tip** Notice that, unlike traditional ASP.NET applications, MVC URLs do not correspond to physical files. Each action method has its own URL, and MVC uses the ASP.NET routing system to translate these URLs into actions.

Creating the Action Method

You will see a 404 Not Found error if you click the link. That's because I have not yet created the action method that corresponds to the `/Home/RsvpForm` URL. I do this by adding a method called `RsvpForm` to the `HomeController` class, as shown in Listing 2-11.

Listing 2-11. Adding a New Action Method in the `HomeController.cs` File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View();
        }

        public ActionResult RsvpForm() {
            return View();
        }
    }
}
```

Adding a Strongly Typed View

I am going to add a view for the `RsvpForm` action method, but in a slightly different way—I am going to create a *strongly typed* view. A strongly typed view is intended to render a specific domain type, and if I specify the type I want to work with (`GuestResponse` in this case), MVC can create some helpful shortcuts to make it easier.

Caution Make sure your MVC project is compiled before proceeding. If you have created the `GuestResponse` class but not compiled it, MVC won't be able to create a strongly typed view for this type. To compile your application, select **Build Solution** from the Visual Studio Build menu.

Right-click the `RsvpForm` method in the code editor and select **Add View** from the pop-up menu to open the **Add View** dialog window. Ensure that the **View Name** is set as `RsvpForm`, set **Template** to **Empty** and select `GuestResponse` from the drop-down list for the **Model Class** field. Leave the **View Options** boxes unchecked, as shown in Figure 2-16.

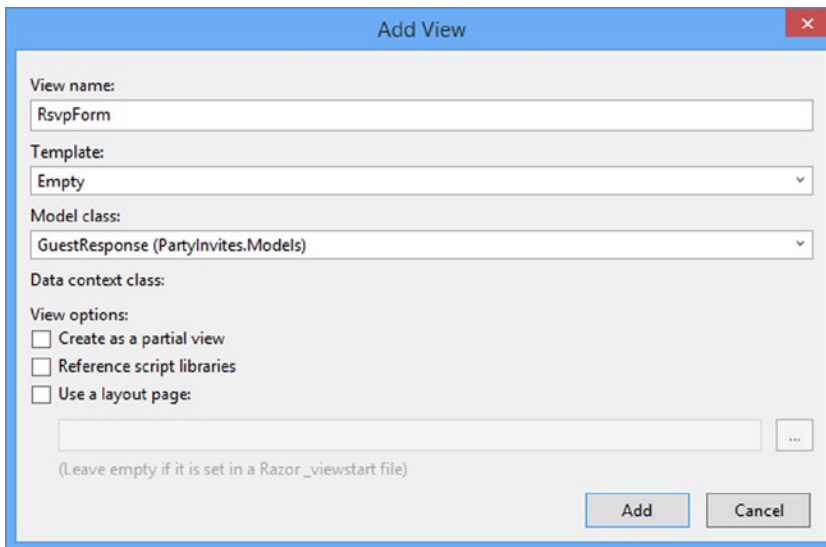


Figure 2-16. Adding a new view to the project

Click the **Add** button and Visual Studio will create a new file called `RsvpForm.cshtml` in the `Views/Home` folder and open it for editing. You can see the initial contents in Listing 2-12. This is another skeletal HTML file, but it contains a `@model` Razor expression. As you will see in a moment, this is the key to a strongly typed view and the convenience it offers.

Listing 2-12. The Initial Contents of the `RsvpForm.cshtml` File

@model PartyInvites.Models.GuestResponse

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

```

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
    </div>
</body>
</html>

```

■ **Tip** The options that you select and check when you create a view determine the initial content of a view file, but that's all. You can change from regular to strongly typed views, for example, just by adding or removing the `@model` directive in the code editor.

Building the Form

Now that I have created the strongly typed view, I can build out the contents of `RsvpForm.cshtml` to make it into an HTML form for editing `GuestResponse` objects, as shown in Listing 2-13.

Listing 2-13. Creating a Form View in the `RsvpForm.cshtml` File

```

@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there",
                    Value = bool.TrueString},

```

```

        new SelectListItem() {Text = "No, I can't come",
                               Value = bool.FalseString}
    }, "Choose an option")
</p>
<input type="submit" value="Submit RSVP" />
}
</body>
</html>

```

For each property of the `GuestResponse` model class, I use an HTML helper method to render a suitable HTML input control. These methods let you select the property that the input element relates to using a lambda expression, like this:

```

...
@Html.TextBoxFor(x => x.Phone)
...

```

The `Html.TextBoxFor` helper method generates the HTML for an input element, sets the type parameter to text, and sets the id and name attributes to `Phone` (the name of the selected domain class property) like this:

```
<input id="Phone" name="Phone" type="text" value="" />
```

This handy feature works because the `RsvpForm` view is strongly typed, and I have told MVC that `GuestResponse` is the type that I want to render with this view. This provides the HTML helper methods with the information they need to understand which data type I want to read properties from via the `@model` expression.

Don't worry if you aren't familiar with C# lambda expressions. I provide an overview in Chapter 4, but an alternative to using lambda expressions is to refer to the name of the model type property as a string, like this:

```

...
@Html.TextBox("Email")
...

```

I find that the lambda expression technique prevents me from mistyping the name of the model type property, because Visual Studio IntelliSense pops up and lets me pick the property automatically, as shown in Figure 2-17.

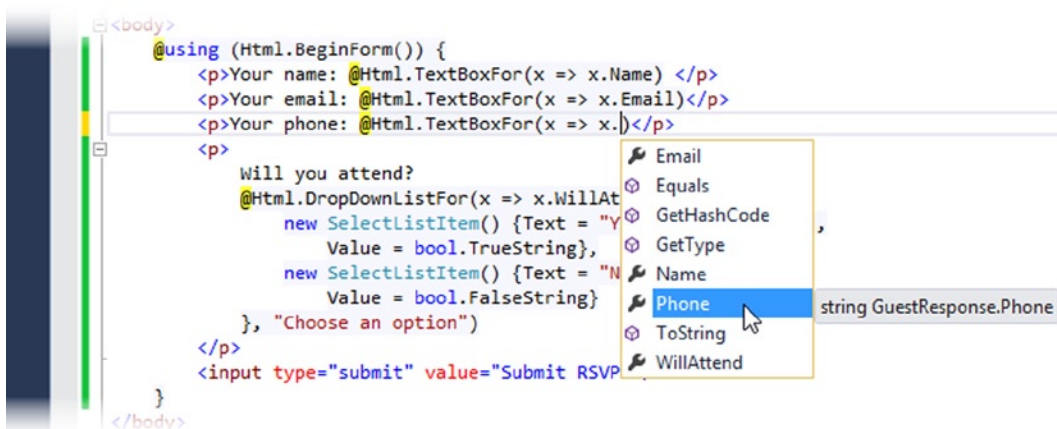


Figure 2-17. Visual Studio IntelliSense for lambda expressions in HTML helper methods

Another convenient helper method is `Html.BeginForm`, which generates an HTML form element configured to post back to the action method. Because I have not passed any arguments to the helper method, it assumes I want to post back to the same URL that the HTML document was requested from. A neat trick is to wrap this in a C# `using` statement, like this:

```
...
@using (Html.BeginForm()) {
    ...form contents go here...
}
...
```

Normally, when applied like this, the `using` statement ensures that an object is disposed of when it goes out of scope. It is commonly used for database connections, for example, to make sure that they are closed as soon as a query has completed. (This application of the `using` keyword is different from the kind that brings classes in a namespace into scope in a class.)

Instead of disposing of an object, the `Html.BeginForm` helper closes the HTML form element when it goes out of scope. This means that the `Html.BeginForm` helper method creates both parts of a form element, like this:

```
<form action="/Home/RsvpForm" method="post">
    ...form contents go here...
</form>
```

Don't worry if you are not familiar with disposing of C# objects. The point here is to demonstrate how to create a form using the HTML helper method.

Setting the Start URL

Visual Studio will, in an effort to be helpful, make the browser request a URL based on the view that is currently being edited. This is a hit-and-miss feature because it doesn't work when you are editing other kinds of file and because you can't just jump in at any point in most complex web apps.

To set a fixed URL for the browser to request, select `PartyInvites Properties` from the Visual Studio Project menu, select the `Web` section and check the `Specific Page` option in the `Start Action` category, as shown in Figure 2-18. You don't have to enter a value into the field—Visual Studio will request the default URL for the project, which will be directive to the `Index` action method on the `Home` controller. (I show you how to use the URL routing system to change the default mapping in Chapters 15 and 16).

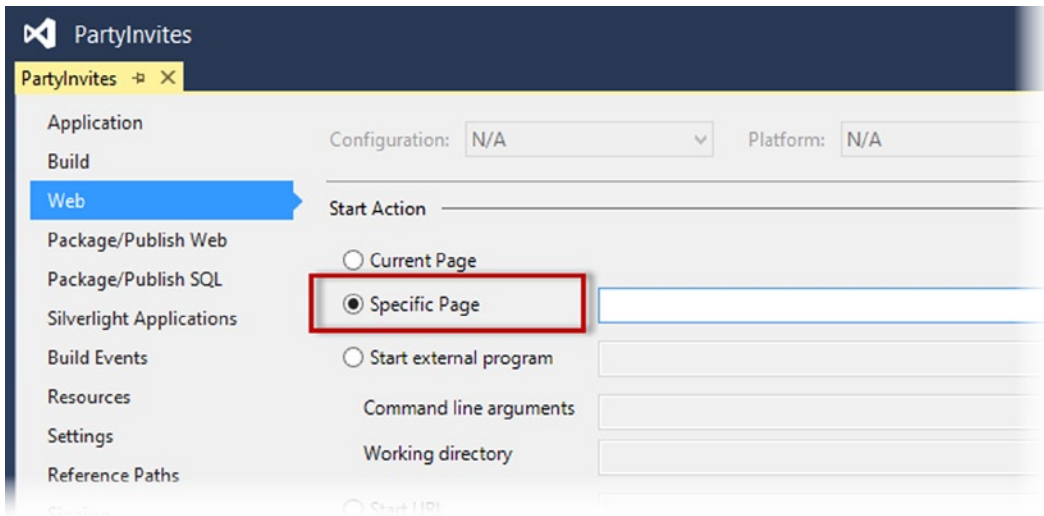


Figure 2-18. Setting the default start URL for the project

You can see the form in the `RsvpForm` view when you run the application and click the `RSVP Now` link. Figure 2-19 shows the result.

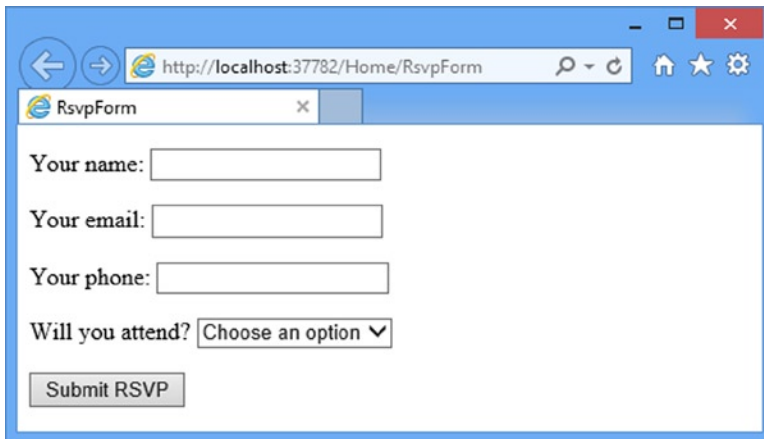


Figure 2-19. The `RsvpForm` view

Handling Forms

I have not yet told MVC what I want to do when the form is posted to the server. As things stand, clicking the `Submit RSVP` button just clears any values you have entered into the form. That is because the form posts back to the `RsvpForm` action method in the `Home` controller, which just tells MVC to render the view again.

■ **Note** You might be surprised that the input data is lost when the view is rendered again. If so, you have probably been developing applications with ASP.NET Web Forms, which automatically preserves data in this situation. I will show you how to achieve the same effect with MVC shortly.

To receive and process submitted form data, I am going to use a clever feature. I will add a second `RsvpForm` action method in order to create the following:

- *A method that responds to HTTP GET requests:* A GET request is what a browser issues normally each time someone clicks a link. This version of the action will be responsible for displaying the initial blank form when someone first visits `/Home/RsvpForm`.
- *A method that responds to HTTP POST requests:* By default, forms rendered using `Html.BeginForm()` are submitted by the browser as a POST request. This version of the action will be responsible for receiving submitted data and deciding what to do with it.

Handling GET and POST requests in separate C# methods helps to keep my controller code tidy, since the two methods have different responsibilities. Both action methods are invoked by the same URL, but MVC makes sure that the appropriate method is called, based on whether I am dealing with a GET or POST request. Listing 2-14 shows the changes I applied to the `HomeController` class.

Listing 2-14. Adding an Action Method to Support POST Requests in the `HomeController.cs` File

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

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View();
        }

        [HttpGet]
        public ActionResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public ActionResult RsvpForm(GuestResponse guestResponse) {
            // TODO: Email response to the party organizer
            return View("Thanks", guestResponse);
        }
    }
}
```


I have added the `HttpGet` attribute to my existing `RsvpForm` action method. This tells MVC that this method should be used only for GET requests. I then added an overloaded version of `RsvpForm`, which takes a `GuestResponse` parameter and applies the `HttpPost` attribute. The attribute tells MVC that the new method will deal with POST requests. I also imported the `PartyInvites.Models` namespace—this is just so I can refer to the `GuestResponse` model type without needing to qualify the class name. I explain how these additions to the listing work in the following sections.

Using Model Binding

The first overload of the `RsvpForm` action method renders the same view as before—the `RsvpForm.cshtml` file—to generate the form shown in Figure 2-18.

The second overload is more interesting because of the parameter, but given that the action method will be invoked in response to an HTTP POST request, and that the `GuestResponse` type is a C# class, how are the two connected?

The answer is *model binding*, an extremely useful MVC feature whereby incoming data is parsed and the key/value pairs in the HTTP request are used to populate properties of domain model types. This process is the opposite of using the HTML helper methods; that is, when creating the form data to send to the client, I generated HTML input elements where the values for the `id` and `name` attributes were derived from the model class property names.

In contrast, with model binding, the names of the input elements are used to set the values of the properties in an instance of the model class, which is then passed to the POST-enabled action method.

Model binding is a powerful and customizable feature that eliminates the grind and toil of dealing with HTTP requests directly and lets us work with C# objects rather than dealing with `Request.Form[]` and `Request.QueryString[]` values. The `GuestResponse` object that is passed as the parameter to the action method is automatically populated with the data from the form fields. I dive into the detail of model binding, including how it can be customized, in Chapter 24.

Rendering Other Views

The second overload of the `RsvpForm` action method also demonstrates how to tell MVC to render a specific view in response to a request, rather than the default view. Here is the relevant statement:

```
...
return View("Thanks", guestResponse);
...
```

This call to the `View` method tells MVC to find and render a view called `Thanks` and to pass the `GuestResponse` object to the view. To create the view I specified, right-click on any of the `HomeController` methods and select `Add View` from the pop-up menu and use the `Add View` dialog to create a strongly typed view called `Thanks` that uses the `GuestResponse` model class and that is based on the `Empty` template. (See the `Adding a Strongly Typed View` section for step-by-step details if needed). Visual Studio will create the view as `Views/Home/Thanks.cshtml`. Edit the new view so that it matches Listing 2-15—I have highlighted the markup you need to add.

Listing 2-15. The Contents of the `Thanks.cshtml` File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}
```

```

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <div>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but thanks for letting us know.
        }
    </div>
</body>
</html>

```

The Thanks view uses Razor to display content based on the value of the `GuestResponse` properties that I passed to the `View` method in the `RsvpForm` action method. The Razor `@model` expression specifies the domain model type that the view is strongly typed with. To access the value of a property in the domain object, I use `Model.PropertyName`. For example, to get the value of the `Name` property, I call `Model.Name`. Don't worry if the Razor syntax doesn't make sense—I explain it in more detail in Chapter 5.

Now that I have created the Thanks view, I have a basic working example of handling a form with MVC. Start the application in Visual Studio, click the `RSVP Now` link, add some data to the form, and click the `Submit RSVP` button. You will see the result shown in Figure 2-20 (although it will differ if your name is not Joe or you said you could not attend).

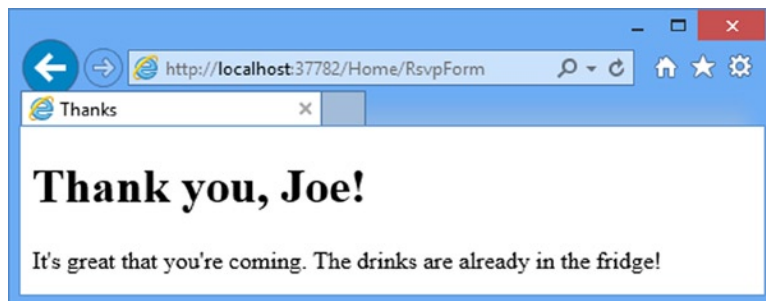


Figure 2-20. The Thanks view

Adding Validation

I am now in a position to add validation to my application. Without validation, users could enter nonsense data or even submit an empty form. In an MVC application, validation is typically applied in the domain model, rather than in the user interface. This means that I am able to define validation criteria in one place and have it take effect anywhere in the application that the model class is used. ASP.NET MVC supports *declarative validation rules* defined with attributes from the `System.ComponentModel.DataAnnotations` namespace, meaning that validation constraints are expressed using the standard C# attribute features. Listing 2-16 shows how I applied these attributes to the `GuestResponse` model class.

Listing 2-16. Applying Validation in the GuestResponse.cs File

```

using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models {

    public class GuestResponse {

        [Required(ErrorMessage = "Please enter your name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter your email address")]
        [RegularExpression(".+\\@.+\\..+",
            ErrorMessage = "Please enter a valid email address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your phone number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whether you'll attend")]
        public bool? WillAttend { get; set; }
    }
}

```

The validations rules are shown in bold. MVC automatically detects the attributes and uses them to validate data during the model-binding process. Notice that I have imported the namespace that contains the validations, so I can refer to them without needing to qualify their names.

■ **Tip** As noted earlier, I used a nullable `bool` for the `WillAttend` property. I did this so that I could apply the `Required` validation attribute. If I had used a regular `bool`, the value I received through model binding could be only `true` or `false`, and I wouldn't be able to tell if the user had selected a value. A nullable `bool` has three possible values: `true`, `false`, and `null`. The `null` value will be used if the user hasn't selected a value, and this causes the `Required` attribute to report a validation error. This is a nice example of how the MVC Framework elegantly blends C# features with HTML and HTTP.

I check to see if there has been a validation problem using the `ModelState.IsValid` property in the controller class. Listing 2-17 shows how I have done this in the POST-enabled `RsvpForm` action method in the `Home` controller class.

Listing 2-17. Checking for Form Validation Errors in the HomeController.cs File

```

...
[HttpPost]
public ActionResult RsvpForm(GuestResponse guestResponse) {
    if (ModelState.IsValid) {
        // TODO: Email response to the party organizer
        return View("Thanks", guestResponse);
    } else {
        // there is a validation error
        return View();
    }
}
...

```

If there are no validation errors, I tell MVC to render the Thanks view, just as I did previously. If there *are* validation errors, I re-render the RsvpForm view by calling the View method without any parameters.

Just displaying the form when there is an error is not helpful—I also need to provide the user with some indication of what the problem is and why I could not accept their form submission. I do this by using the `Html.ValidationSummary` helper method in the RsvpForm view, as shown in Listing 2-18.

Listing 2-18. Using the `Html.ValidationSummary` Helper Method in the RsvpForm.cshtml File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there",
                    Value = bool.TrueString},
                new SelectListItem() {Text = "No, I can't come",
                    Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

If there are no errors, the `Html.ValidationSummary` method creates a hidden list item as a placeholder in the form. MVC makes the placeholder visible and adds the error messages defined by the validation attributes. You can see how this appears in Figure 2-21.

The screenshot shows a web browser window with the address bar displaying `http://localhost:37782/Home/RsvpForm`. The browser tab is titled "RsvpForm". The page content includes a bulleted list of instructions: "Please enter your name", "Please enter a valid email address", "Please enter your phone number", and "Please specify whether you'll attend". Below the list are four form fields: "Your name:" (empty text box), "Your email:" (text box containing "hello"), "Your phone:" (empty text box), and "Will you attend?" (dropdown menu showing "Choose an option"). At the bottom is a "Submit RSVP" button.

Figure 2-21. *The validation summary*

The user won't be shown the Thanks view until all of the validation constraints applied to the `GuestResponse` class have been satisfied. Notice that the data entered into the form was preserved and displayed again when the view was rendered with the validation summary. This is another benefit of the model binding feature and it simplifies working with form data.

Note If you have worked with ASP.NET Web Forms, you will know that Web Forms has a concept of *server controls* that retain state by serializing values into a hidden form field called `__VIEWSTATE`. ASP.NET MVC model binding is not related to the Web Forms concepts of server controls, postbacks, or View State. ASP.NET MVC does not inject a hidden `__VIEWSTATE` field into your rendered HTML pages.

Highlighting Invalid Fields

The HTML helper methods that create text boxes, drop-downs, and other elements have a handy feature that can be used in conjunction with model binding. The same mechanism that preserves the data that a user entered in a form can also be used to highlight individual fields that failed the validation checks.

When a model class property has failed validation, the HTML helper methods will generate slightly different HTML. As an example, here is the HTML that a call to `Html.TextBoxFor(x => x.Name)` generates when there is no validation error:

```
<input data-val="true" data-val-required="Please enter your name" id="Name" name="Name"
      type="text" value="" />
```

And here is the HTML the same call generates when the user doesn't provide a value (which is a validation error because I applied the Required attribute to the Name property in the GuestResponse model class):

```
<input class="input-validation-error" data-val="true" data-val-required="Please enter your name"
id="Name" name="Name" type="text" value="" />
```

I have highlighted the difference in bold: the helper method added a class called `input-validation-error` to the input element. I can take advantage of this feature by creating a style sheet that contains CSS styles for this class and the others that different HTML helper methods apply.

The convention in MVC projects is that static content, such as CSS style sheets, is placed into a folder called `Content`. Create this folder by right-clicking on the `PartyInvites` item in the Solution Explorer, selecting **Add ► New Folder** from the menu and setting the name to `Content`.

To create the CSS file, right click on the newly created `Content` folder, select **Add ► New Item** from the menu and choose **Style Sheet** from the set of item templates. Set the name of the new file to `Styles.css`, as shown in Figure 2-22.

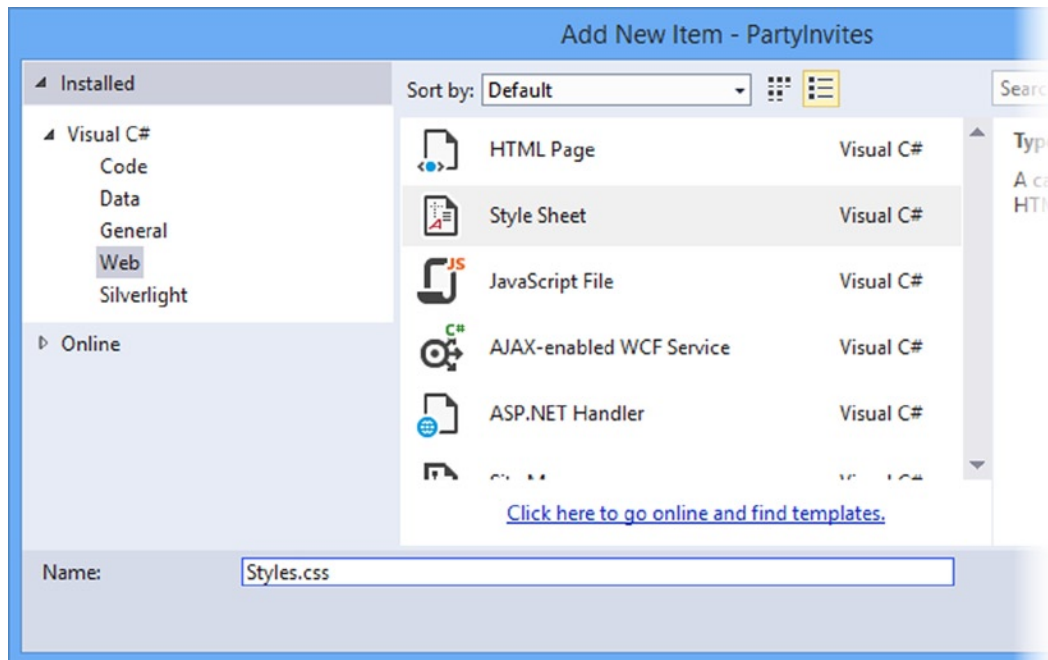


Figure 2-22. Creating a new style sheet

Click the **Add** button and Visual Studio will create the `Content/Styles.css` file. Set the content of the new file to match Listing 2-19.

Listing 2-19. The Contents of the Styles.css File

```
.field-validation-error {color: #f00;}
.field-validation-valid { display: none;}
.input-validation-error { border: 1px solid #f00; background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00;}
.validation-summary-valid { display: none;}
```

To use this style sheet, I add a new reference to the head section of `RsvpForm` view, as shown in Listing 2-20. You add link elements to views just as you would to a regular static HTML file, although in Chapter 27, I show you the *bundles* feature that allows JavaScript and CSS style sheets to be consolidated and delivered to the browsers over a single HTTP request.

■ **Tip** You can drag JavaScript and CSS files from the Solution Explorer windows and drop them on the code editor. Visual Studio will create script and link elements for the files you have selected.

Listing 2-20. Adding the Link Element in the `RsvpForm.cshtml` File

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" type="text/css" href="~/Content/Styles.css" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm()) {
        @Html.ValidationSummary()
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email) </p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone) </p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there",
                    Value = bool.TrueString},
                new SelectListItem() {Text = "No, I can't come",
                    Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

■ **Tip** If you have moved to MVC 5 directly from MVC 3, you might have been expecting us to have added the CSS file to the view by specifying the `href` attribute as `@Href("~/Content/Site.css")` or `@Url.Content("~/Content/Site.css")`. As of MVC 4, Razor detects attributes that begin with `~/` and automatically inserts the `@Href` or `@Url` call for you.

With the application of the style sheet, a more visually obvious validation error will be displayed when data is submitted that causes a validation error, as shown in Figure 2-23.

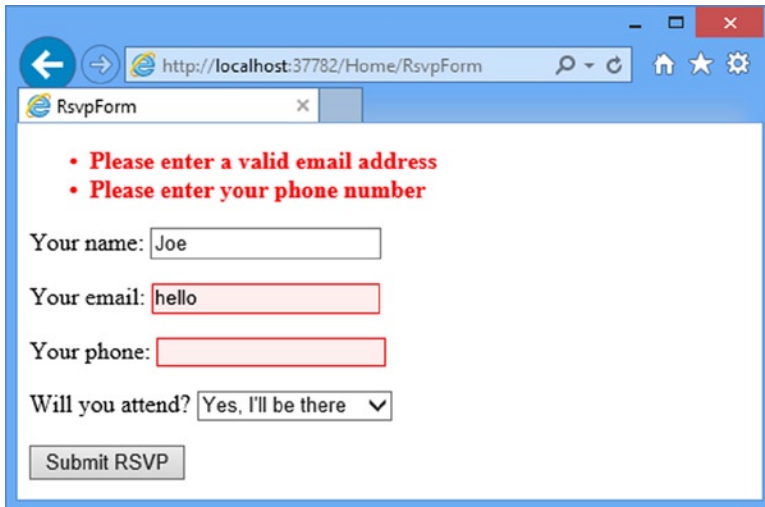


Figure 2-23. Automatically highlighted validation errors

Styling the Content

The basic functionality of the application is in place – except for sending emails, which I’ll get to shortly—but the overall appearance is pretty poor. Although this is a book focused on server-side development, Microsoft has adopted a number of open source libraries and included them in some of the Visual Studio project templates.

I am not a fan of these templates, but I do like some of the libraries they use and one of the new adoptees in MVC 5 is Bootstrap, which is a nice CSS library originally developed by Twitter that has become widely used.

You don’t have to use the Visual Studio project templates to use libraries like Bootstrap, of course. You can download files directly from project web sites or use NuGet, which is integrated into Visual Studio and provides access to a catalogue of pre-packaged software that can be downloaded and installed automatically. One of the best NuGet features is that it manages dependencies between packages such that if you install Bootstrap, for example, NuGet will also download and install jQuery which some Bootstrap features depend on.

Using NuGet to Install Bootstrap

To install the Bootstrap package, select **Library Package Manager ► Package Manager Console** from the Visual Studio **Tools** menu. Visual Studio will open the NuGet command line. Enter the following command and hit return:

```
Install-Package -version 3.0.0 bootstrap
```

The `Install-Package` command tells NuGet to download a package and its dependencies and add them to the project. The name of the package I want is called `bootstrap`, and you can search for package names either from the NuGet web site (<http://www.nuget.org>) or using the Visual Studio NuGet user interface (select **Tools ► Library Package Manager ► Manage NuGet Packages for Solution**).

I have used `-version` to specify that I want Bootstrap version 3, which is the latest stable version available as I write this. Without `-version`, NuGet would have downloaded the latest version of the package, but I want to make sure that you are able to recreate the examples exactly as I have shown them and so installing a specific version helps me to ensure consistency.

NuGet will download all of the files required for Bootstrap and for jQuery, which Bootstrap relies on. CSS files are added to the Content folder and a Scripts folder is created (which is the standard MVC location for JavaScript files) and populated with Bootstrap and jQuery files. (A fonts folder is also created—this is a quirk of the Bootstrap typography features, which expect files to be in certain locations).

Note The reason that I am showing you Bootstrap in this chapter is to illustrate how readily the HTML generated by the MVC Framework can be used with popular CSS and JavaScript libraries. I don't want to lose my focus on server-side development, however, and so if you want complete details of the *client*-side aspects of working with the MVC Framework, then see my book *Pro ASP.NET MVC 5 Client*, which will be published by Apress in 2014.

Styling the Index View

The basic Bootstrap features work by applying classes to elements that correspond to CSS selectors defined in the files added to the Content folder. You can get full details of the classes that Bootstrap defines from <http://getbootstrap.com>, but you can see how I have applied some basic styling to the `Index.cshtml` view file in Listing 2-21.

Listing 2-21. Adding Bootstrap to the `Index.cshtml` File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>Index</title>
    <style>
        .btn a { color: white; text-decoration: none; }
        body { background-color: #F1F1F1; }
    </style>
</head>
<body>
    <div class="text-center">
        <h2>We're going to have an exciting party!</h2>
        <h3>And you are invited</h3>
        <div class="btn btn-success">
            @Html.ActionLink("RSVP Now", "RsvpForm")
        </div>
    </div>
</body>
</html>
```

I have added link elements for the `bootstrap.css` and `bootstrap-theme.css` files in the Content folder. These are the Bootstrap files required for the basic CSS styling that the library provides and there is a corresponding JavaScript file in the Scripts folder, but I won't need it in this chapter. I have also defined a `style` element that sets the background color for the body element and styles the text for a elements.

■ **Tip** You will notice that each of the Bootstrap files in the Content folder has a twin with the prefix `min`—e.g., `bootstrap.css` and `bootstrap.min.cs`. It is common practice to *minify* JavaScript and CSS files when deploying an application into production, which is a process of removing all of the whitespace and, in the case of JavaScript, replacing the function and variable names with shorter labels. The goal of minification is to reduce the amount of bandwidth required to deliver your content to the browser and in Chapter 27, I describe the ASP.NET features for managing this process automatically. For this chapter—and most of the other chapters in this book—I will use the regular files, which is normal practice during development and testing.

Having imported the Bootstrap styles and defined a couple of my own, I need to style my elements. This is a simple example and so I only need to use three Bootstrap CSS classes: `text-center`, `btn` and `btn-success`.

The `text-center` class centers the content of an element and its children. The `btn` class styles a button, input or a element as a pretty button and the `btn-success` specifies which of a range of colors I want the button to be. The color of the button depends on the theme that is being used—I have the default theme (as defined by the `bootstrap-theme.css` file), but there are endless replacements available with a search online. You can see the effect that I have created in Figure 2-24.

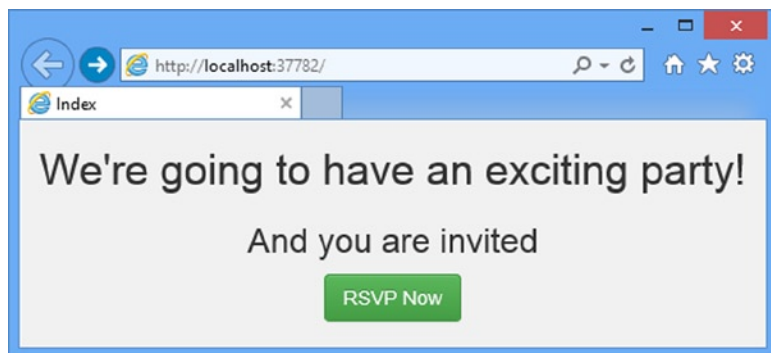


Figure 2-24. Styling the Index view

It will be obvious to you that I am not a web designer. In fact, as a child, I was excused from art lessons on the basis that I had absolutely no talent whatsoever. This had the happy result of making more time for math lessons but meant that my artistic skills have not developed beyond those of the average 10 year old. For a real project, I would seek a professional to help design and style the content, but for this example I am going it alone and that means applying Bootstrap with as much restraint and consistency as I can muster.

Styling the RsvpForm View

Bootstrap defines classes that can be used to style forms. I am not going to go into detail, but you can see how I have applied these classes in Listing 2-22.

Listing 2-22. Adding Bootstrap to the RsvpForm.cshtml File

```

@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Styles.css" rel="stylesheet" />
    <title>RsvpForm</title>
</head>
<body>
    <div class="panel panel-success">
        <div class="panel-heading text-center"><h4>RSVP</h4></div>
        <div class="panel-body">
            @using (Html.BeginForm()) {
                @Html.ValidationSummary()
                <div class="form-group">
                    <label>Your name:</label>
                    @Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
                </div>
                <div class="form-group">
                    <label>Your email:</label>
                    @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
                </div>
                <div class="form-group">
                    <label>Your phone:</label>
                    @Html.TextBoxFor(x => x.Phone, new { @class = "form-control" })
                </div>
                <div class="form-group">
                    <label>Will you attend?</label>
                    @Html.DropDownListFor(x => x.WillAttend, new[] {
                        new SelectListItem() {Text = "Yes, I'll be there",
                            Value = bool.TrueString},
                        new SelectListItem() {Text = "No, I can't come",
                            Value = bool.FalseString}
                    }, "Choose an option", new { @class = "form-control" })
                </div>
                <div class="text-center">
                    <input class="btn btn-success" type="submit" value="Submit RSVP" />
                </div>
            }
        </div>
    </div>
</body>
</html>

```

The Bootstrap classes in this example create a panel with a header, just to give structure to the layout. To style the form, I have used the `form-group` class, which is used to style the element that contains the label and the associated input or select element.

These elements are created using HTML helper methods, which means that there are not statically defined elements available to which I can apply the required `form-control` class. Fortunately, the helper methods take an optional object argument that lets me specify attributes on the elements that they create, as follows:

```
...
@Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
...
```

I created the object using the C# *anonymous type* feature, which I describe in Chapter 4 and specified that the `class` attribute should be set to `form-control` on the element that the `TextBoxFor` helper generates. The properties defined by the object are used for the name of the attribute added to the HTML element and `class` is a reserved word in the C# language, so I have to prefix it with `@`. This is a standard C# feature that allows keywords to be used in expressions. You can see the result of my styles in Figure 2-25.

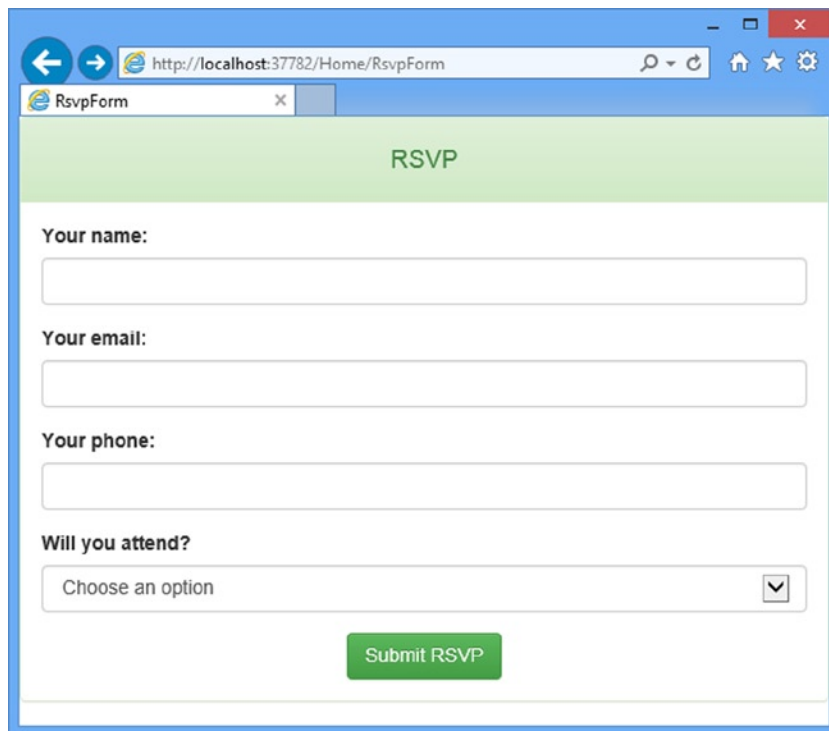


Figure 2-25. Styling the `RsvpForm` view

Styling the Thanks View

The last view file to style is `Thanks.cshtml` and you can see how I have done this in Listing 2-23. You will notice that the markup I have added is similar to that in the `Index.cshtml` view. To make an application easier to manage, it is a good principal to avoid duplicating code and markup wherever possible and in Chapter 5 I will introduce you to *Razor layouts* and in Chapter 20, I describe *partial views*, both of which can be used to reduce duplication of markup.

Listing 2-23. Applying Bootstrap to the Thanks.cshtml File

```

@model PartyInvites.Models.GuestResponse

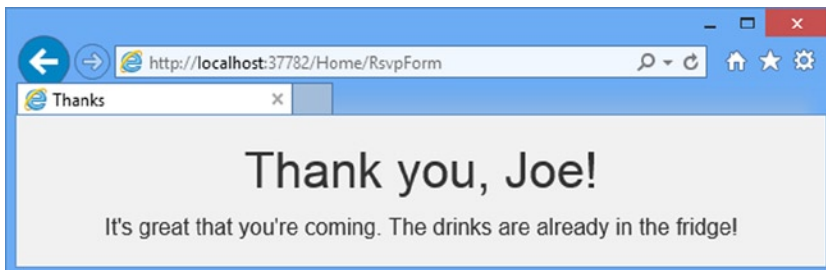
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
    <style>
        body { background-color: #F1F1F1; }
    </style>
</head>
<body>
    <div class="text-center">
        <h1>Thank you, @Model.Name!</h1>
        <div class="lead">
            @if (Model.WillAttend == true) {
                @:It's great that you're coming. The drinks are already in the fridge!
            } else {
                @:Sorry to hear that you can't make it, but thanks for letting us know.
            }
        </div>
    </div>
</body>
</html>

```

The lead class applies one of the Bootstrap typographic styles and you can see the effect in Figure 2-26.

**Figure 2-26.** Styling the Thanks View

Completing the Example

The last requirement for my example application is to e-mail completed RSVPs to the party organizer. I could do this by adding an action method to create and send an e-mail message using the e-mail classes in the .NET Framework—and that would be the technique which is most consistent with the MVC pattern. Instead, I am going to use the `WebMail` helper method. This is not part of the MVC framework, but it does let me complete this example without getting mired in the details of setting up other means of sending e-mail. I want the e-mail message to be sent as I render the Thanks view. Listing 2-24 show the changes that I need to apply.

Listing 2-24. Using the WebMail Helper in the Thanks.cshtml File

```
...
<body>
    @{
        try {
            WebMail.SmtpServer = "smtp.example.com";
            WebMail.SmtpPort = 587;
            WebMail.EnableSsl = true;
            WebMail.UserName = "mySmtpUsername";
            WebMail.Password = "mySmtpPassword";
            WebMail.From = "rsvps@example.com";

            WebMail.Send("party-host@example.com", "RSVP Notification",
                Model.Name + " is " + ((Model.WillAttend ?? false) ? "" : "not")
                    + "attending");

        } catch (Exception) {
            @:<b>Sorry - we couldn't send the email to confirm your RSVP.</b>
        }
    }
    <div class="text-center">
        <h1>Thank you, @Model.Name!</h1>
        <div class="lead">
            @if (Model.WillAttend == true) {
                @:It's great that you're coming. The drinks are already in the fridge!
            } else {
                @:Sorry to hear that you can't make it, but thanks for letting us know.
            }
        </div>
    </div>
</body>
...
```

■ **Note** I used the `WebMail` helper because it lets us demonstrate sending an e-mail message with a minimum of effort. Typically, however, I would prefer to put this functionality in an action method. I will explain why when I describe the MVC architecture pattern in Chapter 3.

I have added a Razor expression that uses the `WebMail` helper to configure the details of my e-mail server, including the server name, whether the server requires SSL connections, and account details. Once I have configured all of the details, I use the `WebMail.Send` method to send the e-mail.

I enclosed all of the e-mail code in a `try...catch` block so that I can alert the user if the e-mail is not sent. I do this by adding a block of text to the output of the `Thanks` view. A better approach would be to display a separate error view when the e-mail message cannot be sent, but I wanted to keep things simple for this first MVC application.

Summary

In this chapter, I created a new MVC project and used it to construct a simple MVC data-entry application, giving you a first glimpse of the MVC Framework architecture and approach. I skipped over some key features (including Razor syntax, routing, and automated testing), but I come back to these topics in depth in later chapters. In the next chapter, I describe the MVC architecture, design patterns, and techniques that I use throughout the rest of this book and which form the foundation for effective development with the MVC Framework.