CHAPTER 22

Templated Helper Methods

The HTML helpers that I looked at in the previous chapter, such as `Html.CheckBoxFor` and `Html.TextBoxFor`, generate a specific type of element, which means that I have to decide in advance what kinds of elements should be used to represent model properties and to manually update the views if the type of a property changes.

In this chapter, I demonstrate the *templated helper methods*, with which I specify the property I want displayed and let the MVC Framework figure out what HTML elements are required. This is a more flexible approach to displaying data to the user, although it requires some initial care and attention to set up. Table 22-1 provides the summary for this chapter.

Table 22-1. *Chapter Summary*

Problem	Solution	Listing
Generate an element that can be used to edit a model property	Use the `Html.Editor` and `Html.EditorFor` helpers	1–5, 18
Generate labels for and read-only representations of model properties	Use the `Html.Label` and `Html.Display` helpers	6–8
Generate elements for a complete model object	Use the `DisplayForModel`, `EditorForModel` and `LabelForModel` helpers	9–10
Hide an element from the user when generating elements using a whole-model helper or prevent it from being edited	Apply the `HiddenInput` attribute to the property	11–12
Set the label that will be used to display model properties	Use the `DisplayName` and `Display` attributes	13
Specify the way in which model properties are displayed	Use the `DataType` attribute	14
Specify the template used to display a model property	Use the `UIHint` attribute	15
Define model metadata separately from the model type	Create a buddy class and use the `MetadataType` attribute	16–17
Change the elements that are generated for a model property	Create a custom template	19–22

Preparing the Example Project

In this chapter I am going to continue using the `HelperMethod` project that I created in Chapter 21. In that project, I created a `Person` model class along with a couple of supporting types. As a reminder, I have listed these in Listing 22-1.

Listing 22-1. The Contents of the Person.cs File

```
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
    }
}
```

The example project contains a simple Home controller that I use to display forms and receive form posts. You can see the definition of the HomeController class in Listing 22-2.

Listing 22-2. The Contents of 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);
        }
    }
}
```

It is the two CreatePerson action methods that I will be using in this chapter, both of which render the /Views/Home/CreatePerson.cshtml view file. In Listing 22-3, you can see the CreatePerson view from the end of the last chapter, with a simple change.

Listing 22-3. The Contents of the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<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 made one addition, which I have marked in bold. By default, the helper methods will add data attributes to the HTML elements they generate to support the kind of form validation I showed you when I created the SportsStore application. I do not want those attributes in this chapter, so I have used the Html.EnableClientValidation method to disable them for the CreatePerson view. The client validation feature is still enabled for the rest of the application and I will explain how validation works in detail (including the purpose of the data attributes) in Chapter 25.

Using Templated Helper Methods

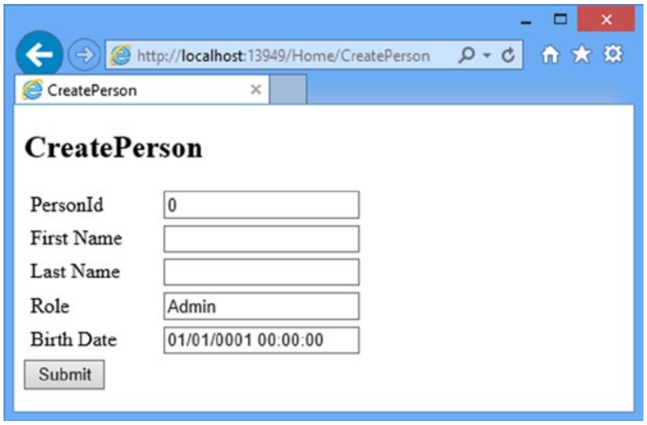
The first templated helper methods that I am going to look at are `Html.Editor` and `Html.EditorFor`. The `Editor` method takes a string argument that specifies the property for which editor element is required. The helper follows the search process that I described in Chapter 20 to locate a corresponding property in the view bag and model object. The `EditorFor` method is the strongly typed equivalent, which allows you to use a lambda expression to specify a model property that you want the editor element for.

In Listing 22-4, you can see how I have applied both the `Editor` and `EditorFor` helper methods to the `CreatePerson` view. As I mentioned in the last chapter, I prefer to use the strongly typed helpers because they reduce the chances of causing an error by mistyping the property name, but I have used both types in this listing just to demonstrate that you can mix and match as you see fit.

Listing 22-4. Using the Editor and EditorFor Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>

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

    <div class="dataElem">
        <label>PersonId</label>
        @Html.Editor("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.Editor("FirstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.EditorFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.EditorFor(m => m.Role)
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
        @Html.EditorFor(m => m.BirthDate)
    </div>
    <input type="submit" value="Submit" />
}
```

The HTML elements that are created by the `Editor` and `EditorFor` methods are the same. The only difference is the way that you specify the property that the editor elements are created for. You can see the effect of the changes that I have made by starting the example application and navigating to the `/Home/CreatePerson` URL, as shown in Figure 22-1.

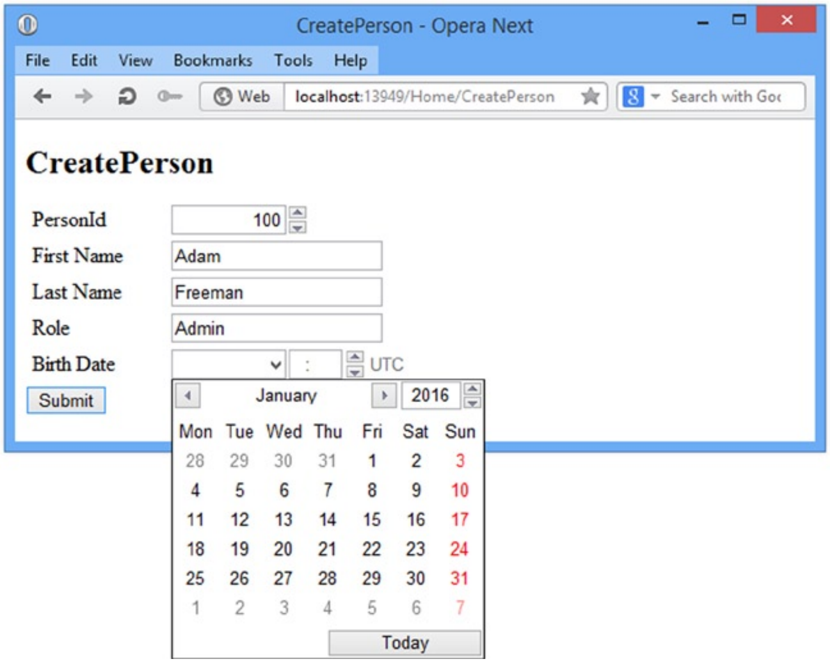


CreatePerson

PersonId	0
First Name	
Last Name	
Role	Admin
Birth Date	01/01/0001 00:00:00

Figure 22-1. *Using the Editor and EditorFor helper methods in a form*

Other than the addition of the `BirthDate` property, this does not look different from the kind of form that I was creating in Chapter 21. However, there is a pretty substantial change, which you can see if you use a different browser. In Figure 22-2, you can see the same URL displayed in the Opera browser (which you can get from `www.opera.com`).



CreatePerson

PersonId	100
First Name	Adam
Last Name	Freeman
Role	Admin
Birth Date	: UTC

Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

Figure 22-2. *Displaying a form created using the Editor and EditorFor helper methods*

Notice that the elements for the `PersonId` and `BirthDate` properties look different. The `PersonId` element has spinner arrows (allowing you to increment and decrement the value) and the `BirthDate` element is presented with a date picker.

The HTML5 specification defines different types of `input` element that can be used to edit common data types, such as numbers and dates. The `Helper` and `HelperFor` methods use the type of the property I want to edit to select one of those new `input` element types. You can see this in Listing 22-5, where I have shown the HTML that was generated for the form.

Listing 22-5. The HTML Input Elements Created by the Editor and EditorFor Helper Methods

```
<!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 class="text-box single-line" id="PersonId" name="PersonId"
            type="number" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input class="text-box single-line" id="FirstName" name="FirstName"
            type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input class="text-box single-line" id="LastName" name="LastName"
            type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Role</label>
        <input class="text-box single-line" id="Role" name="Role"
            type="text" value="Admin" />
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
        <input class="text-box single-line" id="BirthDate" name="BirthDate"
            type="datetime" value="01/01/0001 00:00:00" />
    </div>
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```

The `type` attribute specifies which kind of `input` element should be displayed by the browser. The helper methods have specified the `number` and `datetime` types for the `PersonId` and `BirthDate` properties and the `text` type, which is the default for the other properties. The reason that we only see these types in Opera is because the HTML5 features are still not widely supported, even in the latest version of Internet Explorer and Chrome.

■ **Tip** Most web UI toolkits include date pickers that you can use instead of relying on the HTML5 `input` element types. If you have not already selected such a toolkit for a project, then I suggest you start with jQuery UI (`http://jqueryui.com`), which is an open-source toolkit, built on jQuery.

You can see that by using the templated helper methods I have been able to tailor the form elements to the content, although not in an especially useful way, in part because not all browsers can display the HTML5 `input` element types and in part because some properties, such as `Role`, are not displayed in a helpful way. I will show you how to provide the MVC Framework with additional information that will improve the HTML that the helper methods produce. But I am going to show you the other templated helpers that are available before I get into the detail. You can see the complete set of helpers in Table 22-2 and I demonstrate each of them in the sections that follow.

Table 22-2. *The MVC Templated HTML Helpers*

Helper	Example	Description
`Display`	`Html.Display("FirstName")`	Renders a read-only view of the specified model property, choosing an HTML element according to the property's type and metadata
`DisplayFor`	`Html.DisplayFor(x => x.FirstName)`	Strongly typed version of the previous helper
`Editor`	`Html.Editor("FirstName")`	Renders an editor for the specified model property, choosing an HTML element according to the property's type and metadata
`EditorFor`	`Html.EditorFor(x => x.FirstName)`	Strongly typed version of the previous helper
`Label`	`Html.Label("FirstName")`	Renders an HTML `<label>` element referring to the specified model property
`LabelFor`	`Html.LabelFor(x => x.FirstName)`	Strongly typed version of the previous helper

Generating Label and Display Elements

To demonstrate the other helper methods, I am going to add a new action method and view to the example that will display a read-only view of the data submitted from the HTML form. First, I have updated the `HttpPost` version of the `CreatePerson` action in the `Home` controller, as shown in Listing 22-6.

Listing 22-6. Specifying a Different View 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("DisplayPerson", person);
        }
    }
}
```

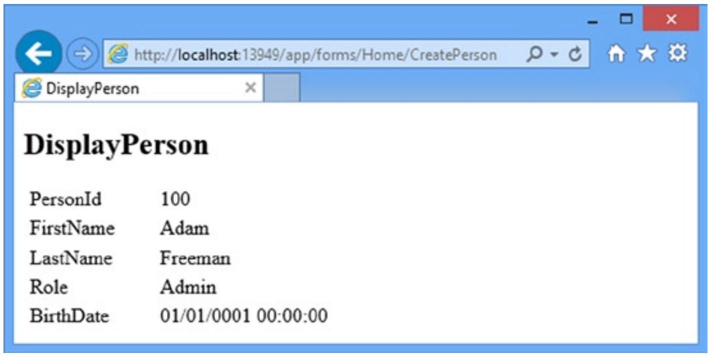
I added the `DisplayPerson.cshtml` view file to the `/Views/Home` folder, and you can see the contents of this file in Listing 22-7.

Listing 22-7. The Contents of the DisplayPerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "DisplayPerson";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>DisplayPerson</h2>
<div class="dataElem">
    @Html.Label("PersonId")
    @Html.Display("PersonId")
</div>
```

```
<div class="dataElem">
    @Html.Label("FirstName")
    @Html.Display("FirstName")
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.LastName)
    @Html.DisplayFor(m => m.LastName)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.Role)
    @Html.DisplayFor(m => m.Role)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.BirthDate)
    @Html.DisplayFor(m => m.BirthDate)
</div>
```

You can see the output that this new view produces by starting the application, navigating to the /Home/CreatePerson URL, filling in the form and clicking the Submit button. The result is shown in Figure 22-3 and you can see that I have taken a small step backward, because the Label and LabelFor helpers have just used the property names as the content for the labels.



DisplayPerson

```
PersonId     100
FirstName    Adam
LastName     Freeman
Role         Admin
BirthDate    01/01/0001 00:00:00
```

Figure 22-3. *Using helpers to generate a read-only view of the Person object*

You can see the output that these helper methods produce in Listing 22-8. Notice that the Display and DisplayFor methods do not generate an HTML element by default. They just emit the value of the property they operate on.

Listing 22-8. The HTML Generated from the DisplayPerson View

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

<h2>DisplayPerson</h2>
<div class="dataElem">
    <label for="PersonId">PersonId</label>
    100
</div>
<div class="dataElem">
    <label for="FirstName">FirstName</label>
    Adam
</div>
<div class="dataElem">
    <label for="LastName">LastName</label>
    Freeman
</div>
<div class="dataElem">
    <label for="Role">Role</label>
    Admin
</div>
<div class="dataElem">
    <label for="BirthDate">BirthDate</label>
    01/01/0001 00:00:00
</div>

</body>
</html>
```

Although these helpers may not seem especially useful at the moment, I will show you how to change their behavior shortly in order to produce output that is much more the kind of thing that you would want to display to users.

Using Whole-Model Templated Helpers

I have been using templated helpers which generate output for a single property, but the MVC Framework also defines helpers that operate on the entire objects, a process known as *scaffolding*. There are scaffolding helpers available, as shown in Table 22-3.

Table 22-3. *The MVC Scaffolding Templated Helper methods*

Helper	Example	Description
`DisplayForModel`	`Html.DisplayForModel()`	Renders a read-only view of the entire model object
`EditorForModel`	`Html.EditorForModel()`	Renders editor elements for the entire model object
`LabelForModel`	`Html.LabelForModel()`	Renders an HTML `<label>` element referring to the entire model object

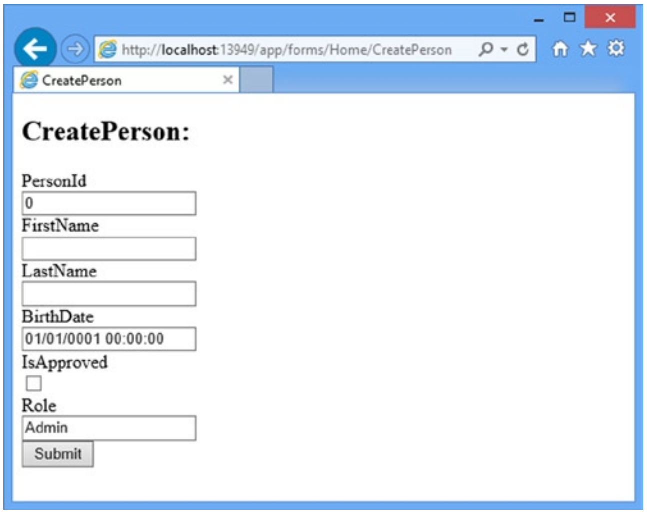
■ **Tip** This is not the same kind of scaffolding that Microsoft added to Visual Studio to create MVC components like controllers and views, but the basic idea is the same in which output is generated based on the characteristics of a data type. In the case of Visual Studio, the output from the scaffolding is a class or Razor file and for the templated helpers, the output is HTML.

In Listing 22-9, you can see how I have used the `LabelForModel` and `EditorForModel` helper methods to simplify the `CreatePerson.cshtml` view.

Listing 22-9. Using the Scaffolding Helper Methods in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}

<h2>CreatePerson: @Html.LabelForModel()</h2>

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

    @Html.EditorForModel()

    <input type="submit" value="Submit" />
}
```

You can see the effect of the scaffold helpers in Figure 22-4. Once again, you can see what the helpers are trying to do but that things are not quite right yet. The `LabelForModel` helper has not generated a useful label. Although more properties from the `Person` model object are being shown than I defined manually in previous examples, not everything is visible (such as the `Address` property) and what is visible is not always useful (such as the `Role` property, which would be more usefully expressed as a `select` instead of an `input` element).



CreatePerson:

PersonId
0
FirstName

LastName

BirthDate
01/01/0001 00:00:00
IsApproved

Role
Admin

Figure 22-4. *Using the scaffolding helpers to create an editor for the Person model object*

Part of the problem is that the HTML that the scaffolding helpers generate doesn't correspond to the CSS styles that I defined in the `/Views/Shared/_Layout.cshtml` file in the previous chapter. Here is an example of the HTML generated to edit the `FirstName` property:

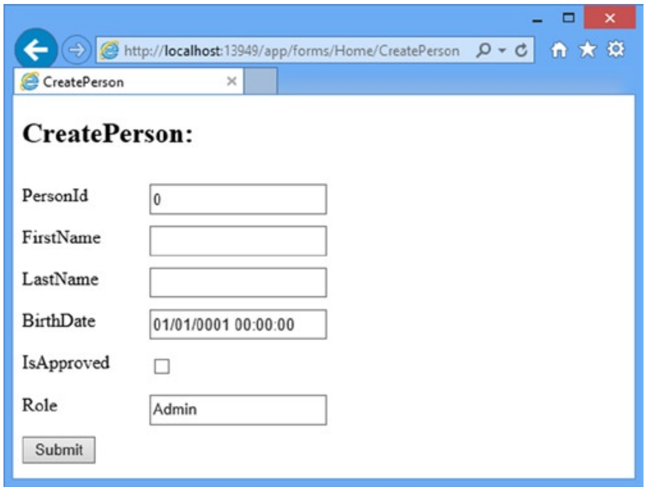
```
...
<div class="editor-label">
    <label for="FirstName">FirstName</label>
</div>
<div class="editor-field">
    <input class="text-box single-line" id="FirstName" name="FirstName"
        type="text" value="" />
</div>
...
```

I can tidy up the appearance of a view by adding styles to the layout that correspond to the CSS `class` values added to the `div` and `input` elements by the scaffolding helpers. In Listing 22-10, you can see the changes I made to the `_Layout.cshtml` file.

Listing 22-10. Making Changes to the CSS Styles Defined in 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; }
        h2 > label { width: inherit; }
        .editor-label, .editor-field { float: left; margin-top: 10px;}
        .editor-field input { height: 20px; }
        .editor-label { clear: left;}
        .editor-field { margin-left: 10px;  }
        input[type=submit] { float: left; clear: both; margin-top: 10px; }
        .column { float: left; margin: 10px;}
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

These new styles produce something that is more in keeping with the layout I used in earlier examples, as shown in Figure 22-5.



http://localhost:13949/app/forms/Home/CreatePerson

CreatePerson

CreatePerson:

PersonId	0
FirstName	
LastName	
BirthDate	01/01/0001 00:00:00
IsApproved	☐
Role	Admin

Submit

Figure 22-5. *The effect of styling elements using the classes defined by the scaffolding helper methods*

Using Model Metadata

As you have seen, the templated helpers have no special knowledge about the application and its model data types, and so I end up with HTML that is not what exactly what I desire. I want the benefits that come with simpler views, but I need to improve the quality of the output that the helper methods generate before I can use them seriously.

I cannot blame the templated helpers in these situations; the HTML that is generated is based on a best guess about what I want. This is, of course, the problem I have with all scaffolding, which has to make a best-effort attempt based on a generic understanding of the application. Fortunately, the templated helpers can be improved by using model metadata to provide guidance about how to handle model types. Metadata is expressed using C# attributes, where attributes and parameter values provide a range of instructions to the view helpers. The metadata is applied to the model class, which the helper methods consult when they generate HTML elements. In the following sections, I show you how to use metadata to provide directions to the helpers for labels, displays, and editors.

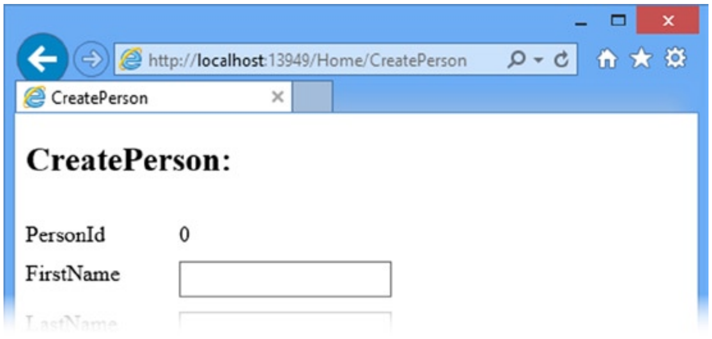
Using Metadata to Control Editing and Visibility

In the Person class, the PersonId property is one that I do not want the user to be able to see or edit. Most model classes have at least one such property, often related to the underlying storage mechanism—a primary key that is managed by a relational database, for example, which I demonstrated when I created the SportsStore application. I can use the HiddenInput attribute, which causes the helper to render a hidden input field. You can see how I have applied the HiddenAttribute to the Person class in Listing 22-11.

Listing 22-11. Using the HiddenInput Attribute in the Person.cs File

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

namespace HelperMethods.Models {

    public class Person {
        [HiddenInput]
        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; }
    }

    // ...other types omitted for brevity...
}
```

When this attribute has been applied, the Html.EditorFor and Html.EditorForModel helpers will render a read-only view of the decorated property (which is the term used to describe a property to which an attribute has been applied), as shown in Figure 22-6, which shows the effect of starting the application and navigating to the /Home/CreatePerson URL.



CreatePerson:

PersonId 0

FirstName

Figure 22-6. *Creating a read-only representation of a property in an editor*

The value of the PersonId property is shown, but the user cannot edit it. The HTML that is generated for the property is as follows:

```
...
<div class="editor-field">
    0
    <input id="PersonId" name="PersonId" type="hidden" value="0" />
</div>
...
```

The value of the property (0 in this case) is rendered literally, but the helper also includes a hidden input element for the property, which is helpful for HTML forms because it ensures that a value for the property is submitted along with the rest of the form. This is something I will return to when I look at model binding in Chapter 24 and model validation in Chapter 25. If I want to hide a property entirely, then I can set the value of the DisplayValue property in the DisplayName attribute to false, as shown in Listing 22-12.

Listing 22-12. Using the HiddenInput Attribute to Hide a Property in the Person.cshtml File

```
...
public class Person {
    [HiddenInput(DisplayValue=false)]
    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; }
}
...
```

When I use the Html.EditorForModel helper on a Person object, a hidden input will be created so that the value for the PersonId property will be included in any form submissions, but the label and the value will be omitted. This has the effect of hiding the PersonId property from the user, as shown by Figure 22-7.

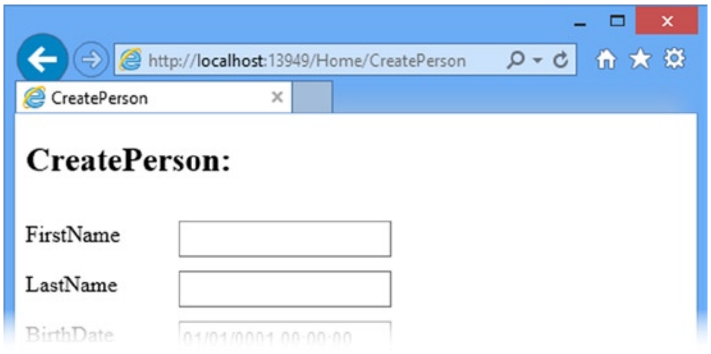


Figure 22-7. *Hiding model object properties from the user*

If I have chosen to render HTML for individual properties, I can still create the hidden input for the PersonId property by using the Html.EditorFor helper, like this:

```
...
@Html.EditorFor(m => m.PersonId)
...
```

The HiddenInput property is detected, and if DisplayValue is true, then the following HTML is generated:

```
...
<input id="PersonId" name="PersonId" type="hidden" value="1" />
...
```

EXCLUDING A PROPERTY FROM SCAFFOLDING

To completely exclude a property from the generated HTML, I can use the ScaffoldColumn attribute. Whereas the HiddenInput attribute includes a value for the property in a hidden input element, the ScaffoldColumn attribute can mark a property as being entirely off limits for the scaffolding process. Here is an example of the attribute in use:

```
...
[ScaffoldColumn(false)]
public int PersonId { get; set; }
...
```

When the scaffolding helpers see the ScaffoldColumn attribute applied in this way, they skip over the property entirely; no hidden input elements will be created, and no details of this property will be included in the generated HTML. The appearance of the generated HTML will be the same as if I had used the HiddenInput attribute, but no value will be returned for the property during a form submission. This has an effect on model binding, which I discuss in Chapter 24. The ScaffoldColumn attribute doesn't have an effect on the per-property helpers, such as EditorFor. If I call @Html.EditorFor(m => m.PersonId) in a view, then an editor for the PersonId property will be generated, even when the ScaffoldColumn attribute is present.

Using Metadata for Labels

By default, the Label, LabelFor, LabelForModel, and EditorForModel helpers use the names of properties as the content for the label elements they generate. For example, if I render a label like this:

```
...
@Html.LabelFor(m => m.BirthDate)
...
```

the HTML element that is generated will be as follows:

```
...
<label for="BirthDate">BirthDate</label>
...
```

Of course, the names given to properties are often not suitable for display to the user. To that end, I can apply the DisplayName attribute from the System.ComponentModel.DataAnnotations namespace, passing in the value I want as a value for the Name property. Listing 22-13 demonstrates this attribute applied to the Person class.

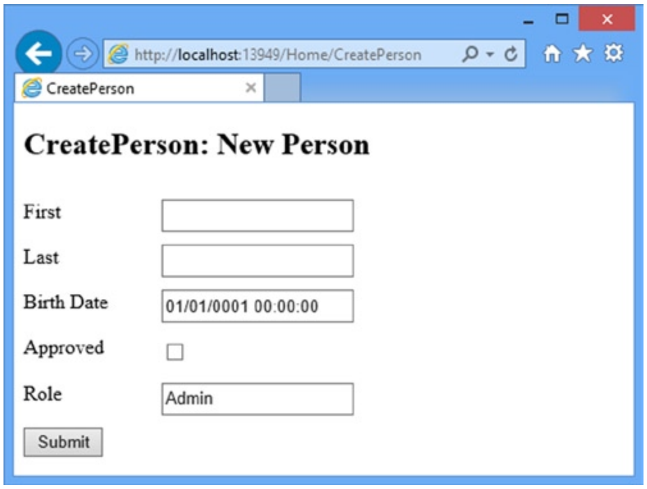
Listing 22-13. Using the DisplayName Attribute to Define a Label in the Person.cs File

```
using System;
using System.Web.Mvc;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public class Person {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        public DateTime BirthDate { get; set; }

        public Address HomeAddress { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ...other types omitted for brevity...
}
```

When the label helpers render a label element for the BirthDate property, they will detect the Display attribute and use the value of the Name parameter for the inner text, like this:

```
...
<label for="BirthDate">Birth Date</label>
...
```

The helpers also recognize the DisplayName attribute, which can be found in the System.ComponentModel namespace. This attribute has the advantage of being able to be applied to classes, which allows me to use the Html.LabelForModel helper. You can see how I have applied this attribute to the Person class in the listing. (I can apply the DisplayName attribute to properties as well, but I tend to use this attribute only for model classes, for no reason other than habit.) You can see the effect of the Display and DisplayName attributes in Figure 22-8.



CreatePerson: New Person

First

Last

Birth Date 01/01/0001 00:00:00

Approved

Role Admin

Submit

***Figure 22-8.** Using the Display and DisplayName attributes to control labels*

Using Metadata for Data Values

I can also use metadata to provide instructions about how a model property should be displayed. I can use this to deal with the fact that the BirthDate property is displayed with a time when I really just want a date, for example. I control the way that data values are displaying using the DataType attribute, which you can see applied to the Person class in Listing 22-14.

Listing 22-14. Applying the DataType Attribute to the Person.cs File

```
...
[DisplayName("New Person")]
public class Person {
    [HiddenInput(DisplayValue=false)]
    public int PersonId { get; set; }

    [Display(Name="First")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    public Address HomeAddress { get; set; }

    [Display(Name="Approved")]
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
}
...
```

The DataType attribute takes a value from the DataType enumeration as a parameter. In the example I have specified the DataType.Date value, which causes the templated helpers to render the value of the BirthDate property as a date without a time component, as shown in Figure 22-9.

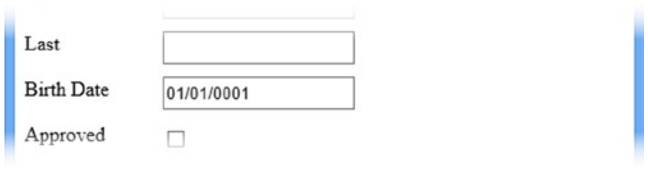

Last
Birth Date: 01/01/0001
Approved

Figure 22-9. Using the DataType attribute to control the display of a DateTime value

---

■ **Tip** The change is more pronounced when the application is viewed using a Web browser that has better support for the HTML5 `input` element types.

Table 22-4 describes the most useful values of the DataType enumeration.

Table 22-4. The Values of the DataType Enumeration

Value	Description
`DateTime`	Displays a date and time (this is the default behavior for `System.DateTime` values)
`Date`	Displays the date portion of a `DateTime`
`Time`	Displays the time portion of a `DateTime`
`Text`	Displays a single line of text
`PhoneNumber`	Displays a phone number
`MultilineText`	Renders the value in a `textarea` element
`Password`	Displays the data so that individual characters are masked from view
`Url`	Displays the data as a URL (using an HTML a element)
`EmailAddress`	Displays the data as an e-mail address (using an a element with a `mailto href`)

The effect of these values depends on the type of the property they are associated with and the helper being used. For example, the MultilineText value will lead those helpers that create editors for properties to create an HTML textarea element but will be ignored by the display helpers. This makes sense. The textarea element allows the user to edit a value, which doesn't have a beating when displaying the data in a read-only form. Equally, the Url value has an effect only on the display helpers, which render an HTML a element to create a link.

Using Metadata to Select a Display Template

As their name suggests, templated helpers use display templates to generate HTML. The template that is used is based on the type of the property being processed and the kind of helper being used. I can use the UIHint attribute to specify the template used to render HTML for a property, as shown in Listing 22-15.

Listing 22-15. Using the UIHint Attribute in the Person.cshtml File

```
...
[DisplayName("New Person")]
public class Person {
    [HiddenInput(DisplayValue=false)]
    public int PersonId { get; set; }

    [Display(Name="First")]
    [UIHint("MultilineText")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    public Address HomeAddress { get; set; }

    [Display(Name="Approved")]
    public bool IsApproved { get; set; }
    public Role Role { get; set; }
}
...
```

In the listing, I specified the `MultilineText` template, which renders an HTML `textarea` element for the `FirstName` property when used with one of the editor helpers, such as `EditorFor` or `EditorForModel`. Table 22-5 shows the set of built-in templates that the MVC Framework includes.

Table 22-5. *The Built-In MVC Framework View Templates*

Value	Effect (Editor)	Effect (Display)
`Boolean`	Renders a checkbox for `bool` values. For nullable `bool?` values, a `select` element is created with options for `True`, `False`, and `Not Set`.	As for the editor helpers, but with the addition of the `disabled` attribute, which renders read-only HTML controls
`Collection`	Renders the appropriate template for each of the elements in an `IEnumerable` sequence. The items in the sequence do not have to be of the same type.	As for the editor helper
`Decimal`	Renders a single-line textbox `input` element and formats the data value to display two decimal places	Renders the data value formatted to two decimal places
`DateTime`	Renders an input element whose `type` attribute is `datetime` and which contains the complete date and time	Renders the complete value of a `DateTime` variable
`Date`	Renders an input element whose `type` attribute is `date` and that contains the date component (but not the time)	Renders the date component of a `DateTime` variable

(*continued*)

Table 22-5. (*continued*)

Value	Effect (Editor)	Effect (Display)
`EmailAddress`	Renders the value in a single-line textbox `input` element	Renders a link using an HTML `a` element and an `href` attribute that is formatted as a `mailto` URL
`HiddenInput`	Creates a hidden `input` element	Renders the data value and creates a hidden `input` element
`Html`	Renders the value in a single-line textbox `input` element	Renders a link using an HTML `a` element
`MultilineText`	Renders an HTML `textarea` element that contains the data value	Renders the data value
`Number`	Renders an `input` element whose `type` attribute is set to `number`	Renders the data value
`Object`	See explanation after this table	See explanation after this table
`Password`	Renders the value in a single-line textbox `input` element so that the characters are not displayed but can be edited	Renders the data value—the characters are not obscured
`String`	Renders the value in a single-line textbox `input` element	Renders the data value
`Text`	Identical to the `String` template	Identical to the `String` template
`Tel`	Renders an `input` element whose `type` attribute is set to `tel`	Renders the data value
`Time`	Renders an input element whose `type` attribute is `time` and which contains the time component (but not the date)	Renders the time component of a `DateTime` variable
`Url`	Renders the value in a single-line textbox `input` element	Renders a link using an HTML `a` element. The inner HTML and the `href` attribute are both set to the data value.

■ **Caution** Care must be taken when using the `UIHint` attribute. I will receive an exception if I select a template that cannot operate on the type of the property I have applied it to, for example, applying the `Boolean` template to a `string` property.

The `Object` template is a special case. It is the template used by the scaffolding helpers to generate HTML for a view model object. This template examines each of the properties of an object and selects the most suitable template for the property type. The `Object` template takes metadata such as the `UIHint` and `DataType` attributes into account.

Applying Metadata to a Buddy Class

It is not always possible to apply metadata to an entity model class. This is usually the case when the model classes are generated automatically, like sometimes with ORM tools such as the Entity Framework (although not the way I used Entity Framework in the SportsStore application). Any changes applied to automatically generated classes, such as applying attributes, will be lost the next time the classes are updated or regenerated.

The solution to this problem is to ensure that the model class is defined as `partial` and to create a second `partial` class that contains the metadata. Many tools that generate classes automatically create `partial` classes by default, including the Entity Framework. Listing 22-16 shows the `Person` class modified such that it could have been generated automatically. There is no metadata, and the class is defined as `partial`.

Listing 22-16. A Partial Model Class in the Person.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace HelperMethods.Models {

    [MetadataType(typeof(PersonMetaData))]
    public partial 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; }
    }

    // ...other types omitted from listing for brevity...
}
```

I tell the MVC Framework about the buddy class through the `MetadataType` attribute, which takes the type of the buddy class as its argument. Buddy classes must be defined in the same namespace and must also be `partial` classes. To demonstrate how this works, I have added a new folder to the example project called `Models/Metadata`. In this folder, I created a new class file called `PersonMetadata.cs`, the contents of which are shown in Listing 22-17.

Listing 22-17. The Contents of the PersonMetadata.cs File

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public partial class PersonMetaData {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }
    }
}
```

The buddy class only needs to contain properties to which metadata applies. I do not have to replicate all of the properties of the Person class, for example.

■ **Tip** Take particular care to change the namespace that Visual Studio adds to the new class file. The buddy class must be in the same namespace as the model class, which is HelperMethods.Models for the example project.

Working with Complex Type Properties

The template process relies on the Object template that I described in the previous section. Each property is inspected, and a template is selected to render HTML to represent the property and its data value.

You may have noticed that the HomeAddress property was not rendered as part of the Person class when I used the EditorForModel helper. This happens because the Object template operates only on *simple types*, which means those types that can be parsed from a string value using the GetConverter method of the System.ComponentModel.TypeDescriptor class. The supported types include the intrinsic C# types, such as int, bool, and double, and many common framework types, including Guid and DateTime.

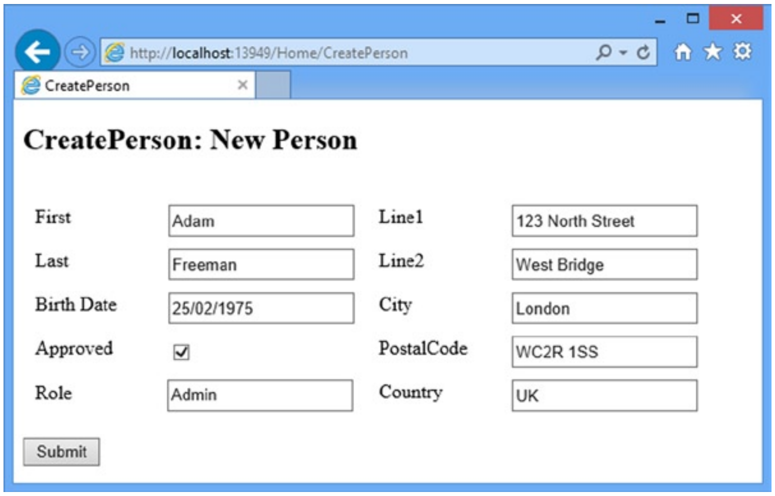
The result is that scaffolding helpers are not recursive. Given an object to process, a scaffolding templated helper method will generate HTML only for simple property types and will ignore any properties that are complex objects.

Although it can be inconvenient, this is a sensible policy. The MVC Framework does not know how the model objects are created. If the Object template was recursive, then it could easily end up triggering an ORM lazy-loading feature, which would lead it to read and render every object in the underlying database. If I want to render HTML for a complex property, I have to do it explicitly by making a separate call to a templated helper method. You can see how I have done this in Listing 22-18, which shows the changes I have made to the CreatePerson.cshtml view.

Listing 22-18. Dealing with a Property That Is a Complex Type in the CreatePerson.cshtml File

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
    Layout = "/Views/Shared/_Layout.cshtml";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson: @Html.LabelForModel()</h2>

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

    <div class="column">
        @Html.EditorForModel()
    </div>
    <div class="column">
        @Html.EditorFor(m => m.HomeAddress)
    </div>
    <input type="submit" value="Submit" />
}
```

To display the HomeAddress property, I added a call to the strongly typed EditorFor helper method. (I also added some div elements to provide structure to the HTML that is generated, relying on the CSS style I defined for the column class back in Listing 22-10.) You can see the result in Figure 22-10.



CreatePerson: New Person

First	Adam	Line1	123 North Street
Last	Freeman	Line2	West Bridge
Birth Date	25/02/1975	City	London
Approved	✓	PostalCode	WC2R 1SS
Role	Admin	Country	UK

Figure 22-10. *Displaying a complex property*

■ **Tip** The HomeAddress property is typed to return an Address object, and I can apply all of the same metadata to the Address class as I did to the Person class. The Object template is invoked explicitly when I use the EditorFor helpers on the HomeAddress property, and so all of the metadata conventions are honored.

Customizing the Templated View Helper System

I have shown you how to use metadata to shape the way that the templated helpers render data, but this is the MVC Framework and so there are some advanced options that completely customize the templated helpers. In the following sections, I will show you how to supplement or replace the built-in support to create specific results.

Creating a Custom Editor Template

One of the easiest ways of customizing the templated helpers is to create a custom template. This allows me to render exactly the HTML I want for a model property.

To demonstrate how this feature works, I am going to create a custom template for the Role property in the Person class. This property is typed to be a value from the Role enumeration, but the way that this is rendered by default is problematic because the templated helpers just create a regular input element that allows the user to enter any value and not just the values defined by the enumeration.

The MVC Framework looks for custom editor templates in the /Views/Shared/EditorTemplates folder, so I created this folder in the example project and then created a new strongly typed partial view called Role.cshtml within it. You can see the contents of this file in Listing 22-19.

Listing 22-19. The Contents of the Role.cshtml File

```
@model HelperMethods.Models.Role

@Html.DropDownListFor(m => m,
    new SelectList(Enum.GetNames(Model.GetType()), Model.ToString()))
```

The model type for this view is the Role enumeration and I use the Html.DropDownListFor helper method to create a select with option elements for the values in the enumeration. I passed an additional value to the SelectList constructor, which specifies the selected value, which I obtained from the view model object. The DropDownListFor method and the SelectList object operate on string values, so I have to make sure that I cast the enumeration values and the view model value.

When I use any of the templated helper methods to generate an editor for the Role type, my /Views/Shared/EditorTemplates/Role.cshtml file will be used, ensuring that I present the user with a consistent and usable representation of the data type. You can see the effect of the custom template in Figure 22-11.

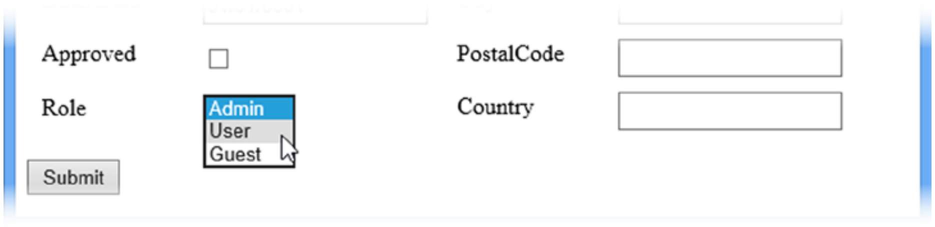


Figure 22-11. *The effect of a custom template for the Role enumeration*

UNDERSTANDING THE TEMPLATE SEARCH ORDER

The Role.cshtml template works because the MVC Framework looks for custom templates for a given C# type before it uses one of the built-in templates. In fact, there is a specific sequence that the MVC Framework follows to find a suitable template:

1. The template passed to the helper. For example, Html.EditorFor(m => m.SomeProperty, "MyTemplate") would lead to MyTemplate being used.
2. Any template that is specified by metadata attributes, such as UIHint
3. The template associated with any data type specified by metadata, such as the DataType attribute
4. Any template that corresponds to the.NET class name of the data type being processed
5. The built-in String template if the data type being processed is a simple type
6. Any template that corresponds to the base classes of the data type

7. If the data type implements `IEnumerable`, then the built-in `Collection` template will be used.

8. If all else fails, the `Object` template will be used, subject to the rule that scaffolding is not recursive.

Some of these steps rely on the built-in templates, which are described in Table 22-5. At each stage in the template search process, the MVC Framework looks for a template called `EditorTemplates/<name>` for editor helper methods or `DisplayTemplates/<name>` for display helper methods. For the `Role` template, I satisfied step 4 in the search process; I created a template called `Role.cshtml` and placed it in the `/Views/Shared/EditorTemplates` folder.

Custom templates are found using the same search pattern as regular views, which means I can create a controller-specific custom template and place it in the `~/Views/<controller>/EditorTemplates` folder to override the templates found in the `~/Views/Shared/EditorTemplates` folder.

Creating a Generic Template

I am not limited to creating type-specific templates. I can, for example, create a template that works for all enumerations and then specify that this template be selected using the `UIHint` attribute. If you look at the template search sequence in the "Understanding the Template Search Order" sidebar, you will see that templates specified using the `UIHint` attribute take precedence over type-specific ones.

To demonstrate how this works, I have created a new view file called `Enum.cshtml` in the `/Views/Shared/EditorTemplates` folder. The contents of this file are shown in Listing 22-20.

Listing 22-20. The Contents of the Enum.cshtml

```
@model Enum

@Html.DropDownListFor(m => m, Enum.GetValues(Model.GetType())
    .Cast<Enum>()
    .Select(m => {
        string enumVal = Enum.GetName(Model.GetType(), m);
        return new SelectListItem() {
            Selected = (Model.ToString() == enumVal),
            Text = enumVal,
            Value = enumVal
        };
    }))
```

The model type for this template is `Enum`, which allows me to work with any enumeration. For variety, I have used some LINQ to generate the strings that are required to create the `select` and `option` elements (although this is not a requirement for a generic template: I just like LINQ).

I can then apply the `UIHint` attribute. The example project defines a metadata buddy class, so I have applied the attribute to the `PersonMetadata` class, as shown in Listing 22-21. (As a reminder, you can find this class defined in the `/Models/Metadata/PersonMetadata.cs` file.)

Listing 22-21. Using the UIHint Attribute to Specify a Custom Template in the PersonMetadata.cs File

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models {

    [DisplayName("New Person")]
    public partial class PersonMetaData {
        [HiddenInput(DisplayValue=false)]
        public int PersonId { get; set; }

        [Display(Name="First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Display(Name="Approved")]
        public bool IsApproved { get; set; }

        [UIHint("Enum")]
        public Role Role { get; set; }
    }
}
```

This approach gives a more general solution that you can apply throughout an application to ensure that all Enum properties are displayed using a select element. I prefer to create model type-specific custom templates, but it can be more convenient to have one template that you can apply widely.

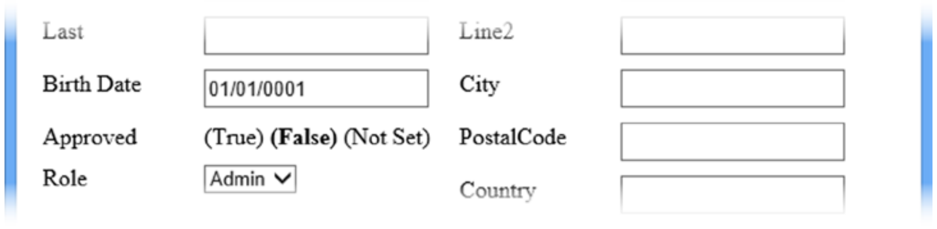
Replacing the Built-in Templates

If I create a custom template that has the same name as one of the built-in templates, the MVC Framework will use the custom version in preference to the built-in one. Listing 22-22 shows the contents of the Boolean.cshtml file that I added to the /Views/Shared/EditorTemplates folder. This view replaces the built-in Boolean template which is used to render bool and bool? values.

Listing 22-22. The Contents of the Boolean.cshtml File

```
@model bool?

@if (ViewData.ModelMetadata.IsNullableValueType && Model == null) {
    @:(True) (False) <b>(Not Set)</b>
} else if (Model.Value) {
    @:<b>(True)</b> (False) (Not Set)
} else {
    @:(True) <b>(False)</b> (Not Set)
}
```

In this view, I display all of the possible values and highlight the one that corresponds to the model object. You can see the effect of this template in Figure 22-12.



Last		Line2	
Birth Date	01/01/0001	City	
Approved	(True) **(False)** (Not Set)	PostalCode	
Role	Admin	Country	

Figure 22-12. *The effect of overriding a built-in editor template*

You can see the flexibility that custom templates offer, even if the example I have shown is not especially useful and does not let the user change the property value. As you have seen, there are a number of different ways that you can control how your model properties are displayed and edited, and you can pick the approach that suits your programming style and application best.

Summary

In this chapter, I have shown you the system of model templates that are accessible through the templated view helper methods. It can take a little while to set up the metadata and to create custom templates, but the result is closely tailored to your application and gives you complete flexibility in how your view model data is displayed and edited.