CHAPTER 2

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 ASP.NET Core MVC. I take things one step at a time so you can see how an MVC application is constructed. To keep things simple, I 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.

UPDATES TO THIS BOOK

Microsoft has an active development schedule for .NET Core and ASP.NET Core MVC, which means that there may be new releases available by the time you read this book. It doesn't seem fair to expect readers to buy a new book every few months, especially since most changes are relatively minor. Instead, I will post free updates to the GitHub repository for this book (`https://github.com/apress/pro-asp.net-core-mvc-2`) for breaking changes caused by minor releases.

This kind of update is an experiment for me (and for Apress), and I don't yet know what form those updates may take—not least because I don't know what the future major releases of ASP.NET Core MVC will contain—but the goal is to extend the life of this book by supplementing the examples it contains.

I am not making any promises about what the updates will be like, what form they will take, or how long I will produce them before folding them into a new edition of this book. Please keep an open mind and check the repository for this book when new ASP.NET Core MVC versions are released. If you have ideas about how the updates could be improved, then e-mail me at `adam@adam-freeman.com` and let me know.

Installing Visual Studio

This book relies on Visual Studio 2017, which provides the development environment for ASP.NET Core MVC projects. I use the free *Visual Studio 2017 Community* edition, which can be downloaded from `www.visualstudio.com`. When installing Visual Studio 2017, you must select the .NET Core cross-platform development workload, as shown in Figure 2-1.

© Adam Freeman 2017

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_2

■ **Note** Visual Studio 2017 predates the release of ASP.NET Core MVC 2. You must apply the latest updates if you have installed Visual Studio for earlier versions of ASP.NET Core MVC. You can apply updates by running the Visual Studio installer and selecting Update for the Visual Studio edition you are using.

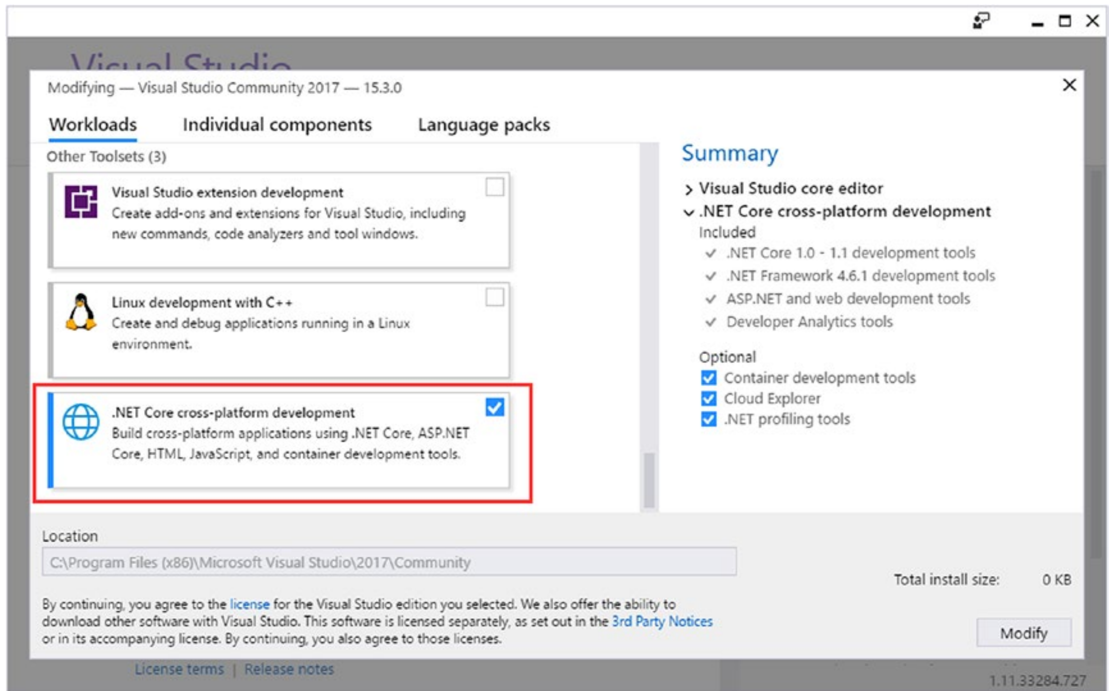

Figure 2-1. *Selecting the Visual Studio workload*

■ **Tip** Visual Studio only supports Windows. You can create ASP.NET Core MVC applications on other platforms using Visual Studio Code. Visual Studio Code doesn't provide all of the features of Visual Studio, but it is an excellent editor and does everything required for MVC application development. See Chapter 13 for details.

Installing the .NET Core 2.0 SDK

The Visual Studio installation includes all of the features required for ASP.NET Core MVC development, but it doesn't include .NET Core 2.0, which must be downloaded and installed separately.

Go to `https://www.microsoft.com/net/core` and download and run the .NET Core SDK installer for Windows. Once the installer has finished, open a new command prompt or PowerShell window and run the following command to display the version of .NET that has been installed:

```
dotnet --version
```

If the installation has been successful, the result of this command will be `2.0.0`.

Creating a New ASP.NET Core MVC Project

I am going to start by creating a new ASP.NET Core MVC project in Visual Studio. Select New ➤ Project from the File menu to open the New Project dialog. If you navigate to the Installed ➤ Visual C# ➤ Web section in the left panel, you will see the ASP.NET Core Web Application (.NET Core) project template. Select this project type, as shown in Figure 2-2.

■ **Tip** The choice of project template can be confusing because their names are so similar. The ASP.NET Web Application (.NET Framework) template is for creating projects using the legacy versions of ASP.NET and the MVC Framework, which predated ASP.NET Core. The other two templates are for creating ASP.NET Core applications, and they differ in the runtime they use, allowing you to select either the .NET Framework or .NET Core. I explain the difference between them in Chapter 6, but I use the .NET Core option throughout this book, so it is the one you should select to ensure that you get the same results from the example applications.

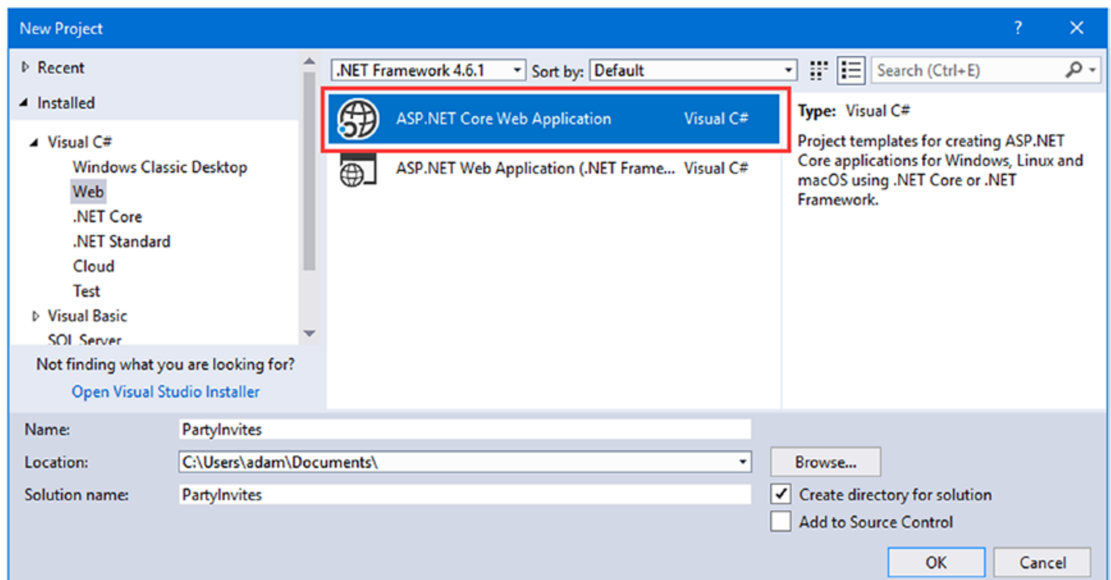

***Figure 2-2.** The ASP.NET Core Web Application project template*

Enter **PartyInvites** in the Name field for the new project. Click the OK button to continue and you will see another dialog box, shown in Figure 2-3, which asks you to set the initial content for the project. Ensure that .NET Core and ASP.NET Core 2.0 are selected from the drop-down menus, as shown in the figure.

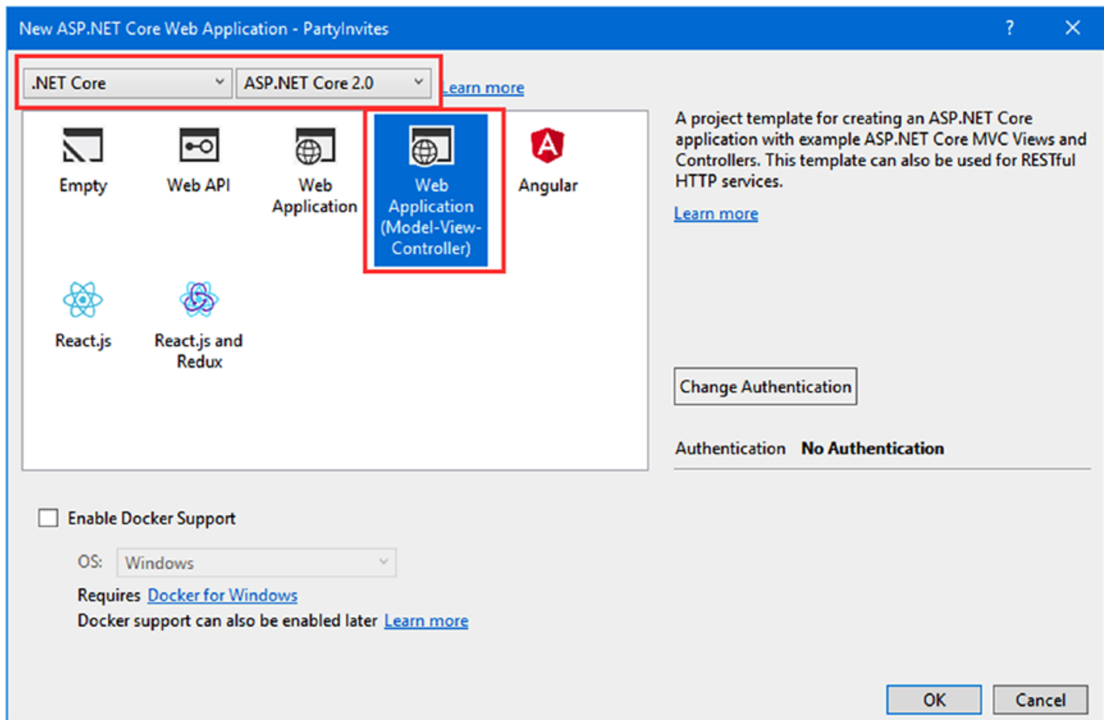

Figure 2-3. *Selecting the initial project configuration*

There are several template options, each of which creates a project with different starting content. For this chapter, select the Web Application (Model-View-Controller) option, which sets up an MVC application with predefined content to jump-start development.

■ **Note** This is the only chapter in which I use the Web Application (Model-View-Controller) project template. I don't like using predefined project 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 applications, so I use the Empty template throughout the rest of the book. This chapter is about getting started quickly, for which the Web Application (Model-View-Controller) template is well-suited.

Click the Change Authentication button and ensure that the No Authentication option is selected, as shown in Figure 2-4. This project doesn't require any authentication, but I explain how to secure ASP.NET applications in Chapters 28, 29, and 30.

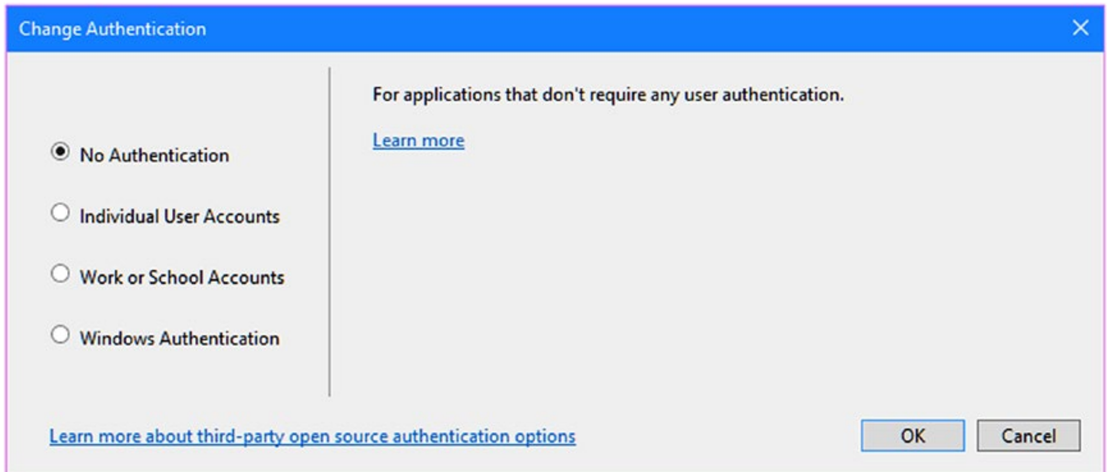

Figure 2-4. *Selecting the authentication settings*

Click OK to close the Change Authentication dialog. Ensure that the Enable Docker Support option is unchecked and then click OK to create the PartyInvites project.

Once Visual Studio has created the project, you will see a number of files and folders displayed in the Solution Explorer window, as shown in Figure 2-5. This is the default project structure for a new MVC project created using the Web Application (Model-View-Controller) template, and you will soon understand the purpose of each file and folder that Visual Studio creates.

■ **Tip** If you see a `Pages` folder, rather than `Controllers`, `Models`, and `Views` folders, then you have selected the Web Application template and not the (confusingly similar) Web Application (Model-View-Controller) template. I have no idea why Microsoft thought that such similar names were a good idea, but you will have to delete the project you created and start over.

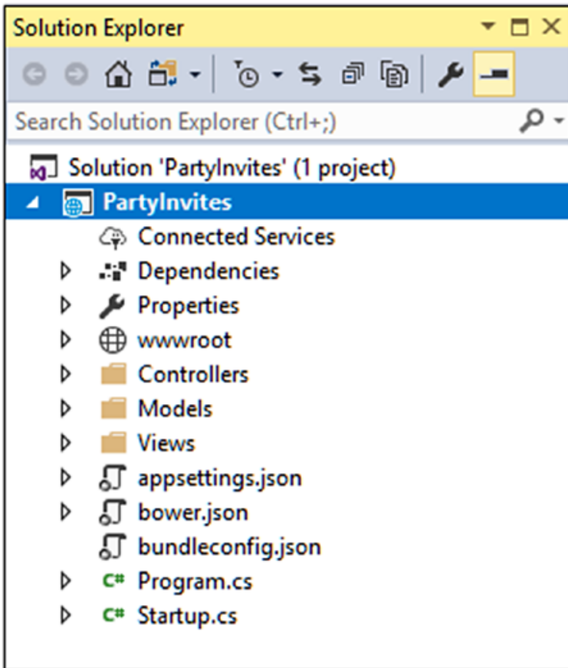

Figure 2-5. *The initial file and folder structure of an ASP.NET Core MVC project*

You can run the application by selecting Start Debugging from the Debug menu (if it prompts you to enable debugging, just click the OK button). When you do this, Visual Studio compiles the application, uses an application server called IIS Express to run it, and opens a web browser to request the application content. It can take Visual Studio some time to run the project for the first time, and when the process is complete, you will see the results shown in Figure 2-6.

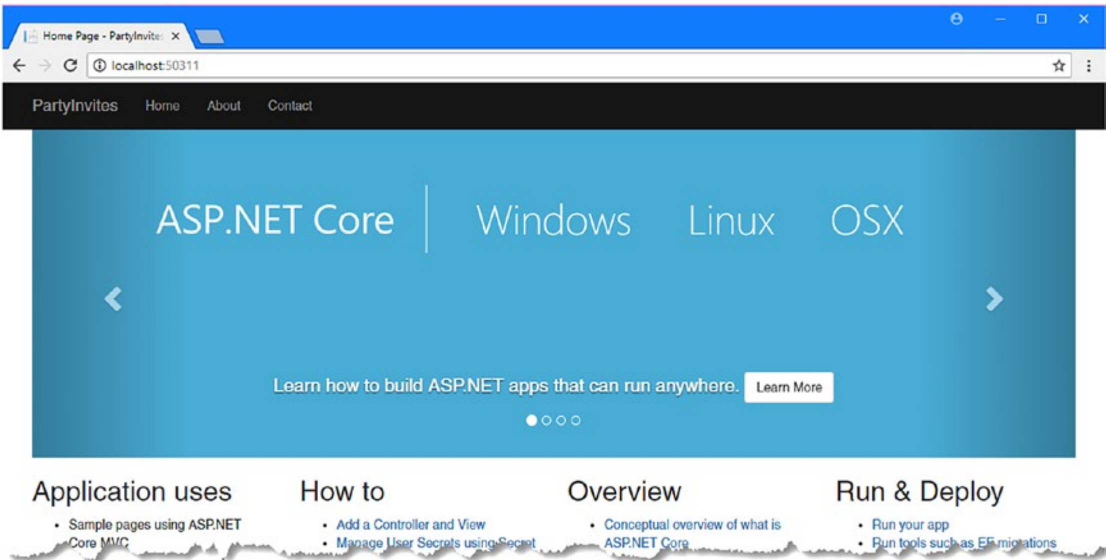

Figure 2-6. *Running the example project*

When Visual Studio creates a project with the Web Application (Model-View-Controller) template, it adds some basic code and content, which is what you see when you run the application. Throughout the rest of the chapter, I will replace this content to create a simple MVC application.

When you are finished, be sure to stop debugging by closing the browser window 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. You can select any browser that you have installed by clicking the arrow to the right of the IIS Express toolbar button and choosing from the list of options in the Web Browser menu, as shown in Figure 2-7.

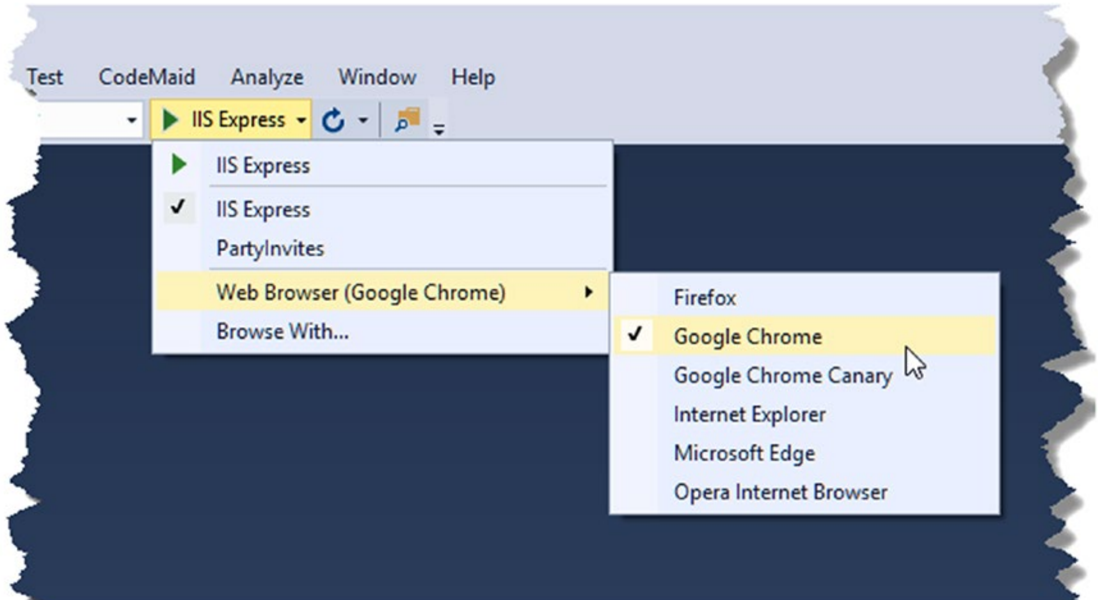

Figure 2-7. *Selecting a browser*

From here on, I will use Google Chrome or Google Chrome Canary for all the screenshots in this book, but you can use any modern browser to display the examples in the books, including Microsoft Edge.

Adding the Controller

In the MVC pattern, incoming requests are handled by *controllers*. In ASP.NET Core MVC, controllers are just C# classes (usually inheriting from the `Microsoft.AspNetCore.Mvc.Controller` class, which is the built-in MVC 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.

Visual Studio adds a default controller class to the project, which you can see if you expand the `Controllers` folder in the Solution Explorer. The file is called `HomeController.cs`. Controller classes contain a name followed by the word `Controller`, which means that when you see a file called `HomeController.cs`, you know that it contains a controller called `Home`, which is the default controller that is used in MVC applications. Click the `HomeController.cs` file in the Solution Explorer so that Visual Studio opens it for editing. You will see the C# code shown in Listing 2-1.

Listing 2-1. The Initial Contents of the HomeController.cs File in the Controllers Folder

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            return View();
        }

        public IActionResult About() {
            ViewData["Message"] = "Your application description page.";

            return View();
        }

        public IActionResult Contact() {
            ViewData["Message"] = "Your contact page.";

            return View();
        }

        public IActionResult Error() {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id
                ?? HttpContext.TraceIdentifier });
        }
    }
}
```

Replace the code in the `HomeController.cs` file so that it matches Listing 2-2. I have removed all but one of the methods, changed the result type and its implementation, and removed the `using` statements for unused namespaces.

Listing 2-2. Changing the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

These changes don't produce a dramatic effect, but they make for a nice demonstration. I have changed the 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.

■ **Tip** If you left the application running from the previous section, then select Restart from the Debugging menu or, if you prefer, select Stop Debugging and then Start Debugging.

The browser will make an HTTP request to the server. The default MVC configuration means that the request will be handled using the `Index` method (known as an *action method* or just an *action*) and the result from the method will be sent back to the browser, as shown in Figure 2-8.

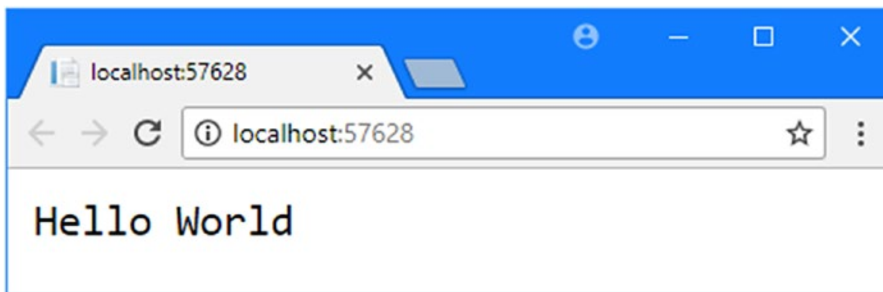

Figure 2-8. *The output from the action method*

■ **Tip** Notice that Visual Studio has directed the browser to port 57628. 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 taskbar notification area, you will find an icon for IIS Express. This is a cut-down version of the full IIS application server that is included with Visual Studio and is used to deliver ASP.NET Core content and services during development. I'll show you how to deploy an MVC project into a production environment in Chapter 12.

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. A route is a rule that is used to decide how a request is handled. When Visual Studio creates the MVC project, it adds some default routes to get you 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:57628/`, except that the port part may be different. If you append `/Home` or `/Home/Index` to the URL, you will see the same `Hello World` result from the MVC application.

This is a good example of benefiting from following conventions implemented by ASP.NET Core MVC. In this case, the convention is that I will have a controller called `HomeController` and it will be the starting point for the MVC application. The default configuration that Visual Studio creates for a new project assumes I will follow this convention. 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 configuration to point to whatever controller I had created instead. For this simple example, the default configuration is all I need.

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*, which tells MVC how to generate a response to a request from a browser.

Creating and Rendering a View

The first thing I need to do is modify my `Index` action method, as shown in Listing 2-3. The changes are shown in bold, which is a convention I follow throughout this book to make the examples easier to follow.

Listing 2-3. Rendering a View in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

When I return a `ViewResult` object from an action method, I am instructing MVC to *render* a view. I create the `ViewResult` object by calling the `View` method, specifying the name of the view that I want to use, which is `MyView`. If you run the application, you can see MVC trying to find the view, as shown in the error message displayed in Figure 2-9.

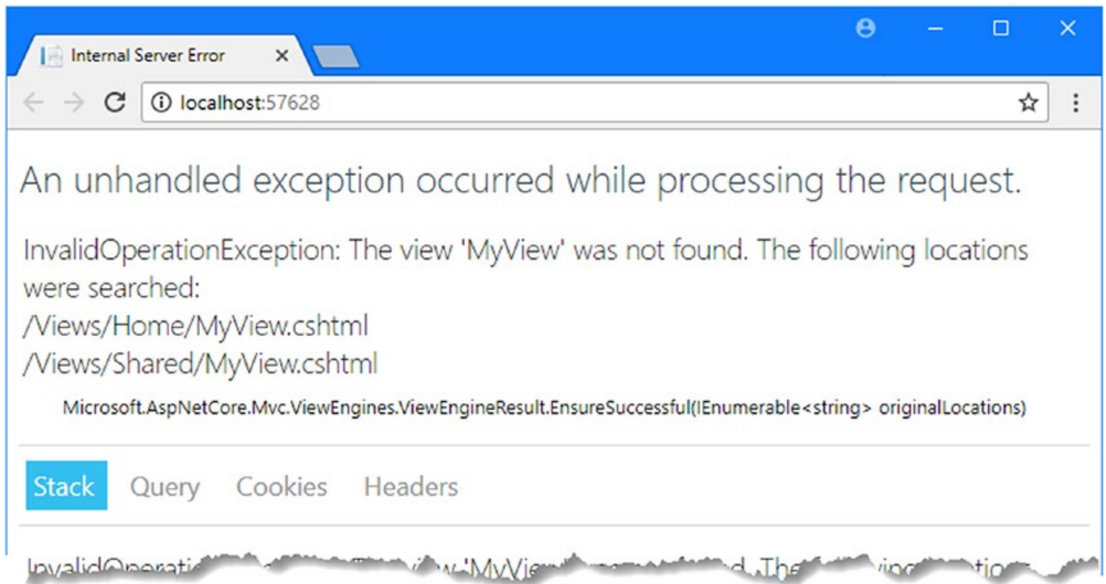

Figure 2-9. *MVC trying to find a view*

This is a helpful error message. It explains that MVC could not find the view I specified for the action method and also shows where it looked. Views are stored in the `Views` folder, organized into subfolders. Views that are associated with the `Home` controller, for example, are stored in a folder called `Views/Home`. Views that are not specific to a single controller are stored in a folder called `Views/Shared`. Visual Studio creates the `Home` and `Shared` folders automatically when the Web Application (Model-View-Controller) template is used and puts in some placeholder views to get the project started.

To create the view needed for this example, expand the `Views` folder in the Solution Explorer, right-click the `Home` folder, and select Add ➤ New Item from the pop-up menu. Visual Studio will present you with a list of item templates. Drill down to the ASP.NET Core ➤ Web ➤ ASP.NET category using the left pane and then select the MVC View Page item in the central pane, as shown in Figure 2-10. (Don't use the Razor Page template, which is not related to the MVC Framework.)

■ **Tip** You will see some existing files in the `Views` folder, which were added to the project by Visual Studio to provide some initial content, some of which you saw in Figure 2-6. You can ignore these files.

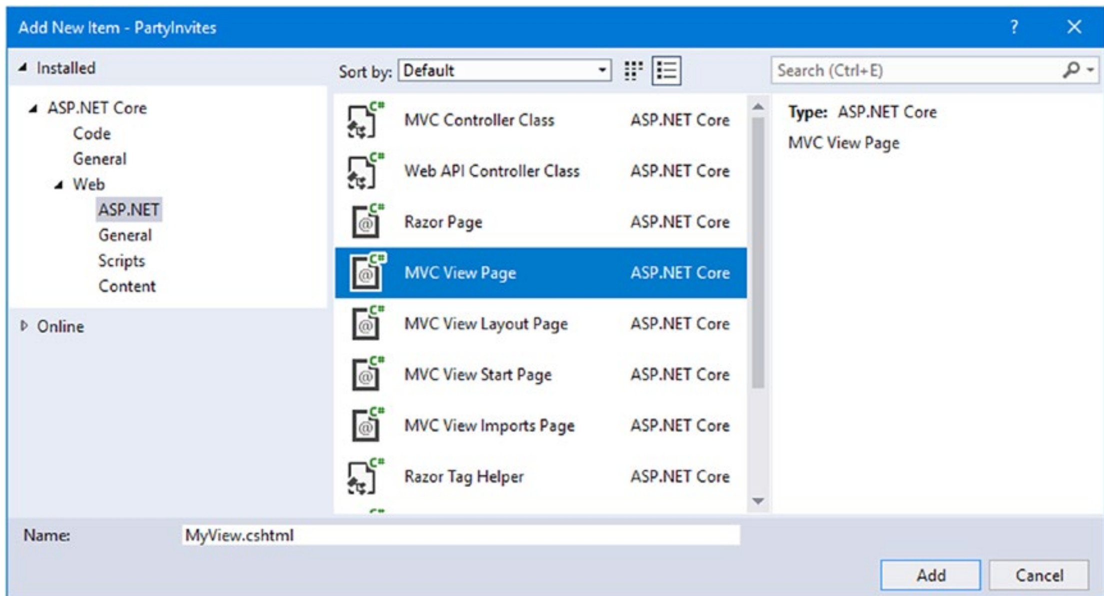

Figure 2-10. *Creating a view*

Set the Name field to `MyView.cshtml` and click the Add button to create the view. Visual Studio will create the `Views/Home/MyView.cshtml` file and open it for editing. The initial content of the view file is just some comments and a placeholder. Replace them with the content shown in Listing 2-4.

■ **Tip** It is easy to end up creating the view file in the wrong folder. If you didn't end up with a file called `MyView.cshtml` in the `Views/Home` folder, then delete the file you did create and try again.

Listing 2-4. Replacing the Content of the MyView.cshtml File in the Views/Home Folder

```
@{
    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 new contents of the view file are 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. To see the effect of creating the view, select Start Debugging from the Debug menu to run the application. You should see the result in Figure 2-11.

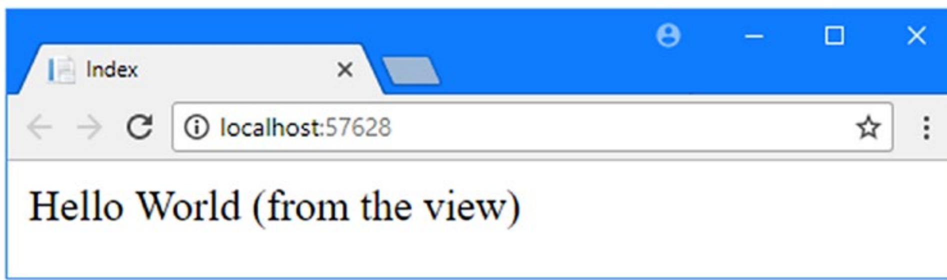

Figure 2-11. *Testing the view*

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`, MVC renders a view and returns the HTML it produces. I told MVC which view should be used, so it used the naming convention to find it 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/MyView.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 can prompt the user to log in. These objects are collectively known as *action results*. The action result system lets you encapsulate and reuse common responses in actions. I'll tell you more about them and explain the different ways they can be used 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 2-5 demonstrates passing some simple dynamic data in this way in the `HomeController.cs` file.

Listing 2-5. Setting View Data in the HomeController.cs File in the Controllers Folder

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

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 2-6, which shows the corresponding change to the `MyView.cshtml` file.

Listing 2-6. Retrieving a ViewBag Data Value in the MyView.cshtml File in the Views/Home Folder

```
@{
    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 the listing is a Razor expression that is evaluated when MVC uses the view to generate a response. When I call the `View` method in the controller's `Index` method, MVC locates the `MyView.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 2-12.

Figure 2-12. *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 these 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
- A summary page that shows who is coming to the party

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. To get started, Listing 2-7 shows the additions I made to the `Views/Home/MyView.cshtml` file.

Listing 2-7. Displaying Details of the Party in the MyView.cshtml File in the Views/Home Folder

```
@{
    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, by selecting Start Debugging from the Debug menu, you'll see the details of the party (well, the placeholder for the details, but you get the idea), as shown in Figure 2-13.

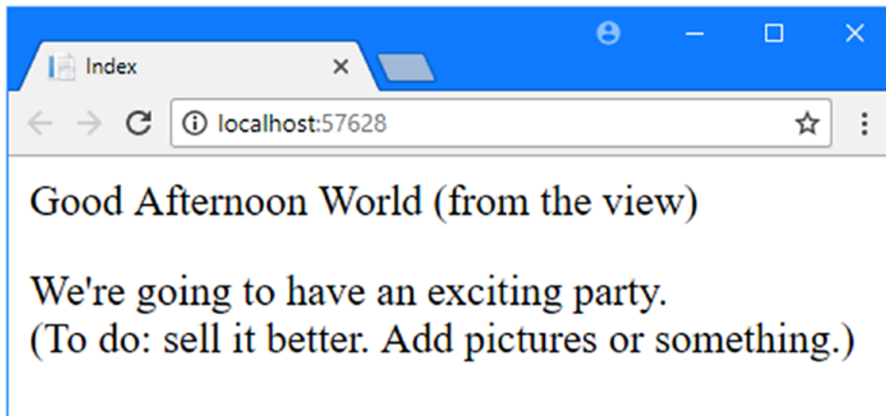

Figure 2-13. *Adding to the view HTML*

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` project because it is such a simple application and I need just one domain class that I will call `GuestResponse`. This object will be responsible for storing, validating, and confirming a RSVP.

The MVC convention is that the classes that make up a model are placed inside a folder called `Models`, which Visual Studio creates automatically when you use the Web Application (Model-View-Controller) template.

To create the class file, right-click the `Models` folder in the Solution Explorer and select Add ➤ Class from the pop-up menu. Set the name of the new class to `GuestResponse.cs` and click the Add button. Edit the contents of the new class file to match Listing 2-8.

Listing 2-8. The Contents of the GuestResponse.cs File in the Models Folder

```
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.

Creating a Second Action and a Strongly Typed View

One of my application goals is to include an RSVP form, which means I need to define an action method that can receive requests for that form. A single controller class can define multiple action methods, and the convention is to group related actions together in the same controller. Listing 2-9 shows the addition of a new action method to the `Home` controller.

Listing 2-9. Adding an Action Method in the HomeController.cs File in the Controllers Folder

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

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

The `RsvpForm` action method calls the `View` method without an argument, which tells MVC to render the default view associated with the action method, which is a view with the same name as the action method, in this case, `RsvpForm.cshtml`.

Right-click the Views/Home folder and select Add ➤ New Item from the pop-up menu. Select the MVC View Page template, set the name of the new file to `RsvpForm.cshtml`, and click the Add button to create the file. Change the content of the file so that it matches Listing 2-10.

Listing 2-10. Setting the Content of the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
        This is the RsvpForm.cshtml View
    </div>
</body>
</html>
```

This content is mostly HTML but with the addition of a `@model` Razor expression, which is used to create a *strongly typed view*. A strongly typed view is intended to render a specific model type, and if I specify the type I want to work with (the `GuestResponse` class in the `PartyInvites.Models` namespace in this case), MVC can create some helpful shortcuts to make it easier. I will take advantage of the strongly typed feature shortly.

To test the new action method and its view, start the application by selecting Start Debugging from the Debug menu and use the browser to navigate to the `/Home/RsvpForm` URL.

MVC will use the naming convention I described earlier to direct the request to the `RsvpForm` action method defined by the `Home` controller. This action method tells MVC to render the default view, which, with another application of the naming convention, renders `RsvpForm.cshml` from the `Views/Home` folder. Figure 2-14 shows the result.

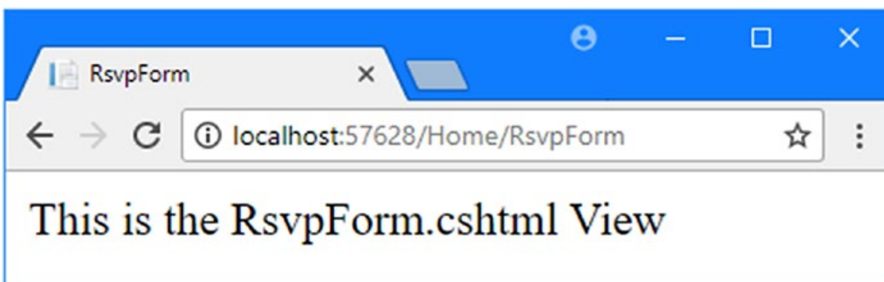

Figure 2-14. *Rendering the second view*

Linking Action Methods

I want to be able to create a link from the `MyView` view so that guests can see the `RsvpForm` view without having to know the URL that targets a specific action method, as shown in Listing 2-11.

Listing 2-11. Adding a Link to the RSVP Form in the MyView.cshtml File in the Views/Home Folder

```
@{
    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>
        <a asp-action="RsvpForm">RSVP Now</a>
    </div>
</body>
</html>
```

The addition to the listing is an a element that has an `asp-action` attribute. The attribute is an example of a *tag helper* attribute, which is an instruction for Razor that will be performed when the view is rendered. The `asp-action` attribute is an instruction to add an `href` attribute to the a element that contains a URL for an action method. I explain how tag helpers work in Chapters 24, 25, and 26, but this is the simplest type of tag helper attribute for a elements, and it tells Razor to insert a URL for an action method defined by the same controller for which the current view is being rendered. You can see the link that the helper creates by starting the project, as shown in Figure 2-15.

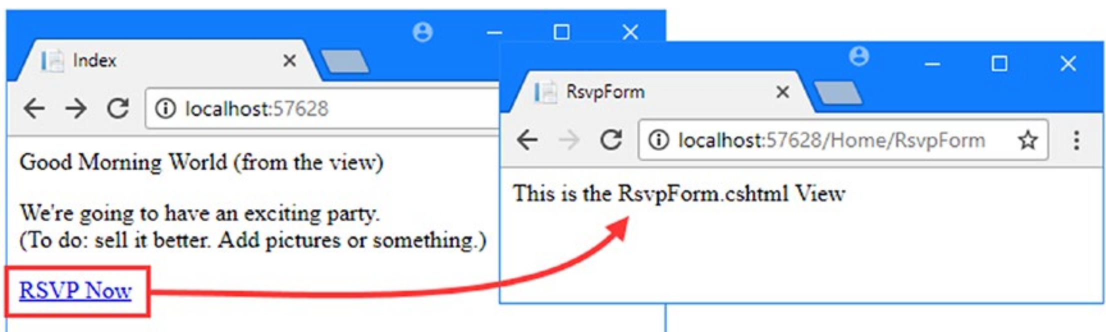

Figure 2-15. *Linking between action methods*

Start the application and roll the mouse over the RSVP Now link the browser. You will see that the link points to the following URL (allowing for the different port number that Visual Studio will have assigned to your project):

```
http://localhost:57628/Home/RsvpForm
```

There is an important principle at work here, which is that you should use the features provided by MVC to generate URLs, rather than hard-code them into your views. When the tag helper created the `href` attribute for the a element, it inspected the configuration of the application to figure out what the URL should be. This allows the configuration of the application to be changed to support different URL formats without needing to update any views. I explain how this works in Chapter 15.

Building the Form

Now that I have created the strongly typed view and can reach it from the `Index` view, I am going to build out the contents of the `RsvpForm.cshtml` file to make it into an HTML form for editing `GuestResponse` objects, as shown in Listing 2-12.

Listing 2-12. Creating a Form View in the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" /></p>
```

```
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

I have defined a label and input element for each property of the GuestResponse model class (or, in the case of the WillAttend property, a select element). Each element is associated with the model property using the asp-for attribute, which is another tag helper attribute. The tag helper attributes configure the elements to tie them to the model object. Here is an example of the HTML that the tag helpers produce and that is sent to the browser:

```
<p>
    <label for="Name">Your name:</label>
    <input type="text" id="Name" name="Name" value="">
</p>
```

The asp-for attribute on the label element sets the value of the for attribute. The asp-for attribute on the input element sets the id and name elements. This doesn't look especially useful at the moment, but you will see that associating elements with a model property offers additional advantages as the application functionality is defined.

Of more immediate use is the asp-action attribute applied to the form element, which uses the application's URL routing configuration to set the action attribute to a URL that will target a specific action method, like this:

```
<form method="post" action="/Home/RsvpForm">
```

As with the helper attribute I applied to the a element, the benefit of this approach is that you can change the system of URLs that the application uses and the content generated by the tag helpers will reflect the changes automatically.

You can see the form by running the application and clicking the RSVP Now link, as shown in Figure 2-16.

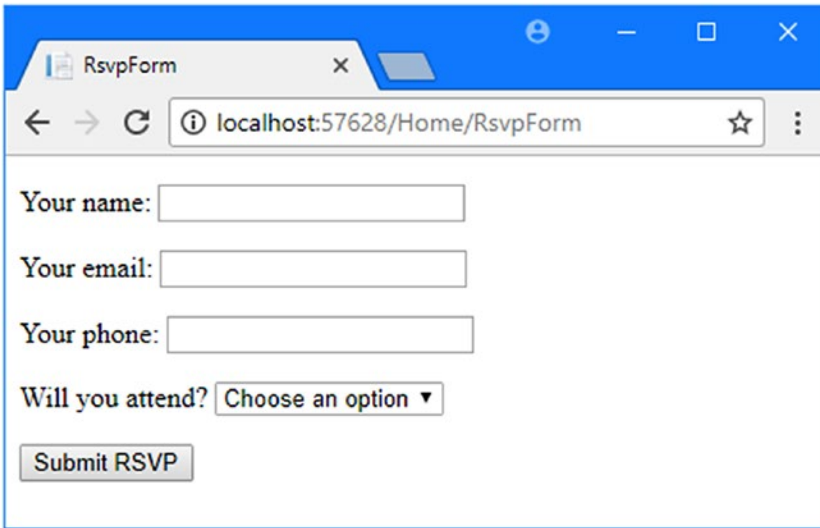

Figure 2-16. *Adding an HTML form to the application*

Receiving Form Data

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. To receive and process submitted form data, I am going to use a core controller feature. I will add a second `RsvpForm` action method 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.

Handing `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 2-13 shows the changes to the `HomeController` class.

Listing 2-13. Adding a Method in the HomeController.cs File in the Controllers Folder

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
```

```
    public class HomeController : Controller {

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

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

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestResponse) {
            // TODO: store response from guest
            return View();
        }
    }
}
```

I have added the HttpGet attribute to the existing RsvpForm action method. This tells MVC that this method should be used only for GET requests. I then added an overloaded version of the RsvpForm method, which accepts a GuestResponse object. I applied the HttpPost attribute to this method, which tells MVC that the new method will deal with POST requests. I explain how these additions to the listing work in the following sections. 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.

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 2-16. 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*, a 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.

Model binding is a powerful and customizable feature that eliminates the grind of dealing with HTTP requests directly and lets you work with C# objects rather than dealing with individual data values sent by the browser. 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 26.

One of the application goals is to present a summary page with details of who is attending, which means that I need to keep track of the responses that I receive. I am going to do this by creating an in-memory collection of objects. This isn't useful in a real application because the response data will be lost when the application is stopped or restarted, but this approach will allow me to keep the focus on MVC and create an application that can easily be reset to its initial state.

■ **Tip** I demonstrate how MVC can be used to store and access data persistently in Chapter 8 as part of a more realistic example application called SportsStore.

I added a file to the project by right-clicking the Models folder and selecting Add ➤ Class from the pop-up menu. I set the name of the file to Repository.cs and used it to define the class shown in Listing 2-14.

Listing 2-14. The Contents of the Repository.cs File in the Models Folder

```
using System.Collections.Generic;

namespace PartyInvites.Models {
    public static class Repository {
        private static List<GuestResponse> responses = new List<GuestResponse>();

        public static IEnumerable<GuestResponse> Responses {
            get {
                return responses;
            }
        }

        public static void AddResponse(GuestResponse response) {
            responses.Add(response);
        }
    }
}
```

The Repository class and its members are set to static, which will make it easy for me to store and retrieve data from different places in the application. MVC provides a more sophisticated approach for defining common functionality, called *dependency injection*, which I describe in Chapter 18, but a static class is a good way to get started for a simple application like this one.

Storing Responses

Now that I have somewhere to store the data, I can update the action method that receives the HTTP POST requests, as shown in Listing 2-15.

Listing 2-15. Updating an Action Method in the HomeController.cs File

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

        [HttpGet]
        public ViewResult RsvpForm() {
```

```
            return View();
        }

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestResponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
        }
    }
}
```

All I have to do to deal with the form data sent in a request is to work with the `GuestResponse` object that is passed to the action method—in this case, to pass it as an argument to the `Repository.AddResponse` method so that the response can be stored.

WHY MODEL BINDING IS NOT LIKE WEB FORMS

In Chapter 1, I explained that one of the disadvantages of traditional ASP.NET Web Forms is that it hides the details of HTTP and HTML from the developers. You may be wondering whether the MVC model binding that I used to create a `GuestResponse` object from an HTTP POST request in Listing 2-15 is doing the same thing.

It isn't. Model binding frees me from the tedious and error-prone task of having to inspect an HTTP request and extract all the data values that I require, but (and this is the important part) if I wanted to process a request manually, I could do so because MVC provides easy access to all of the request data. Nothing is hidden from the developer, but there are a number of useful features that make working with HTTP and HTML simpler and easier; however, using these features is optional.

This may seem like a subtle difference, but as you learn more about MVC, you will see that the development experience is completely different from traditional Web Forms and that you are always aware of how the requests your application receives are handled.

The call to the `View` method in the `RsvpForm` action method tells MVC to render a view called `Thanks` and to pass the `GuestResponse` object to the view. To create the view, right-click the `Views/Home` folder in the Solution Explorer and select Add ➤ New Item from the pop-up menu. Select the MVC View Page template in the ASP.NET category, set the name to `Thanks.cshtml`, and click the Add button. Visual Studio will create the `Views/Home/Thanks.cshtml` file and open it for editing. Change the contents of the file to match Listing 2-16.

Listing 2-16. The Contents of the Thanks.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
```

```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <p>
        <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.
        }
    </p>
    <p>Click <a asp-action="ListResponses">here</a> to see who is coming.</p>
</body>
</html>
```

The `Thanks.cshtml` 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 with which the view is strongly typed.

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 by selecting Start Debugging from the Debug menu, 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 2-17 (although it will differ if your name is not Joe or you said you could not attend).

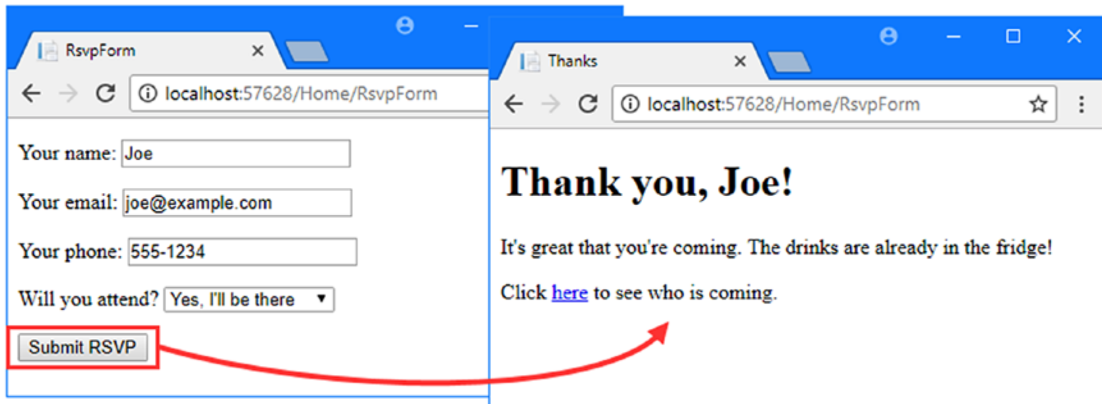

***Figure 2-17.** The Thanks view*

Displaying the Responses

At the end of the `Thanks.cshtml` view, I added an a element to create a link to display the list of people who are coming to the party. I used the `asp-action` tag helper attribute to create a URL that targets an action method called `ListResponses`, like this:

```
...
<p>Click <a asp-action="ListResponses">here</a> to see who is coming.</p>
...
```

If you hover the mouse over the link that is displayed by the browser, you will see that it targets the `/Home/ListResponses` URL. This doesn't correspond to any of the action methods in the `Home` controller, and if you click the link, you will see a 404 Not Found error page

I am going to fix the problem by creating the action method that the URL targets in the `Home` controller, as shown in Listing 2-17.

Listing 2-17. Adding an Action Method in the HomeController.cs File in the Controllers Folder

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

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

        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestResponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
        }

        public ViewResult ListResponses() {
            return View(Repository.Responses.Where(r => r.WillAttend == true));
        }
    }
}
```

The new action method is called `ListResponses`, and it calls the `View` method, using the `Repository.Responses` property as the argument. This is how an action method provides data to a strongly typed view. The collection of `GuestResponse` objects is filtered using LINQ so that only positive responses are used.

The `ListResponses` action method doesn't specify the name of the view that should be used to display the collection of `GuestResponse` objects, which means that the default naming convention will be used and MVC will look for a view called `ListResponses.cshtml` in the `Views/Home` and `Views/Shared` folders. To create the view, right-click the `Views/Home` folder in the Solution Explorer and select Add ➤ New Item from the pop-up menu. Select the MVC View Page template, set the name to `ListResponses.cshtml`, and click the Add button. Edit the contents of the new view to match Listing 2-18.

Listing 2-18. Displaying the Acceptances in the ListResponses.cshtml File in the Views/Home Folder

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Responses</title>
</head>
<body>
    <h2>Here is the list of people attending the party</h2>
    <table>
        <thead>
            <tr>
                <th>Name</th>
                <th>Email</th>
                <th>Phone</th>
            </tr>
        </thead>
        <tbody>
            @foreach (PartyInvites.Models.GuestResponse r in Model) {
                <tr>
                    <td>@r.Name</td>
                    <td>@r.Email</td>
                    <td>@r.Phone</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>
```

Razor view files have the cshtml file extension because they are a mix of C# code and HTML elements. You can see this in Listing 2-18 where I have used a foreach loop to process each of the GuestResponse objects that the action method passes to the view using the View method. Unlike a normal C# foreach loop, the body of a Razor foreach loop contains HTML elements that are added to the response that will be sent back to the browser. In this view, each GuestResponse object generates a tr element that contains td elements populated with the value of an object property.

To see the list at work, run the application by selecting Start Debugging from the Start menu, submit some form data, and then click the link to see the list of responses. You will see a summary of the data you have entered since the application was started, as shown in Figure 2-18. The view does not present the data in an appealing way, but it is enough for the moment, and I will address the styling of the application later in this chapter.

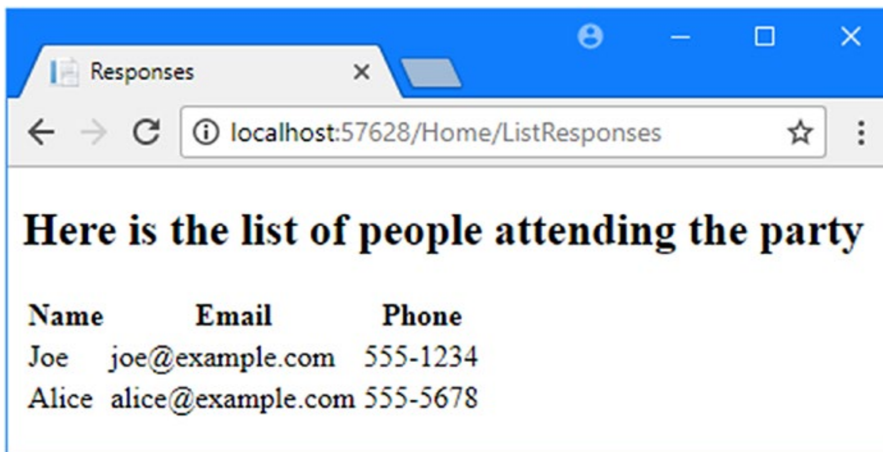

Figure 2-18. *Showing a list of party attendees*

Adding Validation

I am now in a position to add data validation to my application. Without validation, users could enter nonsense data or even submit an empty form. In an MVC application, you typically apply validation to the domain model rather than in the user interface. This means that you define validation in one place, but it takes effect anywhere in the application that the model class is used. 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 2-19 shows how I applied these attributes to the GuestResponse model class.

Listing 2-19. Applying Validation in the GuestResponse.cs File in the Models Folder

```
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; }
    }
}
```

MVC automatically detects the attributes and uses them to validate data during the model-binding process. I imported the namespace that contains the validation attributes, 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 would not be able to tell whether the user had selected a value. A nullable `bool` has three possible values: `true`, `false`, and `null`. The browser sends a `null` value if the user has not selected a value, and this causes the `Required` attribute to report a validation error. This is a nice example of how MVC elegantly blends C# features with HTML and HTTP.

I check to see whether there has been a validation problem using the `ModelState.IsValid` property in the controller class. Listing 2-20 shows how I have done this in the `POST`-enabled `RsvpForm` action method in the `Home` controller class.

Listing 2-20. Checking for Validation Errors in the HomeController.cs File in the Controllers Folder

```
using System;
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {

    public class HomeController : Controller {

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

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

```
        [HttpPost]
        public ViewResult RsvpForm(GuestResponse guestResponse) {
            if (ModelState.IsValid) {
                Repository.AddResponse(guestResponse);
                return View("Thanks", guestResponse);
            } else {
                // there is a validation error
                return View();
            }
        }

        public ViewResult ListResponses() {
            return View(Repository.Responses.Where(r => r.WillAttend == true));
        }
    }
}
```

The Controller base class provides a property called ModelState that provides information about the conversion of HTTP request data into C# objects. If the ModelState.IsValid property returns true, then I know that MVC has been able to satisfy the validation constraints I specified through the attributes on the GuestResponse class. When this happens, I render the Thanks view, just as I did previously.

If the ModelState.IsValid property returns false, then I know that there are validation errors. The object returned by the ModelState property provides details of each problem that has been encountered, but I don't need to get into that level of detail because I can rely on a useful feature that automates the process of asking the user to address any problems by calling the View method without any parameters.

When MVC renders a view, Razor has access to the details of any validation errors associated with the request, and tag helpers can access the details to display validation errors to the user. Listing 2-21 shows the addition of validation tag helper attributes to the RsvpForm view.

Listing 2-21. Adding a Validation Summary to the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <div asp-validation-summary="All"></div>
        <p>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </p>
        <p>
```

```
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </p>
        <p>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" /></p>
        <p>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </p>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

The `asp-validation-summary` attribute is applied to a `div` element, and it displays a list of validation errors when the view is rendered. The value for the `asp-validation-summary` attribute is a value from an enumeration called `ValidationSummary`, which specifies what types of validation errors the summary will contain. I specified `All`, which is a good starting point for most applications, and I describe the other values and explain how they work in Chapter 27.

To see how the validation summary works, run the application, fill out the Name field, and submit the form without entering any other data. You will see a summary of validation errors, as shown in Figure 2-19.

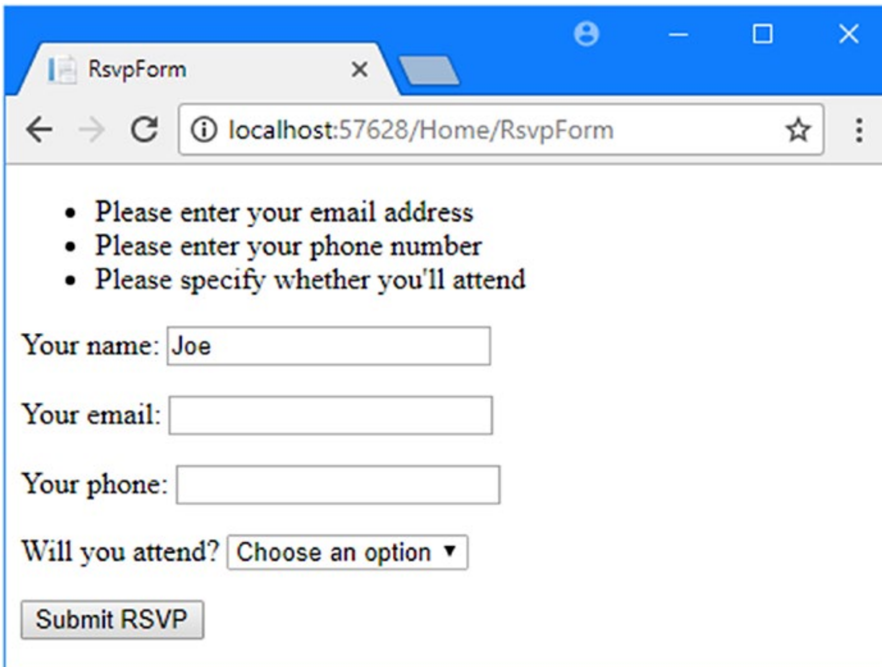

Figure 2-19. *Displaying validation errors*

The RsvpForm action method will not render the Thanks view until all of the validation constraints applied to the GuestResponse class have been satisfied. Notice that the data entered into the Name field was preserved and displayed again when Razor rendered the view with the validation summary. This is another benefit of model binding, 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. MVC model binding is not related to the Web Forms concepts of server controls, postbacks, or View State. MVC does not inject a hidden __VIEWSTATE field into your rendered HTML pages. Instead, it includes the data by setting the value attributes of the input element.

Highlighting Invalid Fields

The tag helper attributes that associate model properties with elements have a handy feature that can be used in conjunction with model binding. When a model class property has failed validation, the helper attributes will generate slightly different HTML. Here is the input element that is generated for the Phone field when there is no validation error:

```
<input type="text" data-val="true" data-val-required="Please enter your phone number"
id="Phone" name="Phone" value="">
```

For comparison, here is the same HTML element after the user has submitted the form without entering any data into the text field (which is a validation error because I applied the Required validation attribute to the Phone property of the GuestResponse class):

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

I have highlighted the difference: the asp-for tag helper attribute added the input element to a class called input-validation-error. I can take advantage of this feature by creating a stylesheet that contains CSS styles for this class and the others that different HTML helper attributes use.

The convention in MVC projects is that static content delivered to clients is placed into the wwwroot folder, organized by content type, so that CSS stylesheets go into the wwwroot/css folder, JavaScript files go into the wwwroot/js folder, and so on.

To create the stylesheet, right-click the wwwroot/css folder in the Visual Studio Solution Explorer, select Add ➤ New Item, navigate to the ASP.NET Core ➤ Web ➤ Content section, and select Style Sheet from the list of templates, as shown in Figure 2-20.

■ **Tip** Visual Studio creates a site.css file in the wwwroot/css folder when a project is created using the Web Application template. You can ignore this file, which I don't use in this chapter.

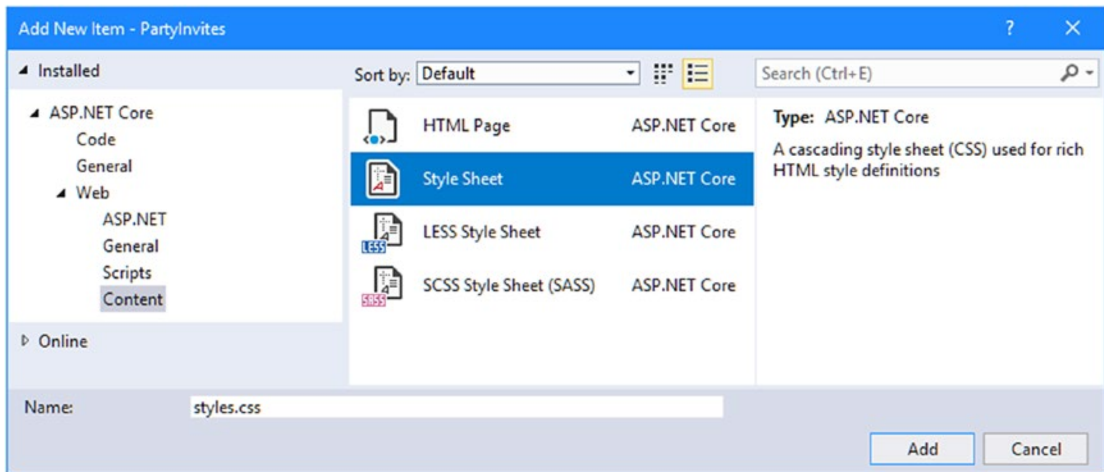

Figure 2-20. *Creating a CSS stylesheet*

Set the name of the file to `styles.css`, click the Add button to create the stylesheet, and edit the new file so that it contains the styles shown in Listing 2-22.

Listing 2-22. The Contents of the styles.css File in the wwwroot/css Folder

```
.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 apply this stylesheet, I have added a link element to the head section of the RsvpForm
view, as shown in Listing 2-23.
```

Listing 2-23. Applying a Stylesheet in the RsvpForm.cshtml File in the Views/Home Folder

```
...
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
</head>
...
```

The `link` element uses the `href` attribute to specify the location of the stylesheet. Notice that the `wwwroot` folder is omitted from the URL. The default configuration for ASP.NET includes support for serving static content, such as images, CSS stylesheets, and JavaScript files, and it maps requests to the `wwwroot` folder automatically. I describe the ASP.NET and MVC configuration process in Chapter 14.

■ **Tip** There is a special tag helper for dealing with stylesheets that can be useful if you have a lot of files to manage. See Chapter 25 for details.

With the application of the stylesheet, a more visually obvious validation error will be displayed when data is submitted that causes a validation error, as shown in Figure 2-21.

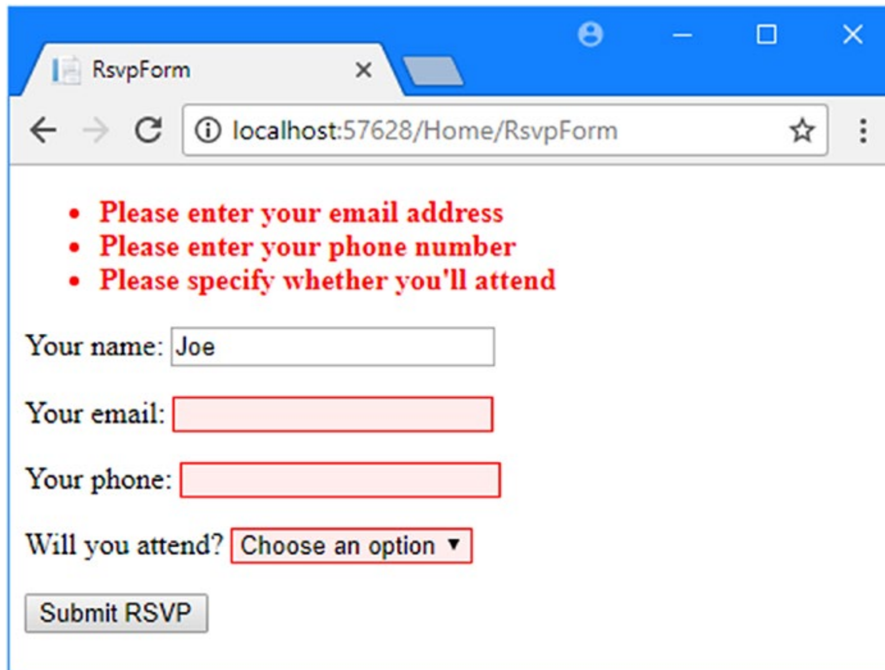

Figure 2-21. *Automatically highlighted validation errors*

Styling the Content

All of the functional goals for the application are complete, but the overall appearance of the application is poor. When you create a project using the Web Application template, as I did for the example in this chapter, Visual Studio installs some common client-side development packages. While I am not a fan of using template projects, I do like the client-side libraries that Microsoft has chosen. One of them is called Bootstrap, which is a nice CSS framework originally developed by Twitter that has become a major open source project in its own right and which has become a mainstay of web application development.

■ **Note** Bootstrap 3 is the current version as I write this, but version 4 is under development. Microsoft may choose to update the version of Bootstrap used by the Web Application template in later releases of Visual Studio, which may cause the content to display differently. This won't be a problem for the other chapters in the book because I show you how to explicitly specify a package version so that you get the expected results.

Styling the Welcome View

The basic Bootstrap features work by applying classes to elements that correspond to CSS selectors defined in the files added to the `wwwroot/lib/bootstrap` 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 `MyView.cshtml` view file in Listing 2-24.

Listing 2-24. Adding Bootstrap to the MyView.cshtml File in the Views/Home Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <div class="text-center">
        <h3>We're going to have an exciting party!</h3>
        <h4>And you are invited</h4>
        <a class="btn btn-primary" asp-action="RsvpForm">RSVP Now</a>
    </div>
</body>
</html>
```

I have added a `link` element whose `href` attribute loads the `bootstrap.css` file from the `wwwroot/lib/bootstrap/dist/css` folder. The convention is that third-party CSS and JavaScript packages are installed into the `wwwroot/lib` folder, and I describe the tool that is used to manage these packages in Chapter 6.

Having imported the Bootstrap stylesheets, I need to style my elements. This is a simple example, so I only need to use a small number of Bootstrap CSS classes: `text-center`, `btn`, and `btn-primary`.

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-primary` class specifies which of a range of colors I want the button to be. You can see the effect by running the application, as shown in Figure 2-22.

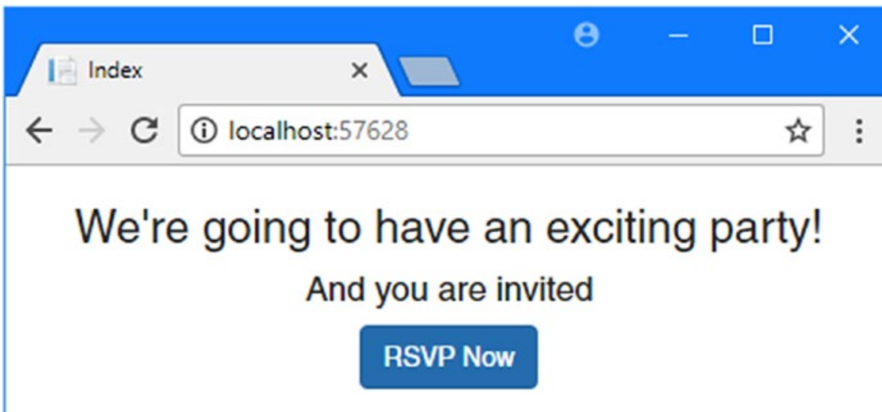

Figure 2-22. *Styling a view*

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 child. 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 2-25.

Listing 2-25. Adding Bootstrap to the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <div class="panel panel-success">
        <div class="panel-heading text-center"><h4>RSVP</h4></div>
        <div class="panel-body">
            <form class="p-a-1" asp-action="RsvpForm" method="post">
                <div asp-validation-summary="All"></div>
                <div class="form-group">
                    <label asp-for="Name">Your name:</label>
                    <input class="form-control" asp-for="Name" />
                </div>
                <div class="form-group">
                    <label asp-for="Email">Your email:</label>
                    <input class="form-control" asp-for="Email" />
                </div>
                <div class="form-group">
                    <label asp-for="Phone">Your phone:</label>
                    <input class="form-control" asp-for="Phone" />
                </div>
                <div class="form-group">
                    <label>Will you attend?</label>
                    <select class="form-control" asp-for="WillAttend">
                        <option value="">Choose an option</option>
                        <option value="true">Yes, I'll be there</option>
                        <option value="false">No, I can't come</option>
                    </select>
```

```
                </div>
                <div class="text-center">
                    <button class="btn btn-primary" type="submit">
                        Submit RSVP
                    </button>
                </div>
            </form>
        </div>
    </div>
</body>
</html>
```

The Bootstrap classes in this example create 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. You can see the effect of the styles in Figure 2-23.

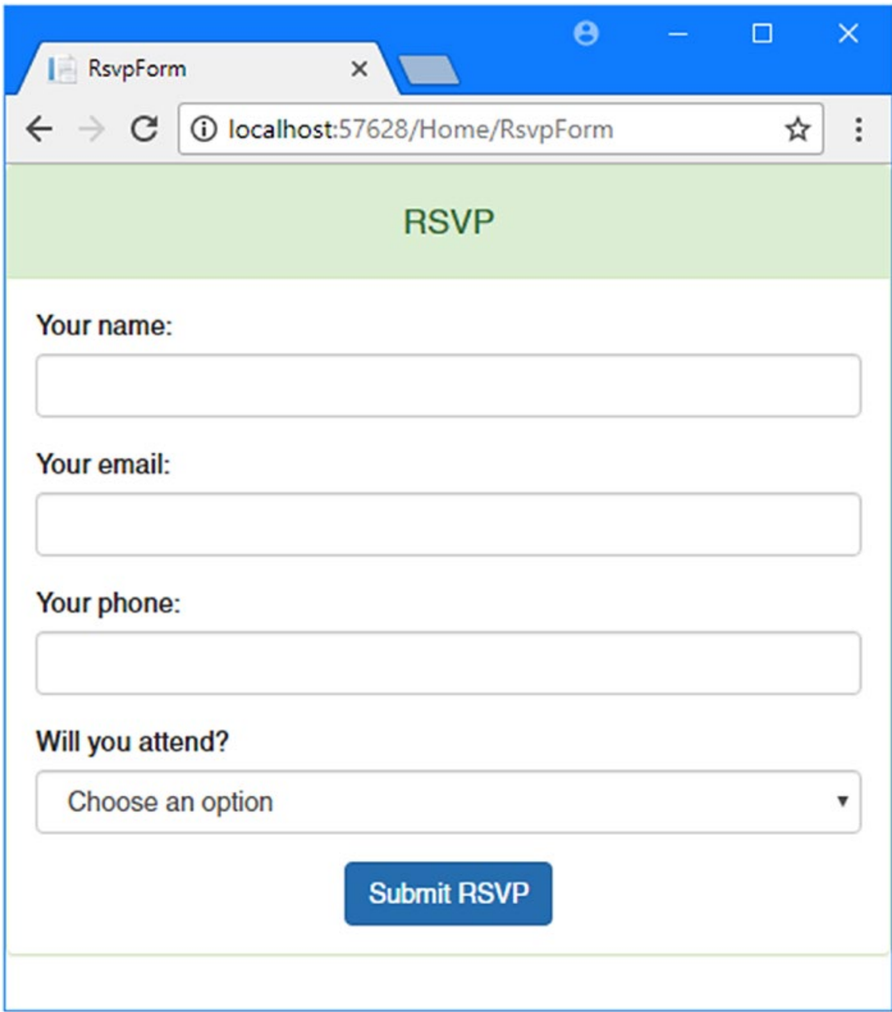

Figure 2-23. *Styling the RsvpForm view*

Styling the Thanks View

The next view file to style is `Thanks.cshtml`, and you can see how I have done this in Listing 2-26, using CSS classes that are similar to the ones I used for the other views. To make an application easier to manage, it is a good principle to avoid duplicating code and markup wherever possible. MVC provides several features to help reduce duplication, which I describe in later chapters. These features include Razor layouts (Chapter 5), partial views (Chapter 21), and view components (Chapter 22).

Listing 2-26. Applying Bootstrap to the Thanks.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="text-center">
    <p>
        <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.
        }
    </p>
    Click <a class="nav-link" asp-action="ListResponses">here</a>
    to see who is coming.
</body>
</html>
```

Figure 2-24 shows the effect of the styles.

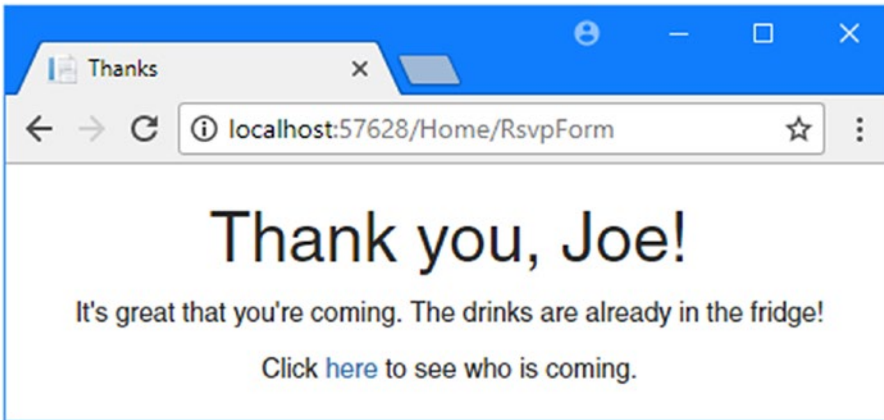

Figure 2-24. *Styling the Thanks view*

Styling the List View

The final view to style is ListResponses, which presents the list of attendees. Styling the content follows the same basic approach as used for all Bootstrap styles, as shown in Listing 2-27.

Listing 2-27. Adding Bootstrap to the ListResponses.cshtml File in the Views/Home Folder

```
@model IEnumerable<PartyInvites.Models.GuestResponse>

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
    <title>Responses</title>
</head>
<body>
    <div class="panel-body">
        <h2>Here is the list of people attending the party</h2>
        <table class="table table-sm table-striped table-bordered">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Email</th>
                    <th>Phone</th>
                </tr>
            </thead>
            <tbody>
```

```
                @foreach (PartyInvites.Models.GuestResponse r in Model) {
                    <tr>
                        <td>@r.Name</td>
                        <td>@r.Email</td>
                        <td>@r.Phone</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

Figure 2-25 shows the way that the table of attendees is presented. Adding these styles to the view completes the example application, which now meets all of the development goals and has an improved appearance.

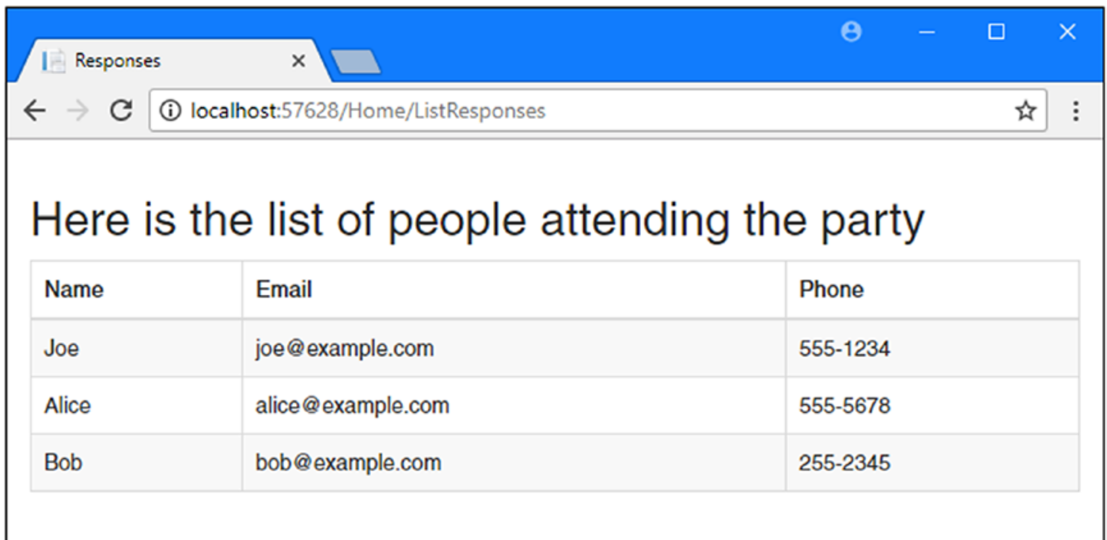

Figure 2-25. *Styling the ListResponses view*

Summary

In this chapter, I created a new MVC project and used it to construct a simple data-entry application, giving you a first glimpse of the ASP.NET Core MVC architecture and approach. I skipped some key features (including Razor syntax, routing, and testing), but I return to these topics in depth in later chapters. In the next chapter, I describe the MVC design patterns, which form the foundation for effective development with ASP.NET Core MVC.