CHAPTER 17

Controllers and Actions

Every request that comes to your application is handled by a controller. The controller is free to handle the request any way it sees fit, as long as it doesn't stray into the areas of responsibility that belong to the model and view. This means that controllers do not contain or store data, nor do they generate user interfaces.

In the ASP.NET MVC Framework, controllers are .NET classes that contain the logic required to handle a request. In Chapter 3, I explained that the role of the controller is to encapsulate your application logic. This means that controllers are responsible for processing incoming requests, performing operations on the domain model, and selecting views to render to the user. In this chapter, I show you how controllers are implemented and the different ways that you can use controllers to receive and generate output. Table 17-1 provides the summary for this chapter.

Table 17-1. *Chapter Summary*

Problem	Solution	Listing
Create a controller	Implement the `IController` interface or derive from the `Controller` class	1–4
Get information about a request	Use the context objects and properties or define action method parameters	5, 6
Generate a response from controller that directly implements the `IController` interface	Use the `HttpResponse` context object	7–8
Generate a response from a controller derived from the `Controller` class	Use an action result	9–12
Tell the MVC Framework to render a view	Use a `ViewResult`	13, 14
Pass data from the controller to the view	Use a view model object or the view bag	15–19
Redirect the browser to a new URL	Use the `Redirect` or `RedirectPermanent` methods	20–21
Redirect the browser to a URL generated by a route	Use the `RedirectToRoute` or `RedirectToRoutePermanent` methods	22
Redirect the browser to another action method	Use the `RedirectToAction` method	23
Send an HTTP result code to the browser	Return an `HttpStatusCodeResult` object or use one of the convenience methods such as `HttpNotFound`.	24–26

Preparing the Example Project

To prepare for this chapter, I created a new project called ControllersAndActions using the Empty template, checking the option for the MVC folders and references to create a unit test project called ControllersAndActions.Tests. The unit tests that I create in this chapter don't need mock implementations and so I don't need to install the Moq package, but I do need to install the MVC package so that my tests have access to the base controller classes. Enter the following command into the Visual Studio NuGet Package Manager Console:

```
Install-Package Microsoft.Aspnet.Mvc -version 5.0.0 -projectname
     ControllersAndActions.Tests
```

Setting the Start URL

Once you have created the project, select ControllersAndActions Properties from the Visual Studio Project menu, switch to the Web tab and check the Specific Page option in the Start Action section. You don't have to provide a value, just check the option.

Introducing the Controller

You have seen the use of controllers in almost all of the chapters so far. Now it is time to take a step back and look behind the scenes.

Creating a Controller with IController

In the MVC Framework, controller classes must implement the IController interface from the System.Web.Mvc namespace, which I have shown in Listing 17-1.

Listing 17-1. The System.Web.Mvc.IController Interface

```
public interface IController {
    void Execute(RequestContext requestContext);
}
```

■ **Tip** I got the definition of this interface by downloading the MVC Framework source code, which is endlessly useful for figuring out how things work behind the curtain. You can download the source code from http://aspnet.codeplex.com.

This is a simple interface. The sole method, Execute, is invoked when a request is targeted at the controller class. The MVC Framework knows which controller class has been targeted in a request by reading the value of the controller property generated by the routing data, or through your custom routing classes as described in the Chapters 15 and 16.

You can create controller classes by implementing IController, but it is a low-level interface and you must do a lot of work to get anything useful done. That said, the IController interface makes for a useful demonstration of how controllers operate and, to that end, I created a new class file called BasicController.cs in the Controllers folder with the content shown in Listing 17-2.

Listing 17-2. The Contents of the BasicController.cs File

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

namespace ControllersAndActions.Controllers {

    public class BasicController : IController {

        public void Execute(RequestContext requestContext) {

            string controller = (string)requestContext.RouteData.Values["controller"];
            string action = (string)requestContext.RouteData.Values["action"];

                  requestContext.HttpContext.Response.Write(
                string.Format("Controller: {0}, Action: {1}", controller, action));
        }
    }
}
```

The `Execute` method of the `IController` interface is passed to a `System.Web.Routing.RequestContext` object that provides information about the current request and the route that matched it (and led to this controller being invoked to process that request). The `RequestContext` class defines two properties, which I have described in Table 17-2.

Table 17-2. *The Properties Defined by the RequestContext Class*

Name	Description
`HttpContext`	Returns an `HttpContextBase` object that describes the current request
`RouteData`	Returns a `RouteData` object that describes the route that matched the request

The `HttpContextBase` object provides access to a set of objects that describe the current request, known as the *context objects*, and which I'll come back to later in the chapter. The `RouteData` object describes the route. I have described the important `RouteData` properties in Table 17-3.

Table 17-3. *The Properties Defined by the RouteData Class*

Name	Description
`Route`	Returns the `RouteBase` implementation that matched the route
`RouteHandler`	Returns the `IRouteHandler` that handled the route
`Values`	Returns a collection of segment values, indexed by name

CLASS NAMES THAT END WITH BASE

The MVC Framework relies on the ASP.NET platform to process requests, which makes a lot of sense because it is proven, feature-rich and integrates well into the IIS application server. One problem is that the classes that the ASP.NET platform uses to provide information about requests are not well-suited to unit testing, a key benefit of using the MVC Framework. Microsoft needed to introduce testability while maintaining compatibility with existing ASP.NET Web Forms applications and so introduced the *`Base` classes*, so-called because they have the same names as core ASP.NET platform classes followed by the word `Base`. So, for example, the ASP.NET platform provides context information about the current request and some key application services through an `HttpContext` object. The `Base` class counterpart is `HttpContextBase`, an instance of which is passed to the `Execute` method defined by the `IController` interface (and you'll see other `Base` classes in the examples that follow). The original and the `Base` classes define the same properties and methods, but the Base classes are always abstract, which means that they can easily be used for unit testing.

Sometime you will receive an instance of one of the original ASP.NET classes, such as `HttpContext` but need to create an MVC-friendly `Base` class, such as `HttpContextBase`. You can do this using one of the `Wrapper` classes, which have the same name as the original classes plus the word `Wrapper`, such as `HttpContextWrapper`. The wrapper classes are derived from the `Base` classes and have constructors that accept an instance of the original class, like this:

```
...
HttpContext myContext = getOriginalObjectFromSomewhere();
HttpContextBase myBase = new HttpContextWrapper(myContext);
...
```

There are original, `Base` and `Wrapper` classes throughout the `System.Web` namespace so that ASP.NET can support the MVC Framework and older Web Forms applications seamlessly.

I showed you how to use the `RouteBase` and `IRouteHandler` types to customize the routing system in Chapter 16. In this example, I use the `Values` property to get the values of the `controller` and `action` segment variables and write them to the response.

■ **Note** Part of the problem when creating custom controllers is that you don't have access to features like views. This means that you have to work at a lower level and is the reason that I write my content directly to the client. The `HttpContextBase.Response` property returns an `HttpResponseBase` object that allows you to configure and add to the response that will be sent to the client. This is another touch-point between the ASP.NET platform and the MVC Framework and one that I describe in depth in the *Pro ASP.NET MVC 5 Framework Platform* book, which will be published by Apress in 2014.

If you run the application and navigate to `/Basic/Index`, you can see the output generated by the custom controller, as shown in Figure 17-1.

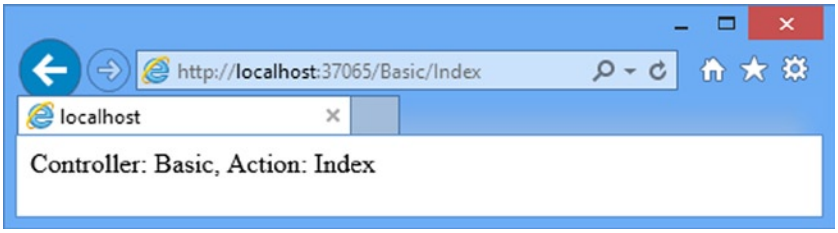

Figure 17-1. *A result generated from the BasicController class*

Implementing the `IController` interface allows you to create a class that the MVC Framework recognizes as a controller and sends requests to, without any limitation on how the request is processed and responded to. This is a nice example because it shows you how extensible the MVC Framework is, even for key building blocks like controllers, but it would be hard to write a complex application this way.

Creating a Controller by Deriving from the Controller Class

As the previous example suggested, the MVC Framework is endlessly customizable and extensible. You can implement the `IController` interface to create any kind of request handling and result generation you require. Don't like action methods? Don't care for rendered views? Then you can just take matters in your own hands and write a better, faster, and more elegant way of handling requests. Or you can build on the features that the Microsoft MVC Framework team has provided, which is achieved by deriving your controllers from the `System.Web.Mvc.Controller` class.

`System.Web.Mvc.Controller` is the class that provides the request handling support that most MVC developers will be familiar with. It is the class I have been using in all of the examples in previous chapters. The `Controller` class provides three key features:

- *Action methods*: A controller's behavior is partitioned into multiple methods (instead of having just one single `Execute()` method). Each action method is exposed on a different URL and is invoked with parameters extracted from the incoming request.
- *Action results*: You can return an object describing the result of an action (for example, rendering a view, or redirecting to a different URL or action method), which is then carried out on your behalf. The separation between *specifying results* and *executing them* simplifies unit testing.
- *Filters*: You can encapsulate reusable behaviors (for example, authentication, as you saw in Chapter 12) as filters, and then tag each behavior onto one or more controllers or action methods by putting an attribute in your source code.

Unless you have a specific requirement in mind, the best way to create controllers is to derive from the `Controller` class and, as you might hope, this is what Visual Studio does when it creates a new controller in response to the `Add ➤ Scaffold` menu item. Listing 17-3 shows a simple controller created this way, called `DerivedController`, generated using the `MVC 5 Controller – Empty` option, with some simple changesto set a `ViewBag` property and select a view.

Listing 17-3. The ontents of the DerivedController.cs File

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

```
namespace ControllersAndActions.Controllers {
    public class DerivedController : Controller {
        public ActionResult Index() {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }
    }
}
```

The `Controller` class is also the link to the Razor view system. In the listing, I return the result of the `View` method, passing in the name of the view I want rendered to the client as a parameter. To create this view, create the `Views/Derived` folder, right-click on it and select `Add ➤ MVC 5 View Page (Razor)` from the menu. Set the name to `MyView.cshtml` and click the `OK` button to create the view file. Set the content of the view file to match Listing 17-4.

Listing 17-4. The Contents of the MyView.cshtml File

```
@{
    ViewBag.Title = "MyView";
}

<h2>MyView</h2>

Message: @ViewBag.Message
```

If you start the application and navigate to `/Derived/Index`, the action method will be invoked and the view I named will be rendered, as shown in Figure 17-2.

Figure 17-2. *A result generated from the DerivedController class*

The job of a derivation of the `Controller` class is to implement action methods, obtain whatever input is needed to process a request, and generate a suitable response. The myriad of ways to do this are covered in the rest of this chapter.

Receiving Request Data

Controllers frequently need to access data from the incoming request, such as query string values, form values, and parameters parsed from the URL by the routing system. There are three main ways to access that data:

- Extract it from a set of *context objects*.
- Have the data passed as *parameters* to your action method.
- Explicitly invoke the framework's *model binding* feature.

Here, I look at the approaches for getting input for your action methods, focusing on using context objects and action method parameters. In Chapter 24, I cover model binding in depth.

Getting Data from Context Objects

When you create a controller by deriving from the `Controller` base class, you get access to a set of convenience properties to access information about the request. These properties include `Request`, `Response`, `RouteData`, `HttpContext`, and `Server`. Each provides information about a different aspect of the request. I refer to these as *convenience properties*, because they each retrieve different types of data from the request's `ControllerContext` instance (which can be accessed through the `Controller.ControllerContext` property). I have described some of the most commonly used context objects and properties in Table 17-4.

Table 17-4. *Commonly Used Context Objects and Properties*

Property	Type	Description
`Request.QueryString`	`NameValueCollection`	`GET` variables sent with this request
`Request.Form`	`NameValueCollection`	`POST` variables sent with this request
`Request.Cookies`	`HttpCookieCollection`	Cookies sent by the browser with this request
`Request.HttpMethod`	`string`	The HTTP method (verb, such as `GET` or `POST`) used for this request
`Request.Headers`	`NameValueCollection`	The full set of HTTP headers sent with this request
`Request.Url`	`Uri`	The URL requested
`Request.UserHostAddress`	`string`	The IP address of the user making this request
`RouteData.Route`	`RouteBase`	The chosen `RouteTable.Routes` entry for this request
`RouteData.Values`	`RouteValueDictionary`	Active route parameters (either extracted from the URL or default values)
`HttpContext.Application`	`HttpApplicationStateBase`	Application state store
`HttpContext.Cache`	`Cache`	Application cache store
`HttpContext.Items`	`IDictionary`	State store for the current request
`HttpContext.Session`	`HttpSessionStateBase`	State store for the visitor's session
`User`	`IPrincipal`	Authentication information about the logged-in user
`TempData`	`TempDataDictionary`	Temporary data items stored for the current user

The individual properties that I refer to here—`Request`, `HttpContext`, and so on—provide *context objects*. I am not going to go into them in detail in this book (because they are part of the ASP.NET platform), but they provide access to some useful information and features and are worth exploring. An action method can use any of these context objects to get information about the request, as Listing 17-5 demonstrates in the form of a hypothetical action method.

Listing 17-5. An Action Method Using Context Objects to Get Information About a Request

```
...
public ActionResult RenameProduct() {
    // Access various properties from context objects
    string userName = User.Identity.Name;
    string serverName = Server.MachineName;
    string clientIP = Request.UserHostAddress;
    DateTime dateStamp = HttpContext.Timestamp;
    AuditRequest(userName, serverName, clientIP, dateStamp, "Renaming product");

    // Retrieve posted data from Request.Form
    string oldProductName = Request.Form["OldName"];
    string newProductName = Request.Form["NewName"];
    bool result = AttemptProductRename(oldProductName, newProductName);

    ViewData["RenameResult"] = result;
    return View("ProductRenamed");
}
...
```

You can explore the vast range of available request context information using IntelliSense (in an action method, type `this.` and browse the pop-up), and the Microsoft Developer Network (look up `System.Web.Mvc.Controller` and its base classes, or `System.Web.Mvc.ControllerContext`).

Using Action Method Parameters

As you've seen in previous chapters, action methods can take parameters. This is a neater way to receive incoming data than extracting it manually from context objects, and it makes your action methods easier to read. For example, suppose I have an action method that uses context objects like this:

```
...
public ActionResult ShowWeatherForecast() {
    string city = (string)RouteData.Values["city"];
    DateTime forDate = DateTime.Parse(Request.Form["forDate"]);
    // ... implement weather forecast here ...
    return View();
}
...
```

I can rewrite it to use parameters, like this:

```
...
public ActionResult ShowWeatherForecast(string city, DateTime forDate) {
    // ... implement weather forecast here ...
    return View();
}
...
```

Not only is this easier to read, but it also helps with unit testing. I can test the action method without needing to mock the convenience properties of the controller class.

■ **Tip** It is worth noting that action methods aren't allowed to have `out` or `ref` parameters. It wouldn't make any sense if they did and the MVC Framework will simply throw an exception if it sees such a parameter.

The MVC Framework will provide values for action method parameters by checking the context objects and properties automatically, including `Request.QueryString`, `Request.Form`, and `RouteData.Values`. The names of the parameters are treated case-insensitively, so that an action method parameter called `city` can be populated by a value from `Request.Form["City"]`, for example.

Understanding How Parameters Objects Are Instantiated

The base `Controller` class obtains values for action method parameters using MVC Framework components called *value providers* and *model binders*. Value providers represent the set of data items available to your controller. There are built-in value providers that fetch items from `Request.Form`, `Request.QueryString`, `Request.Files`, and `RouteData.Values`. The values are then passed to model binders that try to map them to the types that your action methods require as parameters.

The default model binders can create and populate objects of any .NET type, including collections and project-specific custom types. You saw an example of this in Chapter 11 when form posts from administrators were presented to an action method as a single `Product` object, even though the individual values were dispersed among the elements of the HTML form. I cover value providers and model binders in depth in Chapter 24.

Understanding Optional and Compulsory Parameters

If the MVC Framework cannot find a value for a reference type parameter (such as a `string` or `object`), the action method will still be called, but using a `null` value for that parameter. If a value cannot be found for a value type parameter (such as `int` or `double`), then an exception will be thrown, and the action method will *not* be called. Here is another way to think about it:

- Value-type parameters are compulsory. To make them optional, either specify a default value (see the next section) or change the parameter type to a nullable type (such as `int?` or `DateTime?`), so the MVC Framework can pass `null` if no value is available.
- Reference-type parameters are optional. To make them compulsory (to ensure that a non-`null` value is passed), add some code to the top of the action method to reject `null` values. For example, if the value equals `null`, throw an `ArgumentNullException`.

Specifying Default Parameter Values

If you want to process requests that do not contain values for action method parameters, but you would rather not check for `null` values in your code or have exceptions thrown, you can use the C# optional parameter feature instead. Listing 17-6 provides a demonstration.

Listing 17-6. Using the C# Optional Parameter Feature in an Action Method

```
...
public ActionResult Search(string query= "all", int page = 1) {
    // ...process request...
    return View();
}
...
```

You mark parameters as optional by assigning values when you define them. In the listing, I have provided default values for the query and page parameters. The MVC Framework will try to obtain values from the request for these parameters, but if there are no values available, the defaults I have specified will be used instead.

For the string parameter, query, this means that I do not need to check for null values. If the request I am processing didn't specify a query, then the action method will be called with the string all. For the int parameter, I do not need to worry about requests resulting in errors when there is no page value: the method will be called with the default value of 1. Optional parameters can be used for *literal types*, which are types that you can define without using the new keyword, including string, int, and double.

■ **Caution** If a request *does* contain a value for a parameter but it cannot be converted to the correct type (for example, if the user gives a nonnumeric string for an int parameter), then the framework will pass the default value for that parameter type (for example, 0 for an int parameter), and will register the attempted value as a validation error in a special context object called ModelState. Unless you check for validation errors in ModelState, you can get into odd situations where the user has entered bad data into a form, but the request is processed as though the user had not entered any data or had entered the default value. See Chapter 25 for details of validation and ModelState, which can be used to avoid such problems.

Producing Output

After a controller has finished processing a request, it usually needs to generate a response. When I created the bare-metal controller by implementing the IController interface directly, I needed to take responsibility for every aspect of processing a request, including generating the response to the client. If I want to send an HTML response, for example, then I must create and assemble the HTML data and send it to the client using the Response.Write method. Similarly, if I want to redirect the user's browser to another URL, I need to call the Response.Redirect method and pass the URL I am interested in directly. Both of these approaches are shown in Listing 17-7, which shows enhancements to the BasicController class.

Listing 17-7. Generating Results in the BasicController.cs File

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

namespace ControllersAndActions.Controllers {

    public class BasicController : IController {

        public void Execute(RequestContext requestContext) {

            string controller = (string)requestContext.RouteData.Values["controller"];
            string action = (string)requestContext.RouteData.Values["action"];

            if (action.ToLower() == "redirect") {
                requestContext.HttpContext.Response.Redirect("/Derived/Index");
```

```
            } else {
                requestContext.HttpContext.Response.Write(
                    string.Format("Controller: {0}, Action: {1}",
                    controller, action));
            }
        }
    }
}
```

You can use the same approach when you have derived your controller from the `Controller` class. The `HttpResponseBase` class that is returned when you read the `requestContext.HttpContext.Response` property in your `Execute` method is available through the `Controller.Response` property, as shown in Listing 17-8, which shows enhancements to the `DerivedController` class.

Listing 17-8. Using the Response Property to Generate Output in the DerivedController.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class DerivedController : Controller {

        public ActionResult Index() {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }

        public void ProduceOutput() {
            if (Server.MachineName == "TINY") {
                Response.Redirect("/Basic/Index");
            } else {
                Response.Write("Controller: Derived, Action: ProduceOutput");
            }
        }
    }
}
```

The `ProduceOutput` method uses the value of the `Server.MachineName` property to decide what response to send to the client. (TINY is the name of one of my development machines.) This approach works, but it has a few problems:

- The controller classes must contain details of HTML or URL structure, which makes the classes harder to read and maintain.
- It is hard to unit test a controller that generates its response directly to the output. You need to create mock implementations of the `Response` object, and then be able to process the output you receive from the controller in order to determine what the output represents. This can mean parsing HTML for keywords, for example, which is a drawn-out and painful process.
- Handling the fine detail of every response this way is tedious and error-prone. Some programmers will like the absolute control that building a raw controller gives, but normal people get frustrated pretty quickly.

Fortunately, the MVC Framework has a nice feature that addresses all of these issues, called *action results*. The following sections introduce the action result concept and show you the different ways that it can be used to generate responses from controllers.

Understanding Action Results

The MVC Framework uses action results to separate *stating intentions* from *executing intentions*. The concept is simple once you have mastered it, but it takes a while to get your head around the approach at first because there is a little bit of indirection going on.

Instead of working directly with the `Response` object, action methods return an object derived from the `ActionResult` class that describes what the response from controller will be, such as rendering a view or redirecting to another URL or action method. But—and this is where the indirection comes in—you don't generate the response directly. Instead, you create an `ActionResult` object that the MVC Framework processes to produce the result for you, after the action method has been invoked.

■ **Note** The system of action results is an example of the *command pattern*. This pattern describes scenarios where you store and pass around objects that describe operations to be performed. See http://en.wikipedia.org/wiki/Command_pattern for more details.

When the MVC Framework receives an `ActionResult` object from an action method, it calls the `ExecuteResult` method defined by that object. The action result implementation then deals with the `Response` object for you, generating the output that corresponds to your intention. To demonstrate how this works, I created an `Infrastructure` folder and added a new class file called `CustomRedirectResult.cs` to it, which I then used to define the custom `ActionResult` implementation shown in Listing 17-9.

Listing 17-9. The Contents of the CustomRedirectResult.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Infrastructure {
    public class CustomRedirectResult : ActionResult {

        public string Url { get; set; }

        public override void ExecuteResult(ControllerContext context) {
            string fullUrl = UrlHelper.GenerateContentUrl(Url, context.HttpContext);
            context.HttpContext.Response.Redirect(fullUrl);
        }
    }
}
```

I based this class on the way that the `System.Web.Mvc.RedirectResult` class works. One of the benefits of the MVC Framework being open source is that you can see how things work behind the scenes. The `CustomRedirectResult` class is a lot simpler than the MVC equivalent, but is enough for my purposes in this chapter.

When I create an instance of the `RedirectResult` class, I pass in the URL I want to redirect the user to. The `ExecuteResult` method, which will be executed by the MVC Framework when the action method has finished, gets the `Response` object for the query through the `ControllerContext` object that the framework provides, and calls the `Redirect` method, which is exactly what I was doing in the bare-bones `IController` implementation in Listing 17-7. You can see how I have used the `CustomRedirectResult` class in the `Derived` controller in Listing 17-10.

Listing 17-10. Using the CustomRedirectResult Class in the DerivedController.cs File

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

namespace ControllersAndActions.Controllers {
    public class DerivedController : Controller {
        public ActionResult Index() {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }

        public ActionResult ProduceOutput() {
            if (Server.MachineName == "TINY") {
                return new CustomRedirectResult { Url = "/Basic/Index" };
            } else {
                Response.Write("Controller: Derived, Action: ProduceOutput");
                return null;
            }
        }
    }
}
```

Notice that I have had to change the result of the action method to return an `ActionResult`. I return `null` if I do not want the MVC Framework to do anything after the action method has been executed, which is what I have done when I do not return a `CustomRedirectResult` instance.

UNIT TESTING CONTROLLERS AND ACTIONS

Many parts of the MVC Framework are designed to facilitate unit testing, and this is especially true for actions and controllers. There are a few reasons for this support:

- You can test actions and controllers outside a web server. The context objects are accessed through their base classes (such as `HttpRequestBase`), which are easy to mock.
- You do not need to parse any HTML to test the result of an action method. You can inspect the `ActionResult` object that is returned to ensure that you received the expected result.
- You do not need to simulate client requests. The MVC Framework model binding system allows you to write action methods that receive input as method parameters. To test an action method, you simply call the action method directly and provide the parameter values that interest you.

I will show you how to create unit tests for the different kinds of action results throughout this chapter.

Do not forget that unit testing isn't the complete story. Complex behaviors in an application arise when action methods are called in sequence. Unit testing is best combined with other testing approaches.

Now that you have seen how a custom redirection action result works, I can switch to the equivalent one provided by the MVC Framework, which has more features and has been thoroughly tested by Microsoft. Listing 17-11 shows the change to the `Derived` controller.

Listing 17-11. Using the Built-in RedirectResult Object in the DerivedController.cs File

```
...
public ActionResult ProduceOutput() {
    return new RedirectResult("/Basic/Index");
}
...
```

I have removed the conditional statement from the action method, which means that if you start the application and navigate to the `/Derived/ProduceOutput` method, your browser will be redirected to the `/Basic/Index` URL.

To make action method code simpler, the `Controller` class includes convenience methods for generating different kinds of `ActionResult` objects. So, as an example, I can achieve the effect in Listing 17-11 by returning the result of the `Redirect` method, as shown in Listing 17-12.

Listing 17-12. Using a Controller Convenience Method in the DerivedController.cs File

```
...
public ActionResult ProduceOutput() {
    return Redirect("/Basic/Index");
}
...
```

There is nothing in the action result system that is especially complex, but it helps you create with simpler, cleaner and more consistent code, which is easier to read and easier to unit test.

In the case of a redirection, for example, you can simply check that the action method returns an instance of `RedirectResult` and that the `Url` property contains the target you expect.

The MVC Framework contains a number of built-in action result types, which are shown in Table 17-5. All of these types are derived from `ActionResult`, and many of them have convenient helper methods in the `Controller` class. In the following sections, I will show you how to use the most important of these result types.

Table 17-5. *Built-in ActionResult Types*

Type	Description	Helper Methods
`ViewResult`	Renders the specified or default view template	`View`
`PartialViewResult`	Renders the specified or default partial view template	`PartialView`

`RedirectToRouteResult`	Issues an HTTP 301 or 302 redirection to an action method or specific route entry, generating a URL according to your routing configuration	`RedirectToAction`
		`RedirectToActionPermanent`
		`RedirectToRoute`
		`RedirectToRoutePermanent`

`RedirectResult`	Issues an HTTP 301 or 302 redirection to a specific URL	`Redirect` `RedirectPermanent`
`ContentResult`	Returns raw textual data to the browser, optionally setting a content-type header	`Content`

(*continued*)

Table 17-5. (*continued*)

Type	Description	Helper Methods
`FileResult`	Transmits binary data (such as a file from disk or a byte array in memory) directly to the browser	`File`
`JsonResult`	Serializes a .NET object in JSON format and sends it as the response. This kind of response is more typically generated using the Web API feature, which I describe in Chapter 27, but you can see this action type used in Chapter 23.	`Json`
`JavaScriptResult`	Sends a snippet of JavaScript source code that should be executed by the browser	`JavaScript`
`HttpUnauthorizedResult`	Sets the response HTTP status code to 401 (meaning "not authorized"), which causes the active authentication mechanism (forms authentication or Windows authentication) to ask the visitor to log in	None
`HttpNotFoundResult`	Returns a `HTTP 404–Not found` error	`HttpNotFound`
`HttpStatusCodeResult`	Returns a specified HTTP code	None
`EmptyResult`	Does nothing	None

Returning HTML by Rendering a View

The most common kind of response from an action method is to generate HTML and send it to the browser. To demonstrate how to render views, I added a controller called `Example` to the project. You can see the contents of the `ExampleController.cs` class file in Listing 17-13.

Listing 17-13. The Contents of the ExampleController.cs File

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

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

When using the action result system, you specify the view that you want the MVC Framework to render using an instance of the `ViewResult` class. The simplest way to do this is to call the controller's `View` method, passing the name of the view as an argument. In the listing, I called the `View` method with an argument of `Homepage`, which specifies that I want the `HomePage.cshtml` view to be used.

■ **Note** Notice that that the return type for the action method in the listing is `ViewResult`. The method would compile and work just as well if I had specified the more general `ActionResult` type. In fact, some MVC programmers will define the result of every action method as `ActionResult`, even when they know it will always return a more specific type.

When the MVC Framework calls the `ExecuteResult` method of the `ViewResult` object, a search will begin for the view that you have specified. If you are using areas in your project, then the framework will look in the following locations:

- `/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.aspx`
- `/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.ascx`
- `/Areas/<AreaName>/Views/Shared/<ViewName>.aspx`
- `/Areas/<AreaName>/Views/Shared/<ViewName>.ascx`
- `/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml`
- `/Areas/<AreaName>/Views/<ControllerName>/<ViewName>.vbhtml`
- `/Areas/<AreaName>/Views/Shared/<ViewName>.cshtml`
- `/Areas/<AreaName>/Views/Shared/<ViewName>.vbhtml`

You can see from the list that the framework looks for views that have been created for the legacy ASPX view engine (the `.aspx` and `.ascx` file extensions), even though the MVC Framework uses Razor. This is to preserve compatibility with early versions of the MVC Framework that used the rendering features from ASP.NET Web Forms.

The framework also looks for C# and Visual Basic .NET Razor templates. (The `.cshtml` files are the C# ones and `.vbhtml` files are Visual Basic. The Razor syntax is the same in these files, but the code fragments are, as the names suggest, in different languages.) The MVC Framework checks to see if each of these files exists in turn. As soon as it locates a match, it uses that view to render the result of the action method.

If you are not using areas, or you are using areas but none of the files in the preceding list have been found, then the framework continues its search, using the following locations:

- `/Views/<ControllerName>/<ViewName>.aspx`
- `/Views/<ControllerName>/<ViewName>.ascx`
- `/Views/Shared/<ViewName>.aspx`
- `/Views/Shared/<ViewName>.ascx`
- `/Views/<ControllerName>/<ViewName>.cshtml`
- `/Views/<ControllerName>/<ViewName>.vbhtml`
- `/Views/Shared/<ViewName>.cshtml`
- `/Views/Shared/<ViewName>.vbhtml`

Once again, as soon as the MVC Framework tests a location and finds a file, then the search stops, and the view that has been found is used to render the response to the client.

I am not using areas in the example application, so the first place that the framework will look will be `/Views/Example/Index.aspx`. Notice that the `Controller` part of the class name is omitted, so that creating a `ViewResult` in `ExampleController` leads to a search for a directory called `Example`.

UNIT TEST: RENDERING A VIEW

To test the view that an action method renders, you can inspect the `ViewResult` object that it returns. This is not quite the same thing (after all, you are not following the process through to check the final HTML that is generated) but it is close enough, as long as you have reasonable confidence that the MVC Framework view system works properly. I added a new unit test file called `ActionTests.cs` to the test project to hold the unit tests for this chapter.

The first situation I want to test is when an action method selects a specific view, like this:

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

You can determine which view has been selected by reading the `ViewName` property of the `ViewResult` object, as shown in this test method.

```
using System.Web.Mvc;
using ControllersAndActions.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ControllersAndActions.Tests {
    [TestClass]
    public class ActionTests {

        [TestMethod]
        public void ControllerTest() {

            // Arrange - create the controller
            ExampleController target = new ExampleController();


// Act - call the action method
            ViewResult result = target.Index();

            // Assert - check the result
            Assert.AreEqual("Homepage", result.ViewName);
        }
    }
}
```

A slight variation arises when you are testing an action method that selects the default view, like this:

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

In such situations, you need to accept the empty string ("") for the view name, like this:

```
...
Assert.AreEqual("", result.ViewName);
...
```

The empty string is how the `ViewResult` object signals to the Razor view engine that the default view associated with the action method has been selected.

The sequence of directories that the MVC Framework searches for a view is another example of convention over configuration. You do not need to register your view files with the framework. You just put them in one of a set of known locations, and the framework will find them. I can take the convention a step further by omitting the name of the view I want rendered when calling the `View` method, as shown in Listing 17-14.

Listing 17-14. Creating a ViewResult Without Specifying a View in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

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

The MVC Framework assumes that I want to render a view that has the same name as the action method. This means that the call to the `View` method in Listing 17-14 starts a search for a view called `Index`.

■ **Note** The MVC Framework actually gets the name of the action method from the `RouteData.Values["action"]` value, which I explained as part of the routing system in Chapters 15 and 16. The action method name and the routing value will be the same if you are using the built-in routing classes, but this may not be the case if you have implemented custom routing classes which do not follow the MVC Framework conventions.

There are a number of overridden versions of the `View` method. They correspond to setting different properties on the `ViewResult` object that is created. For example, you can override the layout used by a view by explicitly naming an alternative, like this:

```
...
public ViewResult Index() {
    return View("Index", "_AlternateLayoutPage");
}
...
```

SPECIFYING A VIEW BY ITS PATH

The naming convention approach is convenient and simple, but it does limit the views you can render. If you want to render a specific view, you can do so by providing an explicit path and bypass the search phase. Here is an example:

```
using System.Web.Mvc;

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            return View("~/Views/Other/Index.cshtml");
        }
    }
}
```

When you specify a view like this, the path must begin with `/` or `~/` and include the file name extension (such as `.cshtml` for Razor views containing C# code).

If you find yourself using this feature, I suggest that you take a moment and ask yourself what you are trying to achieve. If you are attempting to render a view that belongs to another controller, then you might be better off redirecting the user to an action method in that controller (see the "Redirecting to an Action Method" section later in this chapter for an example). If you are trying to work around the naming scheme because it doesn't suit the way you have organized your project, then see Chapter 20, which explains how to implement a custom search sequence.

Passing Data from an Action Method to a View

The MVC Framework provides a number of different ways to pass data from an action method to a view, which I describe in the following sections. I touch on the topic of views, which I cover in depth in Chapter 20. In this chapter, I discuss only enough view functionality to demonstrate the controller features of interest.

Providing a View Model Object

You can send an object to the view by passing it as a parameter to the `View` method as shown in Listing 17-15.

Listing 17-15. Specifying a View Model Object in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {
```

```
        public ViewResult Index() {
            DateTime date = DateTime.Now;
            return View(date);
        }
    }
}
```

I passed a `DateTime` object as the view model and I can access the object in the view using the Razor `Model` keyword. To demonstrate the `Model` keyword, I added a view called `Index.cshtml` in the `Views/Example` folder, with the contents shown in Listing 17-16.

Listing 17-16. Accessing a View Model in the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @(((DateTime)Model).DayOfWeek)
```

This is an *untyped* or *weakly typed* view. The view does not know anything about the view model object, and treats it as an instance of `object`. To get the value of the `DayOfWeek` property, I need to cast the object to an instance of `DateTime`. This works, but produces messy views. I can tidy this up by creating *strongly typed views*, in which the view includes details of the type of the view model object, as demonstrated in Listing 17-17.

Listing 17-17. Adding Strong Typing to the Index.cshtml File

```
@model DateTime
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @Model.DayOfWeek
```

I specified the view model type using the Razor `model` keyword. Notice that I use a lowercase `m` when specifying the model type and an uppercase `M` when reading the value. Not only does this help tidy up the view, but Visual Studio supports IntelliSense for strongly typed views, as shown in Figure 17-3.

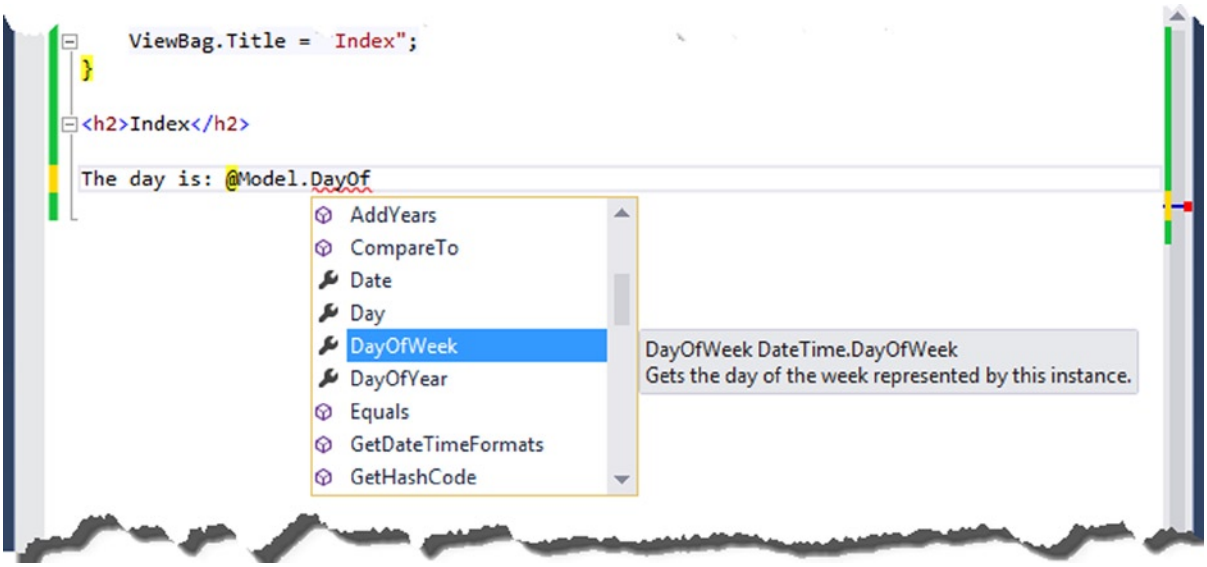

Figure 17-3. *IntelliSense support for strongly typed views*

UNIT TEST: VIEW MODEL OBJECTS

You can access the view model object passed from the action method to the view through the `ViewResult.ViewData.Model` property. Here is the test for the action method in Listing 17-17. You can see that I have used the `Assert.IsInstanceOfType` method to check that the view model object is an instance of `DateTime`:

```
...
[TestMethod]
public void ViewSelectionTest() {

    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    ViewResult result = target.Index();

    // Assert - check the result
    Assert.AreEqual("", result.ViewName);
    Assert.IsInstanceOfType(result.ViewData.Model, typeof(System.DateTime));
}
...
```

I had to change the name of the view that I check for to reflect the changes in the action method since the last unit test I showed you, as follows:

```
...
[TestMethod]
public void ControllerTest() {

    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    ViewResult result = target.Index();

    // Assert - check the result
    Assert.AreEqual("", result.ViewName);
}
...
```

Passing Data with the View Bag

I introduced the View Bag feature in Chapter 2. This feature allows you to define properties on a dynamic object and access them in a view. The dynamic object is accessed through the `Controller.ViewBag` property, as demonstrated in Listing 17-18.

Listing 17-18. Using the View Bag Feature in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
    }
}
```

I have defined View Bag properties called `Message` and `Date` simply by assigning values to them. Before this point, no such properties existed, and I made no preparations to create them. To read the data back in the view, I simply get the same properties that I set in the action method, as Listing 17-19 shows.

Listing 17-19. Reading Data from the ViewBag in the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

The day is: @ViewBag.Date.DayOfWeek
<p />
The message is: @ViewBag.Message
```

The ViewBag has an advantage over using a view model object in that it is easy to send multiple objects to the view. If I were restricted to using view models, then I would need to create a new type that had string and DateTime members in order to get the same effect.

When working with dynamic objects, you can enter any sequence of method and property calls in the view, like this:

```
...
The day is: @ViewBag.Date.DayOfWeek.Blah.Blah.Blah
...
```

Visual Studio cannot provide IntelliSense support for any dynamic objects, including the ViewBag, and errors such as this won't be revealed until the view is rendered.

UNIT TEST: VIEWBAG

You can read values from the ViewBag through the ViewResult.ViewBag property. The following test method is for the action method in Listing 17-18:

```
...
[TestMethod]
public void ControllerTest() {

    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    ViewResult result = target.Index();

    // Assert - check the result
    Assert.AreEqual("Hello", result.ViewBag.Message);
}
...
```

Performing Redirections

A common result from an action method is not to produce any output directly, but to redirect the user's browser to another URL. Most of the time, this URL is another action method in the application that generates the output you want the users to see.

THE POST/REDIRECT/GET PATTERN

The most frequent use of a redirect is in action methods that process HTTP `POST` requests. As I mentioned in the previous chapter, `POST` requests are used when you want to change the state of an application. If you just return HTML following the processing of a request, you run the risk that the user will click the browser's reload button and resubmit the form a second time, causing unexpected and undesirable results.

To avoid this problem, you can follow the pattern called Post/Redirect/Get. In this pattern, you receive a `POST` request, process it, and then redirect the browser so that a `GET` request is made by the browser for another URL. `GET` requests should not modify the state of your application, so any inadvertent resubmissions of this request won't cause any problems.

When you perform a redirect, you send one of two HTTP codes to the browser:

- Send the HTTP code 302, which is a *temporary* redirection. This is the most frequently used type of redirection and when using the Post/Redirect/Get pattern, this is the code that you want to send.
- Send the HTTP code 301, which indicates a permanent redirection. This should be used with caution, because it instructs the recipient of the HTTP code not to request the original URL ever again and to use the new URL that is included alongside the redirection code. If you are in doubt, use temporary redirections; that is, send code 302.

Redirecting to a Literal URL

The most basic way to redirect a browser is to call the `Redirect` method, which returns an instance of the `RedirectResult` class, as shown in Listing 17-20.

Listing 17-20. Redirecting to a Literal URL in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
```

```
        public RedirectResult Redirect() {
            return Redirect("/Example/Index");
        }
    }
}
```

The URL you want to redirect to is expressed as a string and passed as a parameter to the `Redirect` method. The `Redirect` method sends a temporary redirection. You can send a permanent redirection using the `RedirectPermanent` method, as shown in Listing 17-21.

Listing 17-21. Permanently Redirecting to a Literal URL in the ExampleController.cs File

```
...
public RedirectResult Redirect() {
    return RedirectPermanent("/Example/Index");
}
...
```

■ **Tip** If you prefer, you can use the overloaded version of the `Redirect` method, which takes a `bool` parameter that specifies whether or not a redirection is permanent.

UNIT TEST: LITERAL REDIRECTIONS

Literal redirections are easy to test. You can read the URL and test whether the redirection is permanent or temporary using the `Url` and `Permanent` properties of the `RedirectResult` class. The following is a test method for the redirection shown in Listing 17-21:

```
...
[TestMethod]
public void ControllerTest() {
    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    RedirectResult result = target.Redirect();

    // Assert - check the result
    Assert.IsTrue(result.Permanent);
    Assert.AreEqual("/Example/Index", result.Url);
}
...
```

Notice that I have updated the test to receive a `RedirectResult` when I call the action method.

Redirecting to a Routing System URL

If you are redirecting the user to a different part of your application, you need to make sure that the URL you send is valid within your URL schema. The problem with using literal URLs for redirection is that any change in your routing schema means that you need to go through your code and update the URLs. Fortunately, you can use the routing system to generate valid URLs with the `RedirectToRoute` method, which creates an instance of the `RedirectToRouteResult`, as shown in Listing 17-22.

Listing 17-22. Redirecting to a Routing System URL in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {

            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;

            return View();
        }

        public RedirectToRouteResult Redirect() {
            return RedirectToRoute(new {
                controller = "Example",
                action = "Index",
                ID = "MyID"
            });
        }
    }
}
```

The `RedirectToRoute` method issues a temporary redirection. Use the `RedirectToRoutePermanent` method for permanent redirections. Both methods take an anonymous type whose properties are then passed to the routing system to generate a URL. For more details of this process, see the Chapters 15 and 16.

■ **Tip** Notice that the `RedirectToRoute` method returns a `RedirectToRouteResult` object and that I have updated the action method to return this type.

UNIT TESTING: ROUTED REDIRECTIONS

Here is the unit test for the action method in Listing 17-22:

```
...
[TestMethod]
public void ControllerTest() {

    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    RedirectToRouteResult result = target.Redirect();

    // Assert - check the result
    Assert.IsFalse(result.Permanent);
    Assert.AreEqual("Example", result.RouteValues["controller"]);
    Assert.AreEqual("Index", result.RouteValues["action"]);
    Assert.AreEqual("MyID", result.RouteValues["ID"]);
}
...
```

You can see that I have tested the result indirectly by looking at the routing information provided by the `RedirectToRouteResult` object, which means that I don't have to parse a URL.

Redirecting to an Action Method

You can redirect to an action method more elegantly by using the `RedirectToAction` method (for temporary redirections) or the `RedirectToActionPermanent` (for permanent redirections). These are just wrappers around the `RedirectToRoute` method that lets you specify values for the action method and the controller without needing to create an anonymous type, as shown in Listing 17-23.

Listing 17-23. Redirecting Using the RedirectToAction Method in the ExampleController.cs File

```
...
public RedirectToRouteResult Redirect() {
    return RedirectToAction("Index");
}
...
```

If you just specify an action method, then it is assumed that you are referring to an action method in the current controller. If you want to redirect to another controller, you need to provide the name as a parameter, like this:

```
...
public RedirectToRouteResult Redirect() {
    return RedirectToAction("Index", "Basic");
}
...
```

There are other overloaded versions that you can use to provide additional values for the URL generation. These are expressed using an anonymous type, which does tend to undermine the purpose of the convenience method, but can still make your code easier to read.

■ **Note** The values that you provide for the action method and controller are not verified before they are passed to the routing system. You are responsible for making sure that the targets you specify actually exist.

PRESERVING DATA ACROSS A REDIRECTION

A redirection causes the browser to submit an entirely new HTTP request, which means that you do not have access to the details of the original request. If you want to pass data from one request to the next, you can use the Temp Data feature.

`TempData` is similar to `Session` data, except that `TempData` values are marked for deletion when they are read, and they are removed when the request has been processed. This is an ideal arrangement for short-lived data that you want to persist across a redirection. Here is a simple example in an action method that uses the `RedirectToAction` method:

```
...
public RedirectToRouteResult RedirectToRoute() {
    TempData["Message"] = "Hello";
    TempData["Date"] = DateTime.Now;
    return RedirectToAction("Index");
}
...
```

When this method processes a request, it sets values in the `TempData` collection, and then redirects the user's browser to the `Index` action method in the same controller. You can read the `TempData` values back in the target action method, and then pass them to the view, like this:

```
...
public ViewResult Index() {
    ViewBag.Message = TempData["Message"];
    ViewBag.Date = TempData["Date"];
    return View();
}
...
```

A more direct approach would be to read these values in the view, like this:

```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

```
The day is: @(((DateTime)TempData["Date"]).DayOfWeek)
<p />
The message is: @TempData["Message"]
```

Reading the values in the view means that you do not need to use the View Bag feature in the action method. However, you must cast the `TempData` results to an appropriate type.

You can get a value from `TempData` without marking it for removal by using the `Peek` method, like this:

```
...
DateTime time = (DateTime)TempData.Peek("Date");
...
```

You can preserve a value that would otherwise be deleted by using the `Keep` method, like this:

```
...
TempData.Keep("Date");
...
```

The `Keep` method doesn't protect a value forever. If the value is read again, it will be marked for removal once more. If you want to store items so that they won't be removed when the request is processed then use session data instead.

Returning Errors and HTTP Codes

The last of the built-in `ActionResult` classes that I will look at can be used to send specific error messages and HTTP result codes to the client. Most applications do not require these features because the MVC Framework will automatically generate these kinds of results. However, they can be useful if you need to take more direct control over the responses sent to the client.

Sending a Specific HTTP Result Code

You can send a specific HTTP status code to the browser using the `HttpStatusCodeResult` class. There is no controller helper method for this, so you must instantiate the class directly, as shown in Listing 17-24.

Listing 17-24. Sending a Specific Status Code in the ExampleController.cs File

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

namespace ControllersAndActions.Controllers {

    public class ExampleController : Controller {

        public ViewResult Index() {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
```

```
        public RedirectToRouteResult Redirect() {
            return RedirectToAction("Index");
        }

        public HttpStatusCodeResult StatusCode() {
            return new HttpStatusCodeResult(404, "URL cannot be serviced");
        }
    }
}
```

The constructor parameters for `HttpStatusCodeResult` are the numeric status code and an optional descriptive message. In the listing, I returned code 404, which signifies that the requested resource does not exist.

Sending a 404 Result

I can achieve the same effect as Listing 17-24 using the more convenient `HttpNotFoundResult` class, which is derived from `HttpStatusCodeResult` and can be created using the controller `HttpNotFound` convenience method, as shown in Listing 17-25.

Listing 17-25. Generating a 404 Result in the ExampleController.cs File

```
...
public HttpStatusCodeResult StatusCode() {
    return HttpNotFound();
}
...
```

Sending a 401 Result

Another wrapper class for a specific HTTP status code is the `HttpUnauthorizedResult`, which returns the 401 code, used to indicate that a request is unauthorized. Listing 17-26 provides a demonstration.

Listing 17-26. Generating a 401 Result in the ExampleController.cs File

```
...
public HttpStatusCodeResult StatusCode() {
    return new HttpUnauthorizedResult();
}
...
```

There is no helper method in the `Controller` class to create instances of `HttpUnauthorizedResult`, so you must do so directly. The effect of returning an instance of this class is usually to redirect the user to the authentication page, as you saw in Chapter 12.

UNIT TEST: HTTP STATUS CODES

The `HttpStatusCodeResult` class follows the pattern you have seen for the other result types, and makes its state available through a set of properties. In this case, the `StatusCode` property returns the numeric HTTP status code, and the `StatusDescription` property returns the associated descriptive string. The following test method is for the action method in Listing 17-26:

```
...
[TestMethod]
public void ControllerTest() {

    // Arrange - create the controller
    ExampleController target = new ExampleController();

    // Act - call the action method
    HttpStatusCodeResult result = target.StatusCode();

    // Assert - check the result
    Assert.AreEqual(401, result.StatusCode);
}
...
```

Summary

Controllers are one of the key building blocks in the MVC design pattern. In this chapter, you have seen how to create "raw" controllers by implementing the `IController` interface and more convenient controllers by deriving from the `Controller` class. You saw the role that action methods play in MVC Framework controllers and how they ease unit testing. I showed you the different ways that you can receive input and generate output from an action method and demonstrated the different kinds of `ActionResult` that make this a simple and flexible process. In the next chapter, I go deeper into the controller infrastructure to show you the *filters* feature, which changes how requests are processed.