CHAPTER 21

Helper Methods

In this chapter, I look at the *helper methods*, which allow you to package up chunks of code and markup so that they can be reused throughout an MVC Framework application. I start by showing you how to create your own helper methods. The MVC Framework comes with a wide range of built-in helper methods, and I explore them in this chapter and the next two chapters, starting with the helper methods that you can use to create HTML `form`, `input` and `select` elements. Table 21-1 provides the summary for this chapter.

Table 21-1. *Chapter Summary*

Problem	Solution	Listing
Create a region of reusable markup within a view	Create an inline helper	1–4
Create markup that can be used within multiple views	Create an external helper	5–11
Generate a `form` element	Use the `Html.BeginForm` and `Html.EndForm` helpers	12–19
Generate a `form` element using a specific route	Use the `Html.BeginRouteForm` helper	20–21
Generate `input` elements	Use the input helpers	22–24
Generate `input` elements from model objects	Use the strongly typed input helpers	25
Generate `select` elements	Use the `DropDownList` and `ListBox` helpers and their strongly typed counterparts	26–27

Preparing the Example Project

For this chapter, I created a new Visual Studio MVC project called `HelperMethods` using the `Empty` template, checking the option to add the core `MVC` folders and references. I added a `Home` controller, which you can see in Listing 21-1

Listing 21-1. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pear"};
            ViewBag.Cities = new string[] { "New York", "London", "Paris" };
```

```
            string message = "This is an HTML element: <input>";

            return View((object)message);
        }
    }
}
```

In the Index action method, I pass a pair of string arrays to the view via the view bag and set the model object to be a string. I added a view called Index.cshtml to the Views/Home folder. You can see the contents of the view file in Listing 21-2. This is a strongly typed view (where the model type is string) and I have not used a layout.

Listing 21-2. The Contents of the Index.cshtml File

```
@model string

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits:
        @foreach (string str in (string[])ViewBag.Fruits) {
            <b>@str </b>
        }
    </div>
    <div>
        Here are the cities:
        @foreach (string str in (string[])ViewBag.Cities) {
            <b>@str </b>
        }
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

Setting the Start URL

I want Visual Studio to start with the root URL for the application rather than guess the URL based on the file that is being edited. Select HelperMethods 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 checking the option is enough.

Testing the Example Application

You can see how the view is rendered by starting the application. The default routing configuration added to the project by Visual Studio will map the root URL requested automatically by the browser to the `Index` action on the `Home` controller, as shown in Figure 21-1.

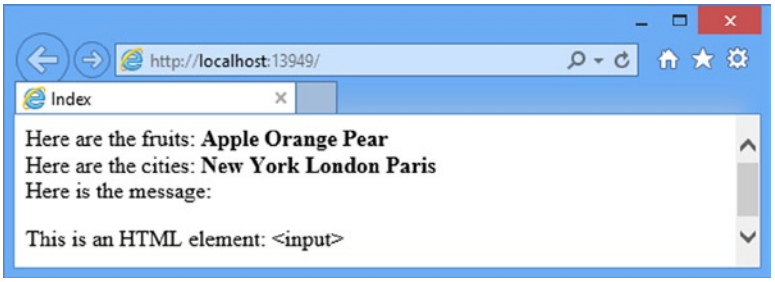

Figure 21-1. *Running the example application*

Creating Custom Helper Methods

I am going to follow the pattern I have established over the last few chapters and introduce you to helper methods by creating my own custom implementation. In the sections that follow, I will show you two different techniques for creating custom helper methods.

Creating an Inline Helper Method

The simplest kind of helper method is an *inline helper*, which is defined within a view. I can create an inline helper to simplify the example view using the `@helper` tag, as shown in Listing 21-3.

Listing 21-3. Creating an Inline Helper Method in the Index.cshtml File

```
@model string

@{
    Layout = null;
}

@helper ListArrayItems(string[] items) {
    foreach(string str in items) {
        <b>@str </b>
    }
}

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

```
<body>
    <div>
        Here are the fruits: @ListArrayItems(ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @ListArrayItems(ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

Inline helpers have names and parameters similar to regular C# methods. In the example, I defined a helper called `ListArrayItems`, which takes a string array as a parameter. Although an inline helper looks like a method, there is no return value. The contents of the helper body are processed and put into the response to the client.

■ **Tip** Notice that I did not have to cast the dynamic properties from the `ViewBag` to string arrays when using the inline helper. One of the nice features of this kind of helper method is that it is happy to evaluate types at runtime.

The body of an inline helper follows the same syntax as the rest of a Razor view. Literal strings are regarded as static HTML, and statements that require processing by Razor are prefixed with the `@` character. The helper in the example mixes static HTML and Razor tags to enumerate the items in the array, which produces the same output as the original view but has reduced the amount of duplication in the view.

The benefit of this approach is that I only have to make one change if I want to change the way that the array contents are displayed. As a simple example, in Listing 21-4 you can see how I have switched from just writing out the values to using the HTML unnumbered list elements.

Listing 21-4. Changing the Contents of a Helper Method in the Index.cshtml File

```
...
@helper ListArrayItems(string[] items) {
    <ul>
        @foreach(string str in items) {
            <li>@str</li>
        }
    </ul>
}
...
```

I only had to make the change in one place, which may seem like a trivial advantage in such a simple project, but this can be a useful way to keep your views simple and consistent in a real project. You can see the result of this change in Figure 21-2.

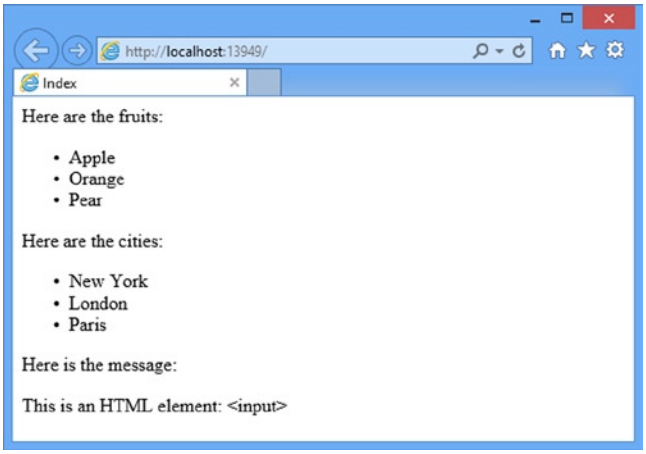

Figure 21-2. *Changing the markup in a helper method*

■ **Tip** Notice that I had to prefix the `foreach` keyword with `@` in this example but not in Listing 21-4. This is because the first element in the helper body changed to become an HTML element, which means I have to use `@` to tell Razor that I am using a C# statement. In the previous example there was no HTML element, so Razor assumed the contents were code. It can be hard to keep track of these quirks, but the Visual Studio will flag up errors like this for you.

Creating an External Helper Method

Inline helpers are convenient, but they can be used only from the view in which they are declared and, if they are too complex, they can take over that view and make it hard to read.

The alternative is to create an external HTML helper method, which is expressed as a C# extension method. External helper methods can be used more widely, but are a little more awkward to write, because C# doesn't naturally handle HTML element generation elegantly. To demonstrate this feature, I added an `Infrastructure` folder to the example project and created a new `CustomHelpers.cs` class file within it. You can see the contents of this file in Listing 21-5.

Listing 21-5. The Contents of the CustomHelpers.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {

        public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list) {

            TagBuilder tag = new TagBuilder("ul");
```

```
            foreach(string str in list) {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }

            return new MvcHtmlString(tag.ToString());
        }
    }
}
```

The helper method I created performs the same function as the inline helper in the previous example. It takes an array of strings and generates an HTML `ul` element, containing a `li` element for each `string` in the array.

The first parameter to the helper method is an `HtmlHelper` object, prefixed with the `this` keyword to tell the C# compiler that I am defining an extension method. The `HtmlHelper` provides access to information that can be useful when creating content, through the properties described in Table 21-2.

Table 21-2. *Useful Properties Defined by the HtmlHelper Class*

Property	Description
`RouteCollection`	Returns the set of routes defined by the application
`ViewBag`	Returns the view bag data passed from the action method to the view that has called the helper method
`ViewContext`	Returns a `ViewContext` object, which provides access to details of the request and how it has been handled (and which I describe below)

The `ViewContext` property is the most useful when you want to create content which adapts to the request being processed. In Table 21-3, I have described some of the most commonly used properties defined by the `ViewContext` class.

Table 21-3. *Useful Properties Defined by the ViewContext Class*

Property	Description
`Controller`	Returns the controller processing the current request
`HttpContext`	Returns the `HttpContext` object that describes the current request
`IsChildAction`	Returns `true` if the view that has called the helper is being rendered by a child action (see Chapter 20 for details of child actions)
`RouteData`	Returns the routing data for the request
`View`	Returns the instance of the `IView` implementation that has called the helper method

The information you can get about the request is fairly comprehensive, but for the most part helper methods are simple and used to keep formatting consistent. You can use the built-in helper methods for generating requests-specific content (I describe these helpers later in the chapter) and you can use partial views or child actions for more complex tasks (I provide guidance about which approach to use in next section of this chapter).

I do not need any information about the request in the example helper, but I do need to construct some HTML elements. The easiest way to create HTML in a helper method is to use the `TagBuilder` class, which allows you to build up HTML strings without needing to deal with all of the escaping and special characters. The `TagBuilder` class is part of the `System.Web.WebPages.Mvc` assembly but uses a feature called *type forwarding* to appear as though it is part of the `System.Web.Mvc` assembly. Both assemblies are added to MVC projects by Visual Studio, so you can use the `TagBuilder` class easily enough, but it does not appear in the Microsoft Developer Network (MSDN) API documentation.

I create a new `TagBuilder` instance, passing in name the HTML element I want to construct as the constructor parameter. I do not need to use the angle brackets (< and >) with the `TagBuilder` class, which means I can create a `ul` element, like this:

```
...
TagBuilder tag = new TagBuilder("ul");
...
```

The most useful members of the `TagBuilder` class are described in Table 21-4.

Table 21-4. *Some Members of the TagBuilder Class*

Member	Description
`InnerHtml`	A property that lets you set the contents of the element as an HTML string. The value assigned to this property will not be encoded, which means that is can be used to nest HTML elements.
`SetInnerText(string)`	Sets the text contents of the HTML element. The `string` parameter is encoded to make it safe to display.
`AddCssClass(string)`	Adds a CSS class to the HTML element
`MergeAttribute(string, string, bool)`	Adds an attribute to the HTML element. The first parameter is the name of the attribute, and the second is the value. The `bool` parameter specifies if an existing attribute of the same name should be replaced.

The result of an HTML helper method is an `MvcHtmlString` object, the contents of which are written directly into the response to the client. For the example helper, I pass the result of the `TagBuilder.ToString` method to the constructor of a new `MvcHtmlString` object, like this:

```
...
return new MvcHtmlString(tag.ToString());
...
```

This statement generates the HTML fragment that contains the `ul` and `li` elements and returns them to the view engine so that it can be inserted into the response.

Using a Custom External Helper Method

Using a custom external helper method is a little different to using an inline one. In Listing 21-6, you can see the changes I have made to the `/Views/Home/Index.cshtml` file to replace the inline helper with the external one.

Listing 21-6. Using a Custom External Helper Method in the Index.cshtml File

```
@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits: @Html.ListArrayItems((string[])ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @Html.ListArrayItems((string[])ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

I need to ensure that the namespace that contains the helper extension method is in scope. I have done this using an `@using` tag, but if you are developing a lot of custom helpers then you will want to add the namespaces that contain them to the `/Views/Web.config` file so that they are always available in your views.

I refer to the helper using `@Html.<helper>`, where `<helper>` is the name of the extension method. In this case, I use `@Html.ListArrayItems`. The `Html` part of this expression refers to a property defined by the view base class, which returns an `HtmlHelper` object, which is the type to which I applied the extension method in Listing 21-5.

I pass data to the helper method as I would for an inline helper or a C# method, although I must take care to cast from the dynamic properties of the `ViewBag` object to the type defined by the external helper (in this case a `string` array). This syntax is not as elegant as using inline helpers, but it is part of the price that you must pay to create a helper that can be used in any view in your project.

KNOWING WHEN TO USE HELPER METHODS

Now that you have seen how helper methods work, you might be wondering when you should use them in preference to partial views or child actions, especially as there is overlap between what these features are capable of.

I only use helper methods to reduce the amount of duplication in views, just as I did in this example, and only for the simplest of content. For more complex markup and content I use partial views and I use a child action when I need to perform any manipulation of model data. I recommend that you follow the same approach and keep your use of helper methods as simple as possible. (If my helpers contain more than a handful of C# statements—or more C# statements that HTML elements—then I tend to switch to a child action.)

Managing String Encoding in a Helper Method

The MVC Framework makes an effort to protect you from malicious data by automatically encoding it so that it can be added to an HTML page safely. You can see an example of this in the Home controller in the example application where I pass a potentially troublesome string to the view as the model object, as shown in Listing 21-7.

Listing 21-7. The Contents of the HomeController.cs File

```
using System.Web.Mvc;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pear" };
            ViewBag.Cities = new string[] { "New York", "London", "Paris" };

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }
    }
}
```

The model object contains a valid HTML element, but when the value is rendered by Razor, the following HTML is produced:

```
...
<div>
    Here is the message:
    <p>This is an HTML element: &lt;input&gt;</p>
</div>
...
```

This is a basic security precaution that prevents data values from being interpreted as valid markup by the browser. This is the foundation for a common form of attack in which malicious users will try to subvert the behavior of an application by trying to add their own HTML markup or JavaScript code. Razor encodes data values automatically when they are used in a view, but helper methods need to be able to generate HTML. As a consequence, they are given a higher level of trust by the view engine, and this can require some careful attention.

Demonstrating the Problem

To demonstrate the problem, I have created a new helper method in the CustomHelpers class, as shown in Listing 21-8. This helper takes a string as a parameter and generates the same HTML that I included in the Index view.

Listing 21-8. Defining a New Helper Method in the CustomHelpers.cs File

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

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {
```

```
        public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list) {

            TagBuilder tag = new TagBuilder("ul");

            foreach(string str in list) {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }

            return new MvcHtmlString(tag.ToString());
        }

        public static MvcHtmlString DisplayMessage(this HtmlHelper html, string msg) {
            string result = String.Format("This is the message: <p>{0}</p>", msg);
            return new MvcHtmlString(result);
        }
    }
}
```

I use the `String.Format` method to generate the HTML markup and pass the result as the argument to the `MvcHtmlString` constructor. In Listing 21-9, you can see how I have changed the `/View/Home/Index.cshtml` view to use the new helper method. (I also made some changes to emphasize the content that comes from the helper method.)

Listing 21-9. Using the DisplayMessage Helper Method in the Index.cshtml File

```
@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <p>This is the content from the view:</p>
    <div style="border: thin solid black; padding: 10px">
        Here is the message:
        <p>@Model</p>
    </div>

    <p>This is the content from the helper method:</p>
    <div style="border: thin solid black; padding: 10px">
        @Html.DisplayMessage(Model)
    </div>
</body>
</html>
```

You can see the effect the new helper method has by starting the application, as shown in Figure 21-3.

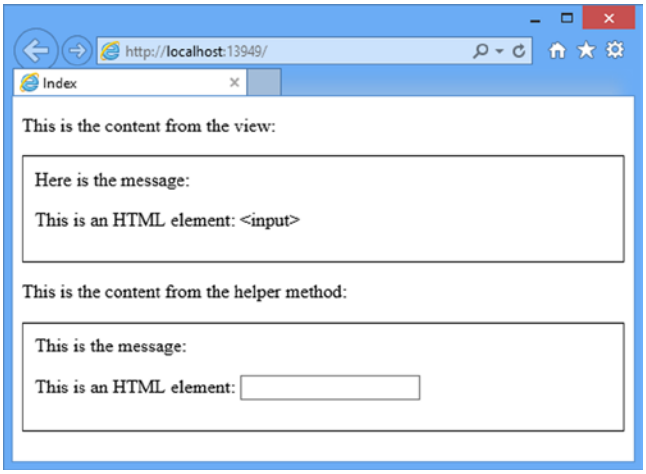

Figure 21-3. *Comparing how data values are encoded*

The helper method is trusted to generate safe content, which is unfortunate because it leads to the browser displaying an `input` element, which is the kind of behavior that can be exploited to subvert an application.

Encoding Helper Method Content

There are a couple of different ways to solve this problem and the choice between them depends on the nature of the content that your helper method produces. The simplest solution is to change the return type of the helper method to `string`, as shown in Listing 21-10. This alerts the view engine that your content is not safe and should be encoded before it is added to the view.

Listing 21-10. Ensuring that Razor Encodes Content in the CustomHelpers.cs File

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

namespace HelperMethods.Infrastructure {
    public static class CustomHelpers {

        public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list) {

            TagBuilder tag = new TagBuilder("ul");

            foreach(string str in list) {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }
```

```
            return new MvcHtmlString(tag.ToString());
        }

        public static string DisplayMessage(this HtmlHelper html, string msg) {
            return String.Format("This is the message: <p>{0}</p>", msg);
        }
    }
}
```

This technique causes Razor to encode all of the content that is returned by the helper, which is a problem when you are generating HTML elements (as I am in the example helper), but which is convenient otherwise. You can see the effect in Figure 21-4.

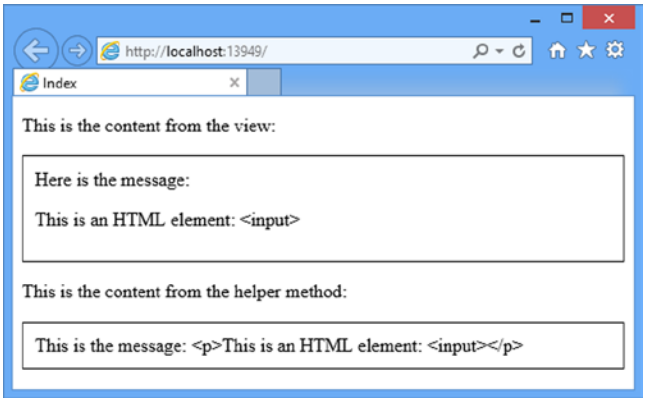

Figure 21-4. *Ensuring that the view engine encodes the response from a helper method*

I have solved the problem with the `input` element, but my `p` elements have been encoded as well, which is not what I need. In these situations, I need to be more selective and encode just the data values, as shown in Listing 21-11.

Listing 21-11. Selectively Encoding Data Values in the CustomHelpers.cs File

```
...
public static MvcHtmlString DisplayMessage(this HtmlHelper html, string msg) {
    string encodedMessage = html.Encode(msg);
    string result = String.Format("This is the message: <p>{0}</p>", encodedMessage);
    return new MvcHtmlString(result);
}
...
```

The `HtmlHelper` class defines an instance method called `Encode`, which solves the problem and encodes a string value so that it can be safely included in a view. The problem with this technique is that you have to remember to use it. I explicitly encode all of the data values at the start of the method as a reminder and I suggest that you adopt a similar approach.

You can see the result of this change in Figure 21-5, where you will see that the content generated by the external helper method matches that generated by using the model value directly in the view.

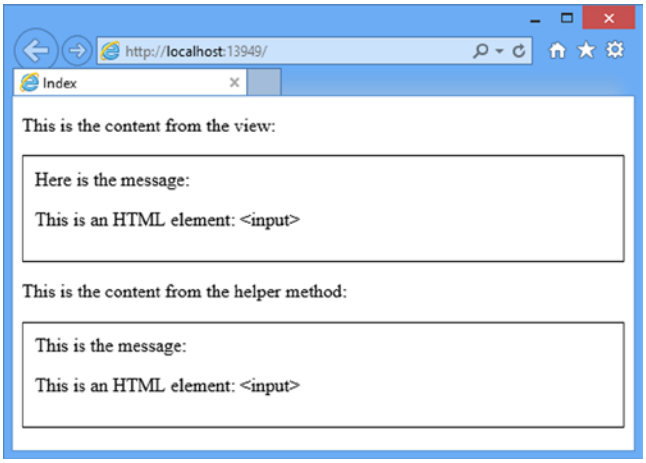

Figure 21-5. *The effect of selectively encoding content in an external helper method*

Using the Built-In Form Helper Methods

The MVC Framework includes a selection of built-in helper methods that help you manage the creating of HTML `form` elements. In the following sections, I will put these helpers in context and show you how they are used.

Creating Form Elements

One of the most common forms of interaction in a web application is the HTML form, which is the subject of a number of different helper methods. To demonstrate the form-related helpers, I made some additions to the example project. I started by creating a new class file called `Person.cs` in the `Models` folder. You can see the contents of this file in Listing 21-12. The `Person` type will be the view model class when I demonstrate the form-related helpers, and the `Address` and `Role` types will help me showcase some more advanced features.

Listing 21-12. The Contents of the Person.cs Model

```
using System;

namespace HelperMethods.Models {

    public class Person {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }
```

```
    public class Address {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role {
        Admin,
        User,
        Guest
    }
}
```

I also added new action methods to the Home controller to use the model objects, as shown in Listing 21-13.

Listing 21-13. Adding Action Methods in the HomeController.cs File

```
using System.Web.Mvc;
using HelperMethods.Models;

namespace HelperMethods.Controllers {
    public class HomeController : Controller {

        public ActionResult Index() {

            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pear" };
            ViewBag.Cities = new string[] { "New York", "London", "Paris" };

            string message = "This is an HTML element: <input>";

            return View((object)message);
        }

        public ActionResult CreatePerson() {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person) {
            return View(person);
        }
    }
}
```

This is the standard two-method approach to dealing with HTML forms, where I rely on model binding so that the MVC Framework will create a Person object from the form data and pass it to the action method with the HttpPost attribute. (I explained the HttpPost attribute in Chapter 19 and model binding is the topic of Chapter 24).

I am not processing the form data in any way because I am focused on how to generate elements in the view. The HttpPost action method just calls the View method and passes the Person object that it received as a parameter, which has the effect of redisplaying the form data to the user.

I am going to start with a standard manual HTML form and show you how to replace different parts of it using helper methods. You can see the initial version of the form in Listing 21-14, which shows the `CreatePerson.cshtml` view file that I added to the `/Views/Home` folder.

Listing 21-14. The Contents of the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}

<h2>CreatePerson</h2>

<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
</form>
```

This view contains a standard manually created form in which I have set the value of the `value` attribute of the `input` elements using the model object.

■ **Tip** Notice that I have set the `name` attribute on all of the `input` elements so that it corresponds to the model property that the `input` element displays. The `name` attribute is used by the MVC Framework default model binder to work out which `input` elements contain values for the model type properties when processing a post request. If you omit the `name` attribute, your form will not work properly. I describe model binding fully in Chapter 24, including how you can change this behavior.

I have created the `Views/Shared` folder and added a layout file called `_Layout.cshtml` with the contents shown in Listing 21-15. This is a simple layout with some CSS for the `input` elements in the form.

Listing 21-15. The Contents of the _Layout.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

You can see the basic form functionality by starting the application and navigating to the `/Home/CreatePerson` URL, as shown in Figure 21-6. Because the form data is not used by the application in any way, clicking the `Submit` button will just cause whatever data is in the form to be redisplayed.

Figure 21-6. *Using the simple HTML form in the example application*

In Listing 21-16, you can see the HTML that the example MVC application has sent to the browser. I will use this to show you changes caused by helper methods.

Listing 21-16. The HTML Sent to the Browser for the Example Form

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
```

```
    <style type="text/css">
        label { display: inline-block; width: 100px; }
        div.dataElem { margin: 5px; }
    </style>
</head>
<body>

<h2>CreatePerson</h2>

<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" />
    </div>
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```

■ **Note** Using the helper methods to generate HTML elements like `form` and `input` is not compulsory. If you prefer, you can code them using static HTML tags and populate values using view data or view model objects, just as I did in this section. The HTML that I generate from the helper methods in the following sections is clean and there are no special attribute values or sneaky tricks that mean you have to use them. But they make it easy to ensure that the HTML is in sync with the application so that, for example, changes in routing configuration will be reflected automatically in your forms. The helpers are there for convenience, rather than because they create essential or special HTML, and you do not have to use them if they do not suit your development style.

Creating Form Elements

Two of the most useful (and most commonly used) helpers are `Html.BeginForm` and `Html.EndForm`. These helpers create HTML `form` tags and generate a valid `action` attribute for the form that is based on the routing mechanism for the application.

There are 13 different versions of the `BeginForm` method, allowing you to be increasingly specific about how the resulting `form` element will be generated. I only need the most basic for the example application, which takes no arguments and creates a `form` element whose `action` attribute ensures that the form will be posted back to the same action method which led to the current view being generated. You can see how I have applied this overload of `BeginForm` and the `EndForm` helper in Listing 21-17. The `EndForm` helper has only one definition and it just closes the form element by adding `</form>` to the view.

Listing 21-17. Using the BeginForm and EndForm Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@{Html.BeginForm();}
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
@{Html.EndForm();}
```

Notice that I had to treat the call to the helper methods as a C# statements. This is because of the way that the helper methods write their tags to the output. It is a pretty ugly result, but it doesn't matter because these helpers are rarely used in this way. A much more common approach is shown in Listing 21-18, which wraps the call to the `BeginForm` helper method in a `using` expression. At the end of the `using` block, the .NET runtime calls the `Dispose` method on the object returned by the `BeginForm` method, which calls the `EndForm` method for you. (You can see how this works by downloading the source code for the MVC Framework and taking a look at the `System.Web.Mvc.Html.FormExtensions` class.)

Listing 21-18. Creating a Self-Closing Form in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginForm()) {
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
```

```
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

This approach, known as a self-closing form, is the one I use in my own projects. I like the way that the code block contains the form and makes it clear which elements will appear between the opening and closing `form` tags.

The other 12 variations for the `BeginForm` method allow you to change different aspects of the `form` element that is created. There is a lot of repetition in these overloads, as they allow you to be incrementally more specific about the details you provide. In Table 21-5 I have listed the most important overloads, which are the ones that you will use on a regular basis in an MVC application. The other overloads of the `BeginForm` method—which I have omitted—are provided for compatibility with the versions of the MVC Framework that were released before C# had support for creating dynamic objects.

Table 21-5. *The Overloads of the BeginForm Helper Method*

Overload	Description
`BeginForm()`	Creates a form which posts back to the action method it originated from
`BeginForm(`**`action`**`,` **`controller`**`)`	Creates a form which posts back to the action method and controller, specified as strings
`BeginForm(action, controller,` **`method`**`)`	As for the previous overload, but allows you to specify the value for the method attribute using a value from the `System.Web.Mvc.FormMethod` enumeration
`BeginForm(action, controller, method,` **`attributes`**`)`	As for the previous overload, but allows you to specify attributes for the form element an object whose properties are used as the attribute names
`BeginForm(action, controller,` **`routeValues`**`, method, attributes)`	As for the previous overload, but allows you to specify values for the variable route segments in your application routing configuration as an object whose properties correspond to the routing variables

I have shown you the simplest version of the `BeginForm` method, which is all I need for the example app, but in Listing 21-19 you can see the most complex, in which I specify additional information for how the `form` element should be constructed.

Listing 21-19. Using the Most Complex Overload of the BeginForm Method in the CreatePerson.cs File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using (Html.BeginForm("CreatePerson", "Home",
    new { id = "MyIdValue" }, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {
```

```
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

In this example, I have explicitly specified some details that would have been inferred automatically by the MVC Framework, such as the action name and the controller. I also specified that the form should be submitted using the HTTP POST method, which would have been used anyway.

The more interesting arguments are the ones that set values for the route variable and set attributes on the form element. I route values arguments to specify a value for the `id` segment variable in the default route added by Visual Studio to the `/App_Start/RouteConfig.cs` file when the project was created and I defined `class` and `data` attributes. (Data attributes are custom attributes which you can add to elements to make processing HTML content.) Here is the HTML `form` tag that this call to `BeginForm` produces:

```
...
<form action="/Home/CreatePerson/MyIdValue" class="personClass" data-formType="person"
    method="post">
...
```

You can see that the value for the `id` attribute has been appended to the target URL and that the class and data attributes have been applied to the element. Notice that I specified an attribute called `data_formType` in the call to `BeginForm` but ended up with a `data-formType` attribute in the output. You cannot specify property names in a dynamic object that contain hyphens, so I use an underscore that is then automatically mapped to a hyphen in the output, neatly side-stepping a mismatch between the C# and HTML syntaxes. (And, of course, I had to prefix the property name `class` with a `@` so that I can use a C#-reserved keyword as a property name for the `class` attribute.)

Specifying the Route Used by a Form

When you use the `BeginForm` method, the MVC Framework finds the first route in the routing configuration that can be used to generate a URL that will target the required action and controller. In essence, you leave the route selection to be figured out for you. If you want to ensure that a particular route is used, then you can use the `BeginRouteForm` helper method instead. To demonstrate this feature, I have added a new route to the `/App_Start/RouteConfig.cs` file, as shown in Listing 21-20.

Listing 21-20. Adding a New Route to the RouteConfig.cs File

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

```
namespace HelperMethods {
    public class RouteConfig {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
            );

            routes.MapRoute(
                name: "FormRoute",
                url: "app/forms/{controller}/{action}"
            );
        }
    }
}
```

If I call the `BeginForm` method with this routing configuration, I will end up with a `form` element whose `action` attribute contains a URL which is created from the default route. In Listing 21-21, you can see how I have specified that the new route should be used through the `BeginRouteForm` method.

Listing 21-21. Specifying Which Route Should Be Used in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

This produces the following form tag, whose action attribute corresponds to the structure of the new route:

```
...
<form action="/app/forms/Home/CreatePerson" class="personClass"
    data-formType="person" method="post">
...
```

■ **Tip** There are a range of different overloads for the BeginRouteForm method allowing you to specify differing degrees of details for the form element, just as with the BeginForm method. These follow the same structure as their BeginForm counterparts. See the API documentation for details.

Using Input Helpers

An HTML form is of no use unless you also create some input elements. Table 21-6 shows the basic helper methods that are available to create input elements and gives examples of the HTML they produce. For all of these helper methods, the first argument is used to set the value of the id and name attributes for the input element and the second argument is used to set the value attribute.

Table 21-6. *Basic Input HTML Helpers*

HTML Element	Example

Check box	`Html.CheckBox("myCheckbox", false)`
	Output:
	`<input id="myCheckbox" name="myCheckbox" type="checkbox" value="true" />`
	`<input name="myCheckbox" type="hidden" value="false" />`

Hidden field	`Html.Hidden("myHidden", "val")`
	Output:
	`<input id="myHidden" name="myHidden" type="hidden" value="val" />`

Radio button	`Html.RadioButton("myRadiobutton", "val", true)`
	Output:
	`<input checked="checked" id="myRadiobutton" name="myRadiobutton"`
	`type="radio" value="val" />`

Password	`Html.Password("myPassword", "val")`
	Output:
	`<input id="myPassword" name="myPassword" type="password" value="val" />`

Text area	`Html.TextArea("myTextarea", "val", 5, 20, null)`
	Output:
	`<textarea cols="20" id="myTextarea" name="myTextarea" rows="5">`
	`val</textarea>`

Text box	`Html.TextBox("myTextbox", "val")`
	Output:
	`<input id="myTextbox" name="myTextbox" type="text" value="val" />`

Each of these helpers is overloaded. The table shows the simplest version, but you can provide an additional `object` argument that you use to specify HTML attributes, just as I did with the `form` element in the previous section.

■ **Note** Notice that the checkbox helper (`Html.CheckBox`) renders *two* `input` elements. It renders a checkbox and then a hidden `input` element of the same name. This is because browsers do not submit a value for checkboxes when they are not selected. Having the hidden control ensures that the MVC Framework will get a value from the hidden field when this happens.

You can see how I have used these basic `input` helper methods in Listing 21-22.

Listing 21-22. Using the Basic Input Element Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("personId", @Model.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName", @Model.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName", @Model.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```

You can see the HTML `input` elements that this view produces in Listing 21-23. The output is similar to the original `form` element, but you can see some hints of the MVC Framework have appeared in the form of some `data` attributes which have been added to support form validation, which I describe in Chapter 25.

Listing 21-23. The Input Elements Created by the Basic Input Helper Methods

```
...
<form action="/app/forms/Home/CreatePerson" class="personClass" data-formType="person"
        method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input data-val="true" data-val-number="The field PersonId must be a number."
            data-val-required="The PersonId field is required." id="personId"
            name="personId" type="text" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input id="firstName" name="firstName" type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input id="lastName" name="lastName" type="text" value="" />
    </div>
    <input type="submit" value="Submit" />
</form>
...
```

Generating the Input Element from a Model Property

The helper methods I used in the previous section are fine, but I still have to ensure that the value I pass as the first argument corresponds to the model value I pass as the second argument. If they are not consistent, then the MVC Framework will not be able to reconstruct the model object from the form data because the `name` attributes and the forms values of the `input` elements will not match. For each of the methods I listed in Table 21-6, there is an alternative overload which takes a `single` string argument, which I have used in Listing 21-24.

Listing 21-24. Generating the Input Element from the Model Property Name in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("PersonId")
    </div>
```

```
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName")
    </div>
    <input type="submit" value="Submit" />
}
```

The `string` argument is used to search the view data, `ViewBag`, and view model to find a corresponding data item that can be used as the basis for the input element. So, for example, if you call `@Html.TextBox("DataValue")`, the MVC Framework tries to find some item of data that corresponds with the key `DataValue`. The following locations are checked:

- `ViewBag.DataValue`
- `@Model.DataValue`

The first value that is found is used to set the `value` attribute of the generated HTML. (The last check, for `@Model.DataValue`, works only if the view model for the view contains a property or field called `DataValue`.)

If I specify a string like `DataValue.First.Name`, the search becomes more complicated. The MVC Framework will try different arrangements of the dot-separated elements, such as the following:

- `ViewBag.DataValue.First.Name`
- `ViewBag.DataValue["First"].Name`
- `ViewBag.DataValue["First.Name"]`
- `ViewBag.DataValue["First"]["Name"]`

Many permutations will be checked. Once again, the first value that is found will be used, terminating the search. There is an obvious performance consideration to this technique, but bear in mind that usually only a few items are in the view bag, so it does not take much time to search through them.

Using Strongly Typed Input Helpers

For each of the basic input helpers that I described in Table 21-6, there are corresponding *strongly typed* helpers. You can see these helpers in Table 21-7 along with samples of the HTML they produce. These helpers can be used only with strongly typed views. (Some of these helpers generate attributes that help with client-side form validation. I have omitted these from the table for brevity.)

Table 21-7. *Strongly Typed Input HTML Helpers*

HTML Element	Example

Check box	`Html.CheckBoxFor(x => x.IsApproved)`
	Output:
	`<input id="IsApproved" name="IsApproved" type="checkbox" value="true" />`
	`<input name="IsApproved" type="hidden" value="false" />`

Hidden field	`Html.HiddenFor(x => x.FirstName)`
	Output:
	`<input id="FirstName" name="FirstName" type="hidden" value="" />`

Radio button	`Html.RadioButtonFor(x => x.IsApproved, "val")`
	Output:
	`<input id="IsApproved" name="IsApproved" type="radio" value="val" />`

Password	`Html.PasswordFor(x => x.Password)`
	Output:
	`<input id="Password" name="Password" type="password" />`

Text area	`Html.TextAreaFor(x => x.Bio, 5, 20, new{})`
	Output:
	`<textarea cols="20" id="Bio" name="Bio" rows="5">Bio value</textarea>`

Text box	`Html.TextBoxFor(x => x.FirstName)`
	Output:
	`<input id="FirstName" name="FirstName" type="text" value="" />`

The strongly typed input helpers work on lambda expressions. The value that is passed to the expression is the view model object, and you can select the field or property that will be used to set the `value` attribute. You can see how I have used this kind of helper in the `CreatePerson.cshtml` view from the example application in Listing 21-25.

Listing 21-25. Using the Strongly Typed Input Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
```

```
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <input type="submit" value="Submit" />
}
```

The HTML generated by these helpers is not any different, but I use the strongly typed helper methods in my own projects because they reduce the chances of causing an error by mistyping a property name.

Creating Select Elements

Table 21-8 shows the helper methods that can be used to create `select` elements. These can be used to select a single item from a drop-down list or present a multiple-item `select` element that allows several items to be selected. As with the other form elements, there are versions of these helpers that are weakly and strongly typed.

Table 21-8. *HTML Helpers That Render Select Elements*

HTML Element	Example

Drop-down list	`Html.DropDownList("myList", new SelectList(new [] {"A", "B"}), "Choose")`
	Output:
	`<select id="myList" name="myList">`
	`  <option value="">Choose</option>`
	`  <option>A</option>`
	`  <option>B</option>`
	`</select>`

Drop-down list	`Html.DropDownListFor(x => x.Gender, new SelectList(new [] {"M", "F"}))`
	Output:
	`<select id="Gender" name="Gender">`
	`  <option>M</option>`
	`  <option>F</option>`
	`</select>`

Multiple-select	`Html.ListBox("myList", new MultiSelectList(new [] {"A", "B"}))`
	Output:
	`<select id="myList" multiple="multiple" name="myList">`
	`  <option>A</option>`
	`  <option>B</option>`
	`</select>`

Multiple-select	`Html.ListBoxFor(x => x.Vals, new MultiSelectList(new [] {"A", "B"}))`
	Output:
	`<select id="Vals" multiple="multiple" name="Vals">`
	`  <option>A</option>`
	`  <option>B</option>`
	`</select>`

The `select` helpers take `SelectList` or `MultiSelectList` parameters. The difference between these classes is that `MultiSelectList` has constructor options that let you specify that more than one item should be selected when the page is rendered initially.

Both of these classes operate on IEnumerable sequences of objects. In Table 21-8, I created inline arrays that contained the list items I wanted displayed, but a nice feature of SelectList and MultiSelectList is that they will extract values from objects, including the model object, for the list items. You can see how I have created a select element for the Role property of the Person model in Listing 21-26.

Listing 21-26. Creating a Select Element for the Person.Role Property in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>CreatePerson</h2>

@using(Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType="person"})) {

    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,
            new SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}
```

I defined the Role property so that it is a value from the Role enumeration defined in the same class file. Because the SelectList and MultiSelectList objects operate on IEnumerable objects, I have to use the Enum.GetNames method to be able to use the Role enum as the source for the select element. You can see the HTML that the latest version of the view creates, including the select element, in Listing 21-27.

Listing 21-27. The HTML Generated by the CreatePerson View

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
```

```
    <style type="text/css">
        label { display: inline-block; width: 100px;}
        .dataElem { margin: 5px;}
    </style>
</head>
<body>

<h2>CreatePerson</h2>

<form action="/app/forms/Home/CreatePerson" class="personClass"
            data-formType="person" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input data-val="true" data-val-number="The field PersonId must be a number." data-val-
required="The PersonId field is required." id="PersonId" name="PersonId" type="text" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input id="FirstName" name="FirstName" type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input id="LastName" name="LastName" type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Role</label>
        <select data-val="true" data-val-required="The Role field is required."
                id="Role" name="Role">
            <option selected="selected">Admin</option>
            <option>User</option>
            <option>Guest</option>
        </select>
    </div>
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```

Summary

In this chapter, I introduced the concept of helper methods, which you can use in views to generate chunks of content in a reusable way. I started by showing you how to create custom inline and external helpers and then showed you the helpers that are available to create HTML `form`, `input` and `select` elements. In the next chapter, I continue on this theme and show you how to use *templated helpers*.