2

Controllers

—by Jon Galloway

WHAT'S IN THIS CHAPTER?

- Understanding the controller's role
- Setting up a sample application: The MVC Music Store
- Controller 101

This chapter explains how controllers respond to user HTTP requests and return information to the browser. It focuses on the function of controllers and controller actions. We haven't covered views and models yet, so our controller action samples will be a little high level. This chapter lays the groundwork for the following several chapters.

Chapter 1 discussed the Model-View-Controller (MVC) pattern in general and then followed up with how ASP.NET MVC compares with ASP.NET Web Forms. Now it's time to get into a bit more detail about one of the core elements of the three-sided pattern that is MVC—the controller.

THE CONTROLLER'S ROLE

Starting out with a quick definition and then diving into detail from there is probably best. Keep this definition in mind while reading this chapter. It can help to ground the discussion ahead with what a controller is all about and what it's supposed to do.

Controllers within the MVC pattern are responsible for responding to user input, often making changes to the model in response to user input. In this way, controllers in the MVC pattern are concerned with the flow of the application, working with data coming in, and providing data going out to the relevant view.

Way back in the day, web servers served up HTML stored in static files on disk. As dynamic web pages gained prominence, web servers served HTML generated on the fly from dynamic scripts

that were also located on disk. With MVC, it's a little different. The URL tells the routing mechanism (which you'll begin to explore in the next few chapters, and learn about in depth in Chapter 9) which controller class to instantiate and which action method to call, and supplies the required arguments to that method. The controller's method then decides which view to use, and that view then renders the HTML.

Rather than having a direct relationship between the URL and a file living on the web server's hard drive, a relationship exists between the URL and a method on a controller class. ASP.NET MVC implements the front controller variant of the MVC pattern, and the controller sits in front of everything except the routing subsystem, as discussed in Chapter 9.

A good way to think about how MVC works in a web scenario is that MVC serves up the results of method calls, not dynamically generated (also known as scripted) pages.

A BRIEF HISTORY OF CONTROLLERS

The MVC pattern has been around for a long time—decades before this era of modern web applications. When MVC first developed, graphical user interfaces (GUIs) were just a few years old, and the interaction patterns were still evolving. Back then, when the user pressed a key or clicked the screen, a process would "listen," and that process was the controller. The controller was responsible for receiving that input, interpreting it and updating whatever data class was required (the model), and then notifying the user of changes or program updates (the view, which Chapter 3 covers in more detail).

In the late 1970s and early 1980s, researchers at Xerox PARC (which, coincidentally, was where the MVC pattern was incubated) began working with the notion of the GUI, wherein users "worked" within a virtual "desktop" environment on which they could click and drag items around. From this came the idea of *event-driven programming*—executing program actions based on events fired by a user, such as the click of a mouse or the pressing of a key on the keypad.

Over time, as GUIs became the norm, it became clear that the MVC pattern wasn't entirely appropriate for these new systems. In such a system, the GUI components themselves handled user input. If a button was clicked, it was the button that responded to the mouse click, not a controller. The button would, in turn, notify any observers or listeners that it had been clicked. Patterns such as the Model-View-Presenter (MVP) proved to be more relevant to these modern systems than the MVC pattern.

ASP.NET Web Forms is an event-based system, which is unique with respect to web application platforms. It has a rich control-based, event-driven programming model that developers code against, providing a nice componentized GUI for the Web. When a button is clicked, a button control responds and raises an event on the server indicating that it has been clicked. The beauty of this approach is that it allows the developer to work at a higher level of abstraction when writing code.

Digging under the hood a bit, however, reveals that a lot of work is going on to simulate that componentized event-driven experience. At its core, when a button is clicked, the browser submits a request to the server containing the state of the controls on the page encapsulated in an encoded hidden input. On the server side, in response to this request, ASP.NET has to rebuild the entire control hierarchy and then interpret that request, using the contents of that request to restore the current state of the application for the current user. All this happens because the Web, by its nature, is stateless. With a rich-client Windows GUI app, no need exists to rebuild the entire screen and control hierarchy every time the user clicks a UI widget, because the application doesn't go away.

With the Web, the state of the app for the user essentially vanishes and then is restored with every click. Well, that's an oversimplification, but the user interface, in the form of HTML, is sent to the browser from the server. This raises the question: "Where is the application?" For most web pages, the application is a dance between client and server, each maintaining a tiny bit of state, perhaps a cookie on the client or chunk of memory on the server, all carefully orchestrated to cover up the Tiny Lie. The Lie is that the Internet and HTTP can be programmed again in a stateful manner.

The underpinning of event-driven programming (the concept of *state*) is lost when programming for the Web, and many are not willing to embrace the Lie of a *virtually stateful* platform. Given this, the industry has seen the resurgence of the MVC pattern, albeit with a few slight modifications.

One example of such a modification is that in traditional MVC, the model can "observe" the view via an indirect association to the view. This allows the model to change itself based on view events. With MVC for the Web, by the time the view is sent to the browser, the model is generally no longer in memory and does not have the ability to observe events on the view. (Note that exceptions to this change exist, as described in Chapter 8, regarding the application of Ajax to MVC.)

With MVC for the Web, the controller is once again at the forefront. Applying this pattern requires that every user input to a web application simply take the form of a request. For example, with ASP.NET MVC, each request is routed (using routing, discussed in Chapter 9) to a method on a controller (called an *action*). The controller is entirely responsible for interpreting that request, manipulating the model if necessary, and then selecting a view to send back to the user via the response.

With that bit of theory out of the way, let's dig into ASP.NET MVC's specific implementation of controllers. You'll be continuing from the new project you created in Chapter 1. If you skipped over that, you can just create a new MVC 5 application using the Internet Application template and the Razor view engine, as shown in Figure 1-9 in the previous chapter.

A SAMPLE APPLICATION: THE MVC MUSIC STORE

As mentioned in Chapter 1, we will use the MVC Music Store application for a lot of our samples in this book. You can find out more about the MVC Music Store application at `http://mvcmusicstore.codeplex.com`. The Music Store tutorial is intended for beginners and moves at a pretty slow pace; because this is a Professional Series book, we'll move faster and cover some more advanced background detail. If you want a slower, simpler introduction to any of these topics, feel free to refer to the MVC Music Store tutorial. It's available online in HTML format and as a 150-page downloadable PDF. MVC Music Store was published under the Creative Commons license to allow for free reuse, and we'll be referencing it at times.

The MVC Music Store application is a simple music store that includes basic shopping, checkout, and administration, as shown in Figure 2-1.

FIGURE 2-1

The following store features are covered:

- **Browse:** Browse through music by genre and artist, as shown in Figure 2-2.

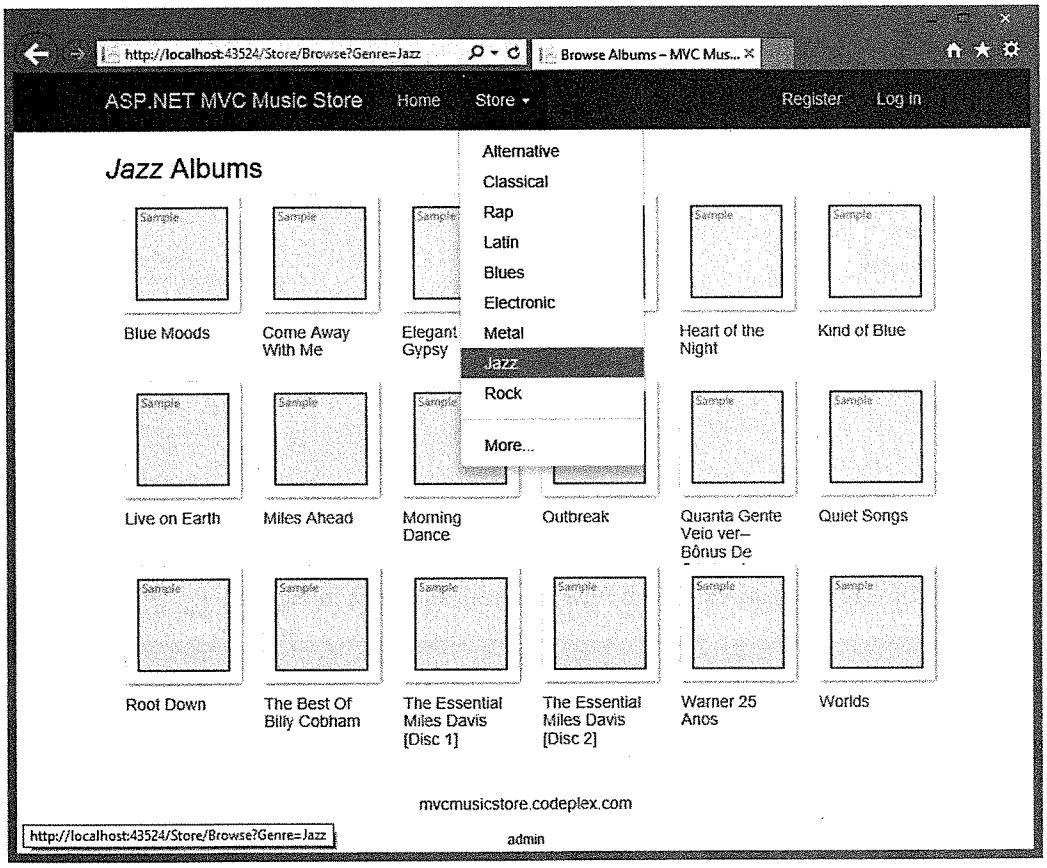

FIGURE 2-2

- **Add:** Add songs to your cart, as shown in Figure 2-3.

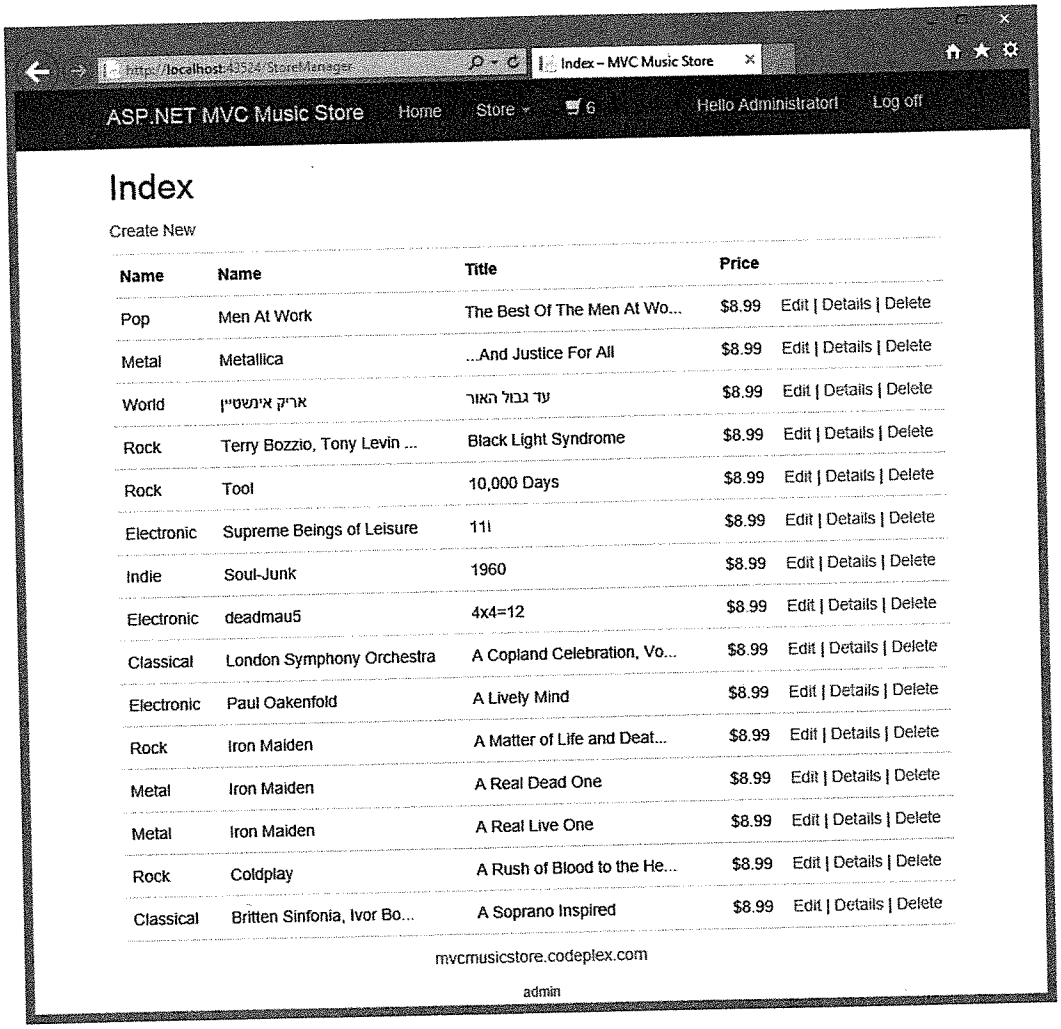

FIGURE 2-6

CONTROLLER BASICS

Getting started with MVC presents something of a chicken and egg problem: There are three parts (model, view, and controller) to understand, and really digging into one of those parts without understanding the others is difficult. To get started, you'll first learn about controllers at a very high level, ignoring models and views for a bit.

After learning the basics of how controllers work, you'll be ready to learn about views, models, and other ASP.NET MVC development topics at a deeper level. You'll then be ready to circle back to advanced controller topics in Chapter 15.

A Simple Example: The Home Controller

Before writing any real code, let's start by looking at what's included by default in a new project. Projects created using the MVC template with Individual User Accounts include two controller classes:

- **`HomeController`:** Responsible for the "home page" at the root of the website, as well as an "about page" and a "contact page"
- **`AccountController`:** Responsible for account-related requests, such as login and account registration

In the Visual Studio project, expand the `/Controllers` folder and open `HomeController.cs`, as shown in Figure 2-7.

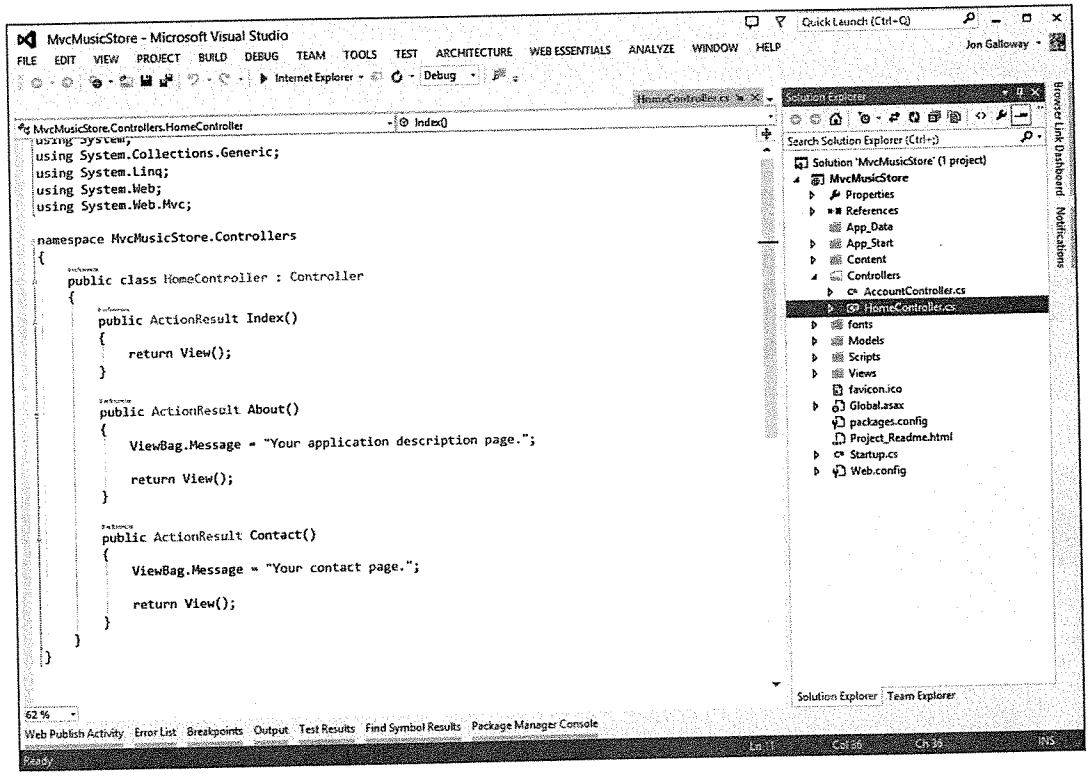

FIGURE 2-7

Notice that this is a pretty simple class that inherits from the `Controller` base class. The `Index` method of the `HomeController` class is responsible for deciding what happens when you browse to the homepage of the website. Follow these steps to make a simple edit and run the application:

1. Replace "Your application description page." in the `About` method with the phrase of your choice—perhaps, "I like cake!:

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

```
using System.Web;
using System.Web.Mvc;

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

        public ActionResult About()
        {
                    ViewBag.Message = "I like cake!";
            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";
            return View();
        }
    }
}
```

2. Run the application by pressing the F5 key (or by using the Debug ⇨ Start Debugging menu item, if you prefer). Visual Studio compiles the application and launches the site running under IIS Express.

IIS EXPRESS AND ASP.NET DEVELOPMENT SERVER

Visual Studio 2013 includes IIS Express, a local development version of IIS, which will run your website on a random free "port" number. In Figure 2-8, the site is running at `http://localhost:26641/`, so it's using port 26641. Your port number will be different. When we talk about URLs such as `/Store/Browse` in this tutorial, that will go after the port number. Assuming a port number of 26641, browsing to `/Store/Browse` will mean browsing to `http://localhost:26641/Store/Browse`.

Visual Studio 2010 and below use the Visual Studio Development Server (sometimes referred to by its old codename, Cassini) rather than IIS Express. Although the Development Server is similar to IIS, IIS Express actually is a version of IIS that has been optimized for development purposes. You can read more about using IIS Express on Scott Guthrie's blog at `http://weblogs.asp.net/scottgu/7673719.aspx`.

3. A browser window opens and the home page of the site appears, as shown in Figure 2-8.

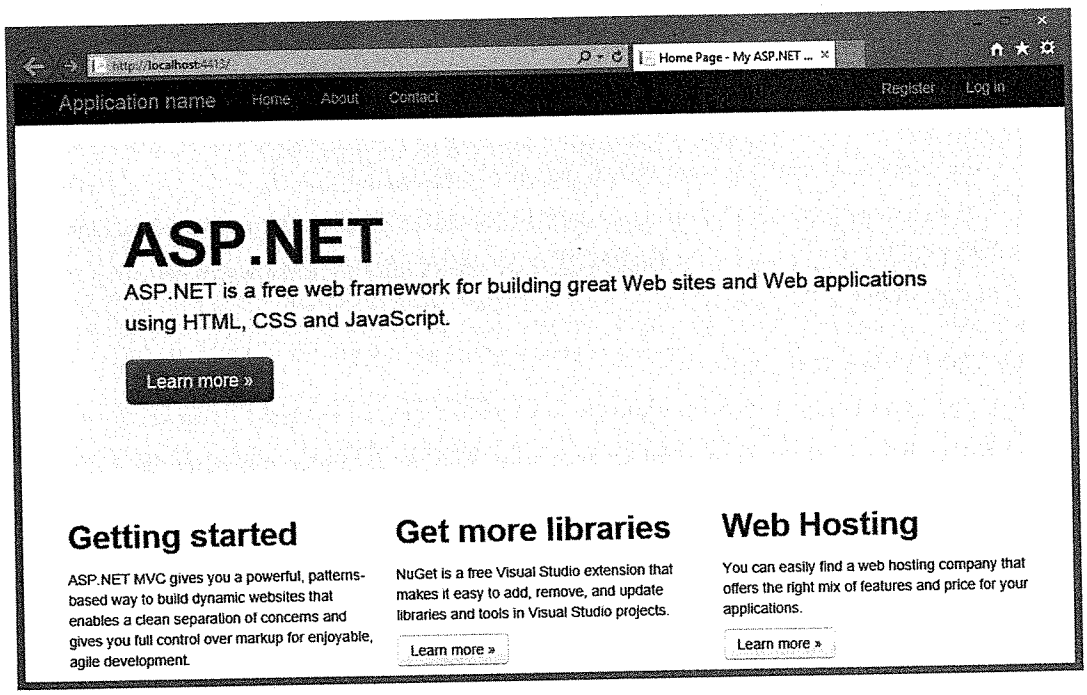

FIGURE 2-8

4. Navigate to the About page by browsing to `/Home/About` (or by clicking the About link in the header). Your updated message displays, as shown in Figure 2-9.

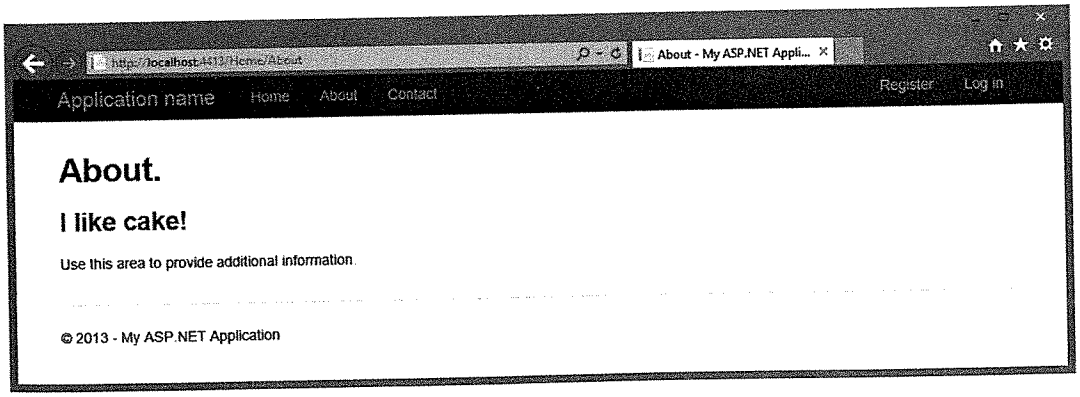

FIGURE 2-9

Great—you created a new project and put some words on the screen! Now let's get to work on building an actual application by creating a new controller.

Writing Your First Controller

In this section, you'll create a controller to handle URLs related to browsing through the music catalog. This controller will support three scenarios:

- The index page lists the music genres that your store carries.
- Clicking a genre leads to a browse page that lists all the music albums in a particular genre.
- Clicking an album leads to a details page that shows information about a specific music album.

Creating the New Controller

To create the controller, you start by adding a new `StoreController` class. To do so:

1. Right-click the `Controllers` folder within the Solution Explorer and select the Add ⇨ Controller menu item, as shown in Figure 2-10.

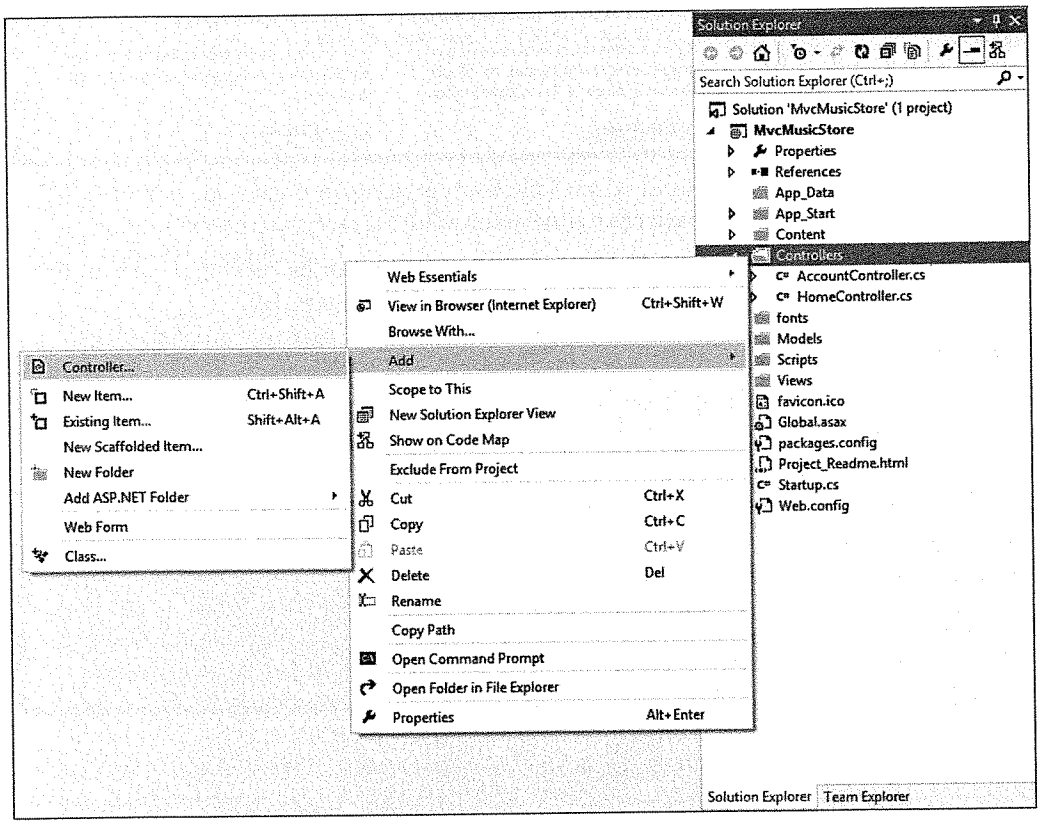

FIGURE 2-10

2. Select the MVC 5 Controller - Empty scaffolding template, as shown in Figure 2-11.

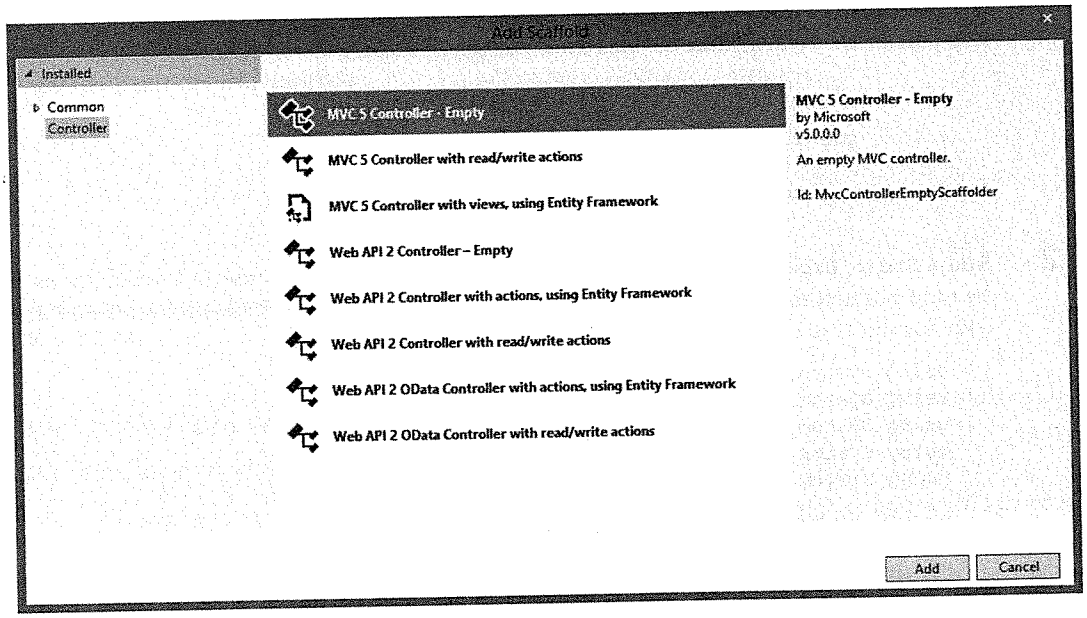

FIGURE 2-11

3. Name the controller **StoreController** and press the Add button, as shown in Figure 2-12.

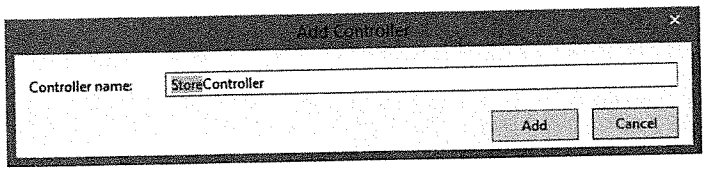

FIGURE 2-12

Writing Your Action Methods

Your new `StoreController` already has an `Index` method. You'll use this `Index` method to implement your listing page that lists all genres in your music store. You'll also add two additional methods to implement the two other scenarios you want your `StoreController` to handle: `Browse` and `Details`.

These methods (`Index`, `Browse`, and `Details`) within your controller are called *controller actions*. As you've already seen with the `HomeController.Index` action method, their job is to respond to URL requests, perform the appropriate actions, and return a response back to the browser or user that invoked the URL.

To get an idea of how a controller action works, follow these steps:

1. Change the signature of the `Index` method to return a string (rather than an `ActionResult`) and change the return value to `"Hello from Store.Index()"` as follows:

```
//
// GET: /Store/
public string Index()
{
    return "Hello from Store.Index()";
}
```

2. Add a `Store.Browse` action that returns `"Hello from Store.Browse()"` and a `Store.Details` action that returns `"Hello from Store.Details()"`, as shown in the complete code for the `StoreController` that follows:

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

namespace MvcMusicStore.Controllers
{
    public class StoreController : Controller
    {
        //
        // GET: /Store/
        public string Index()
        {
            return "Hello from Store.Index()";
        }
        //
        // GET: /Store/Browse
        public string Browse()
        {
            return "Hello from Store.Browse()";
        }
        //
        // GET: /Store/Details
        public string Details()
        {
            return "Hello from Store.Details()";
        }
    }
}
```

3. Run the project again and browse the following URLs:
   - `/Store`
   - `/Store/Browse`
   - `/Store/Details`

Accessing these URLs invokes the action methods within your controller and returns string responses, as shown in Figure 2-13.

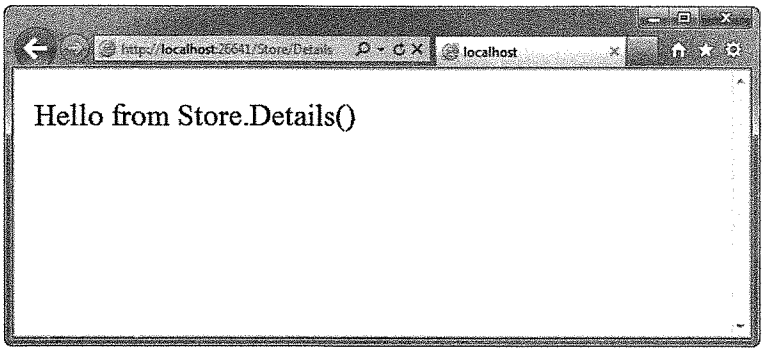

FIGURE 2-13

A Few Quick Observations

Let's draw some conclusions from this quick experiment:

- Browsing to `/Store/Details` caused the `Details` method of the `StoreController` class to be executed, without any additional configuration. This is routing in action. We'll talk a little more about routing later in this chapter and go into detail in Chapter 9.
- Though we used Visual Studio tooling to create the controller class, it's a very simple class. The only way you would know from looking that it was a controller class was that it *inherits from* `System.Web.Mvc.Controller`.
- We've put text in a browser with just a controller—we didn't use a model or a view. Although models and views are incredibly useful within ASP.NET MVC, controllers are really at the heart. Every request goes through a controller, whereas some will not need to make use of models and views.

Parameters in Controller Actions

The previous examples have been of writing out constant strings. The next step is to make them dynamic actions by reacting to parameters that are passed in via the URL. You can do so by following these steps:

1. Change the `Browse` action method to retrieve a query string value from the URL. You can do this by adding a "genre" parameter of type `string` to your action method. When you do this, ASP.NET MVC automatically passes any query string or form post parameters named "genre" to your action method when it is invoked.

```
//
// GET: /Store/Browse?genre=?Disco
public string Browse(string genre)
{
    string message =
        HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);

    return message;
}
```

HTML ENCODING USER INPUT

We're using the `HttpUtility.HtmlEncode` utility method to sanitize the user input. This prevents users from injecting JavaScript code or HTML markup into our view with a link like `/Store/Browse?Genre=<script>window.location='http://hacker.example.com'</script>`.

2. Browse to `/Store/Browse?Genre=Disco`, as shown in Figure 2-14.

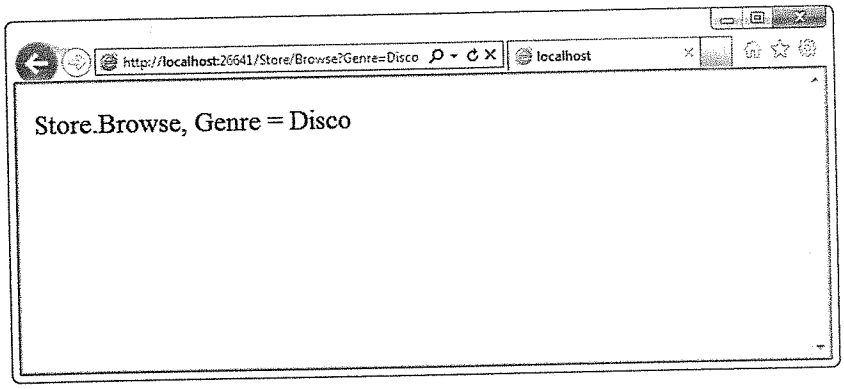

FIGURE 2-14

This shows that your controller actions can read a query string value by accepting it as a parameter on the action method.

3. Change the `Details` action to read and display an input parameter named ID. Unlike the previous method, you won't be embedding the ID value as a query string parameter. Instead you'll embed it directly within the URL itself. For example: `/Store/Details/5`.

   ASP.NET MVC lets you easily do this without having to configure anything extra. ASP.NET MVC's default routing convention is to treat the segment of a URL after the action method name as a parameter named `ID`. If your action method has a parameter named ID, then ASP.NET MVC automatically passes the URL segment to you as a parameter.

```
//
// GET: /Store/Details/5
public string Details(int id)
{
    string message = "Store.Details, ID = " + id;

    return message;
}
```

4. Run the application and browse to `/Store/Details/5`, as shown in Figure 2-15.

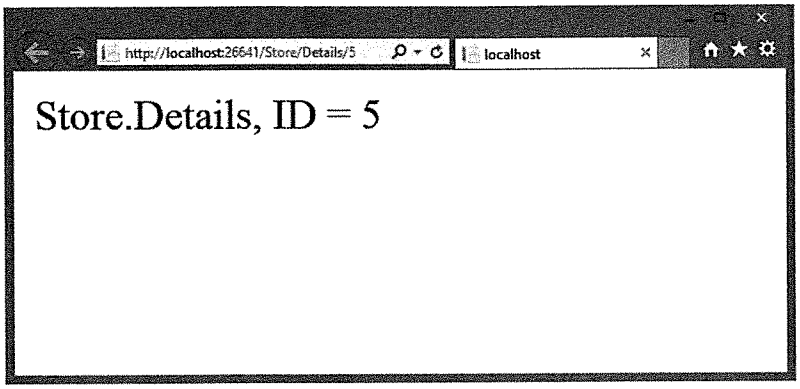

FIGURE 2-15

As the preceding examples indicate, you can look at controller actions as if the web browser were directly calling methods on your controller class. The class, method, and parameters are all specified as path segments or query strings in the URL, and the result is a string that's returned to the browser. That's a huge oversimplification, ignoring things such as:

- The way routing maps the URL to actions.
- The fact that you'll almost always use views as templates to generate the strings (usually HTML) to be returned to the browser.
- The fact that actions rarely return raw strings; they usually return the appropriate `ActionResult`, which handles things such as HTTP status codes, calling the View templating system, and so on.

Controllers offer a lot of opportunities for customization and extensibility, but you'll probably find that you rarely—if ever—need to take advantage of that fact. In general use, controllers are called via a URL, they execute your custom code, and they return a view. With that in mind, we'll defer our look at the gory details behind how controllers are defined, invoked, and extended. You can find those, with other advanced topics, discussed in Chapter 15. You've learned enough about the basics of how controllers work to throw views into the mix, and we cover those in Chapter 3.

SUMMARY

Controllers are the conductors of an MVC application, tightly orchestrating the interactions of the user, the model objects, and the views. They are responsible for responding to user input, manipulating the appropriate model objects, and then selecting the appropriate view to display back to the user in response to the initial input.

In this chapter, you learned the fundamentals of how controllers work in isolation from views and models. With this basic understanding of how your application can execute code in response to URL requests, you're ready to tackle the user interface. We'll look at that next.