CHAPTER 24

Model Binding

Model binding is the process of creating .NET objects using the data sent by the browser in an HTTP request. I have been relying on the model binding process each time I have defined an action method that takes a parameter. The parameter objects are created through model binding from the data in the request. In this chapter, I'll show you how the model binding system works and demonstrate the techniques required to customize it for advanced use. Table 24-1 provides the summary for this chapter.

Table 24-1. *Chapter Summary*

Problem	Solution	Listing
Bind to a simple type or a collection	Add a parameter to an action method.	1–6, 21–27
Provide a fallback value for model binding	Use a nullable type for the action method parameter or use a default value.	7–8
Bind to a complex type	Ensure that the HTML generated by your views contains nested property values.	9–13
Override the default approach to locating nested complex types	Use the `Prefix` property `Bind` attribute applied to the action method parameter.	14–18
Selectively bind properties	Use the `Include` or `Exclude` properties of the `Bind` attribute, applied either to the action method parameter or to the model class.	19–20
Manually invoke model binding	Call the `UpdateModel` or `TryUpdateModel` methods.	28–32
Create a custom value provider	Implement the `IValueProvider` interface.	33–37
Create a custom model binder	Implement the `IModelBinder` interface.	38–40

Preparing the Example Project

I have created a new Visual Studio project called `MvcModels` using the `Empty` template option and checking the option to include the core MVC folders and references. I will be using the same model class that you have seen in previous chapters, so create a new class file called `Person.cs` in the `Models` folder and ensure that the contents match Listing 24-1.

Listing 24-1. The Contents of the Person.cs File

```
using System;

namespace MvcModels.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 have also defined a Home controller, as shown in Listing 24-2. This controller defines a collection of sample Person objects and defines the Index action, which allows me to select a single Person by the value of the PersonId property.

Listing 24-2. The Contents of the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private Person[] personData = {
            new Person {PersonId = 1, FirstName = "Adam", LastName = "Freeman",
                Role = Role.Admin},
            new Person {PersonId = 2, FirstName = "Jacqui", LastName = "Griffyth",
                Role = Role.User},
            new Person {PersonId = 3, FirstName = "John", LastName = "Smith",
                Role = Role.User},
            new Person {PersonId = 4, FirstName = "Anne", LastName = "Jones",
                Role = Role.Guest}
        };
        public ActionResult Index(int id) {
            Person dataItem = personData.Where(p => p.PersonId == id).First();
            return View(dataItem);
        }
    }
}
```

I have created a view file called `/Views/Home/Index.cshtml` to support the action method. You can see the contents of this file in Listing 24-3. I have used the templated display helper to show some of the property values of the `Person` view model.

Listing 24-3. The Contents of the /Views/Home/Index.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>
<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>
```

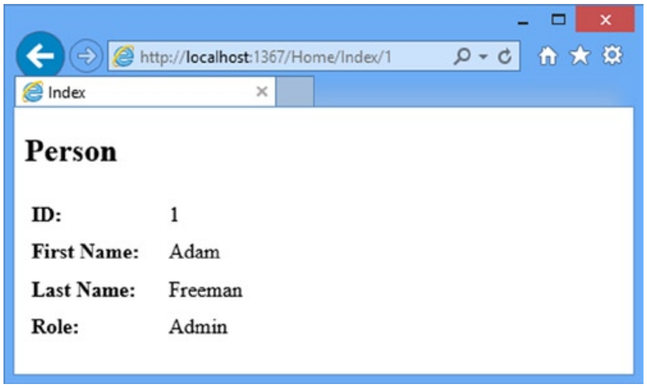
Finally, I created the `Views/Shared` folder and added a layout called `_Layout.cshtml` to it, the contents of which can be seen in Listing 24-4.

Listing 24-4. The Contents of the _Layout.cshtml File

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        label { display: inline-block; width: 100px; font-weight: bold; margin: 5px; }
        form label { float: left; }
        input.text-box { float: left; margin: 5px; }
        button[type=submit] { margin-top: 5px; float: left; clear: left; }
        form div { clear: both; }
    </style>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Understanding Model Binding

Model binding is an elegant bridge between the HTTP request and the C# methods that define actions. Most MVC Framework applications rely on model binding to some extent, including the simple example application that I created in the previous section. To see model binding at work, start the application and navigate to `/Home/Index/1`. The result is illustrated in Figure 24-1.



Person

ID: 1
First Name: Adam
Last Name: Freeman
Role: Admin

Figure 24-1. *A simple demonstration of model binding*

The URL contained the value of the `PersonId` property of the `Person` object I wanted to view, like this:

```
/Home/Index/1
```

The MVC Framework translated that part of the URL and used it as the argument when it called on the `Index` method in the `Home` controller class to service the request:

```
...
public ActionResult Index(int id) {
...
```

The process by which the URL segment was converted into the `int` method argument is an example of model binding. In the sections that follow, I show the process that this simple demonstration initiated, and then move on to explain some of the more complex model binding features. The process that leads to model binding begins when the request is received and processed by the routing engine. I have not changed the routing configuration for the example application, and the default route that Visual Studio adds to the `/App_Start/RouteConfig.cs` file was used to process the request. As a reminder, you can see the default route in Listing 24-5.

Listing 24-5. The Contents of 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 MvcModels {
    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 }
            );
        }
    }
}
```

I described how routes are defined and how they work in detail in Chapters 15 and 16, so I am not going to repeat that information here. For the model binding process, the important part is the `id` optional segment variable. When I navigated to the `/Home/Index/1` URL, the last segment of the URL, which specifies the `Person` object I am interested in, is assigned to the `id` routing variable.

The action invoker, which I introduced in Chapter 17, used the routing information to figure out that the `Index` action method was required to service the request, but it couldn't call the `Index` method until it had some useful values for the method argument.

The default action invoker, `ControllerActionInvoker`, (introduced in Chapter 17), relies on *model binders* to generate the data objects that are required to invoke the action. Model binders are defined by the `IModelBinder` interface, which is shown in Listing 24-6. I'll come back to this interface later in the chapter when I show you how to create a custom model binder.

Listing 24-6. The IModelBinder Interface from the MVC Framework

```
namespace System.Web.Mvc {

    public interface IModelBinder {
        object BindModel(ControllerContext controllerContext,
            ModelBindingContext bindingContext);
    }
}
```

There can be multiple model binders in an MVC application, and each binder can be responsible for binding one or more model types. When the action invoker needs to call an action method, it looks at the parameters that the method defines and finds the responsible model binder for the type of each one.

For the example in this section, the action invoker would examine the `Index` method and find that it has one `int` parameter. It would then locate the binder responsible for `int` values and call its `BindModel` method.

The model binder is responsible for providing an `int` value that can be used to call the `Index` method. This usually means transforming some element of the request data (such as form or query string values), but the MVC Framework doesn't put any limits on how the data is obtained.

I will show you some examples of custom binders later in this chapter. I will also show you some of the features of the `ModelBindingContext` class, which is passed to the `IModelBinder.BindModel` method.

Using the Default Model Binder

Although an application can define custom model binders, most just rely on the built-in binder class, `DefaultModelBinder`. This is the binder that is used by the action invoker when it cannot find a custom binder to bind the type. By default, this model binder searches four locations, shown in Table 24-2, for data matching the name of the parameter being bound.

Table 24-2. *The Order in Which the DefaultModelBinder Class Looks for Parameter Data*

Source	Description
`Request.Form`	Values provided by the user in HTML `form` elements
`RouteData.Values`	The values obtained using the application routes
`Request.QueryString`	Data included in the query string portion of the request URL
`Request.Files`	Files that have been uploaded as part of the request (see Chapter 12 for a demonstration of uploading files)

The locations are searched in order. For example, in my simple example, the `DefaultModelBinder` looks for a value for the `id` parameter as follows:

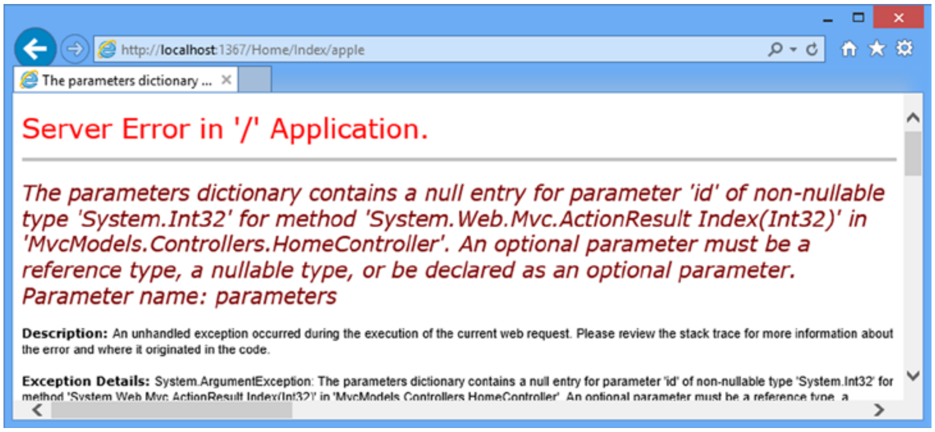
1. `Request.Form["id"]`
2. `RouteData.Values["id"]`
3. `Request.QueryString["id"]`
4. `Request.Files["id"]`

The search is stopped as soon as a value is found. In the example, the form data is searched without success, but a routing variable is found with the right name. This means that the query string and the names of uploaded files will not be searched at all.

■ **Tip** When replying on the default model binder, it is important that the parameters for your action method match the data property you are looking for. My example application works because the name of the action method parameter corresponds to the name of a routing variable. If I had named the action method parameter `personId`, for example, the default model binder would not have been able to locate a matching data value and my request would have failed.

Binding to Simple Types

When dealing with simple parameter types, the `DefaultModelBinder` tries to convert the string value, which has been obtained from the request data into the parameter type using the `System.ComponentModel.TypeDescriptor` class. If the value cannot be converted (for example, if I have supplied a value of `apple` for a parameter that requires an `int` value), then the `DefaultModelBinder` won't be able to bind to the model. You can see the problem that this creates by starting the example application and navigating to the URL `/Home/Index/apple`. Figure 24-2 illustrates the response from the server.


http://localhost:1367/Home/Index/apple

The parameters dictionary ...

Server Error in '/' Application.

The parameters dictionary contains a null entry for parameter 'id' of non-nullable type 'System.Int32' for method 'System.Web.Mvc.ActionResult Index(Int32)' in 'MvcModels.Controllers.HomeController'. An optional parameter must be a reference type, a nullable type, or be declared as an optional parameter.
Parameter name: parameters

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.ArgumentException: The parameters dictionary contains a null entry for parameter 'id' of non-nullable type 'System.Int32' for method 'System.Web.Mvc.ActionResult Index(Int32)' in 'MvcModels.Controllers.HomeController'. An optional parameter must be a reference type, a

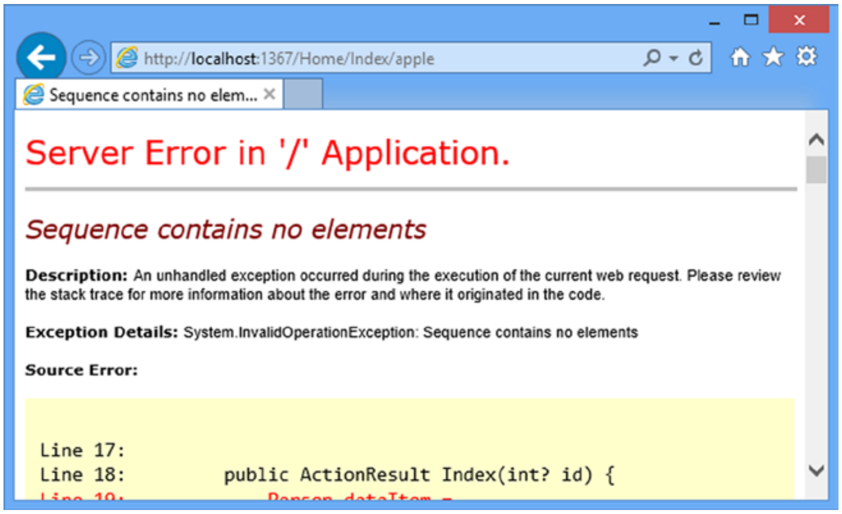
Figure 24-2. *An error processing a model property*

The default model binder is a little dogged. It sees that an `int` value is required and it tries to convert the value I provided in the URL, `apple`, into an `int`, which causes the error shown by the figure. I can make things easier for the model binder by using a `nullable` type, which provides a fallback position. Instead of requiring a numeric value, a nullable `int` parameter gives the model binder the choice of setting the action method argument to `null` when invoking the action. You can see how I have applied a nullable type to the `Index` action in Listing 24-7.

Listing 24-7. Using a Nullable Type for an Action Method Parameter in the HomeController.cs File

```
...
public ActionResult Index(int? id) {
    Person dataItem = personData.Where(p => p.PersonId == id).First();
    return View(dataItem);
}
...
```

If you run the application and navigate to `/Home/Index/apple`, you can see that I only changed, rather than solved, the problem, as shown by Figure 24-3.


http://localhost:1367/Home/Index/apple
Sequence contains no elem...

Server Error in '/' Application.

Sequence contains no elements

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.InvalidOperationException: Sequence contains no elements

Source Error:

```
Line 17:
Line 18:        public ActionResult Index(int? id) {
```

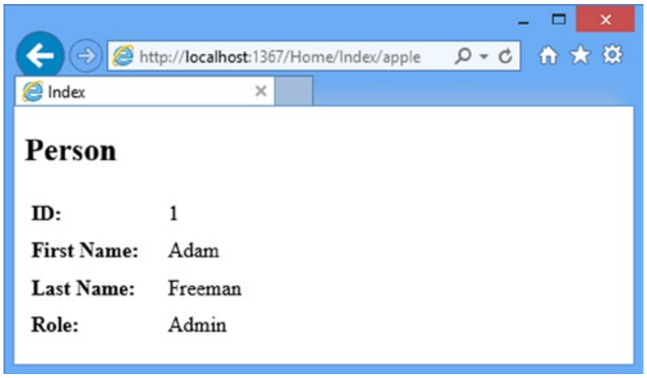
Figure 24-3. *A request for a null value*

The model binder is able to use `null` as the value for the `id` argument to the `Index` method, but the code inside the action method doesn't check for `null` values. I could fix that by explicitly checking for `null` values, but I can also set a default value for the parameter that will be used instead of `null`. You can see how I have applied a default parameter value to the `Index` action method in Listing 24-8.

Listing 24-8. Applying a Default Parameter Value in the HomeController.cs File

```
...
public ActionResult Index(int id = 1) {
    Person dataItem = personData.Where(p => p.PersonId == id).First();
    return View(dataItem);
}
...
```

Whenever the model binder is unable to find a value for the `id` parameter, the default value of `1` will be used instead, which has the effect of selecting the `Person` object whose `PersonId` property has a value of 1, as shown in Figure 24-4.



Person

ID:	1
First Name:	Adam
Last Name:	Freeman
Role:	Admin

Figure 24-4. *The effect of using a default parameter value in an action method*

■ **Tip** Bear in mind that I have solved the problem of non-numeric values for the model binder, but that I can still get `int` values for which there are no valid `Person` objects defined by the `Home` controller. For example, the model binder will happily convert the final segment of the URLs `/Home/Index/-1` and `/Home/Index/500` to `int` values. This will allow the action method to call the `Index` method with a real value, but will still result in an error because I don't perform any additional checks in the controller. I recommend you pay attention to the range of parameter values your action method may receive, and test accordingly.

CULTURE-SENSITIVE PARSING

The `DefaultModelBinder` class uses culture-specific settings to perform type conversions from different areas of the request data. The values that are obtained from URLs (the routing and query string data) are converted using culture-insensitive parsing, but values obtained from form data are converted taking culture into account.

The most common problem that this causes relates to `DateTime` values. Culture-insensitive dates are expected to be in the universal format `yyyy-mm-dd`. Form date values are expected to be in the format specified by the server. This means that a server set to the UK culture will expect dates to be in the form `dd-mm-yyyy`, whereas a server set to the US culture will expect the format `mm-dd-yyyy`, though in either case `yyyy-mm-dd` is acceptable, too.

A date value won't be converted if it isn't in the right format. This means that you must make sure that all dates included in the URL are expressed in the universal format. You must also be careful when processing date values that users provide. The default binder assumes that the user will express dates in the format of the server culture, something that is unlikely to always happen in an MVC application that has international users.

Binding to Complex Types

When the action method parameter is a complex type (i.e., any type which cannot be converted using the `TypeConverter` class), then the `DefaultModelBinder` class uses reflection to obtain the set of public properties and then binds to each of them in turn. To demonstrate how this works, I have added two new action methods to the `Home` controller, as shown in Listing 24-9.

Listing 24-9. Adding New Action Methods to the HomeController.cs File

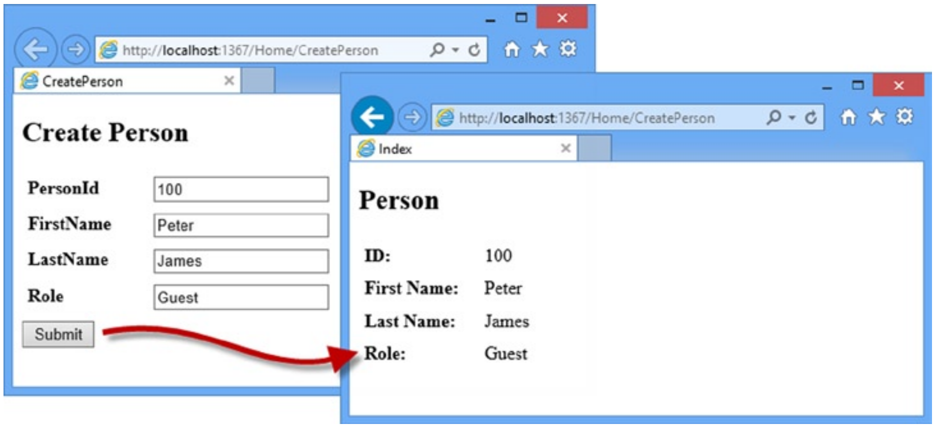
```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {
        private Person[] personData = {
            new Person {PersonId = 1, FirstName = "Adam", LastName = "Freeman",
                Role = Role.Admin},
            new Person {PersonId = 2, FirstName = "Jacqui", LastName = "Griffyth",
                Role = Role.User},
            new Person {PersonId = 3, FirstName = "John", LastName = "Smith",
                Role = Role.User},
            new Person {PersonId = 4, FirstName = "Anne", LastName = "Jones",
                Role = Role.Guest}
        };

        public ActionResult Index(int? id = 1) {
            Person dataItem = personData.Where(p => p.PersonId == id).First();
            return View(dataItem);
        }

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

        [HttpPost]
        public ActionResult CreatePerson(Person model) {
            return View("Index", model);
        }
    }
}
```

The CreatePerson overload without any parameters creates a new Person object and passes it to the view method, which has the effect of rendering the /Views/Home/CreatePerson.cshtml view, which I created to support the action method and the contents of which you can see in Listing 24-10.

Listing 24-10. The Contents of the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using (Html.BeginForm()) {
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m => m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m => m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m => m.Role)</div>
    <button type="submit">Submit</button>
}
```

This view renders a simple set of labels and editors for the properties of a Person object and contains a form element that posts the editor data back to the CreatePerson action method (the version decorated with the HttpPost attribute). This action method uses the /Views/Home/Index.cshtml view to display the data that the form contained. You can see how the new action methods work by starting the application and navigating to /Home/CreatePerson, as shown in Figure 24-5.



Create Person

PersonId	100
FirstName	Peter
LastName	James
Role	Guest

Submit

Person

ID:	100
First Name:	Peter
Last Name:	James
Role:	Guest

Figure 24-5. *Using the CreatePerson action methods*

I create a different kind of model binding situation when I post the form back to the `CreatePerson` method. The default model binder discovers that the action method requires a `Person` object and process each of the properties in turn. For each simple type property, the binder tries to locate a request value, just as it did in the previous example. So, for example, when it encounters the `PersonId` property, the binder will look for a `PersonId` data value, which it finds in the form data in the request.

If a property requires another complex type, then the process is repeated for the new type. The set of public properties are obtained and the binder tries to find values for all of them. The difference is that the property names are nested. For example, the `HomeAddress` property of the `Person` class is of the `Address` type, which is shown in Listing 24-11.

Listing 24-11. A Nested Model Class in the Person.cs File

```
...
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; }
}
...
```

When looking for a value for the `Line1` property, the model binder looks for a value for `HomeAddress.Line1`, as in the name of the property in the model object combined with the name of the property in the property type.

Creating Easily-Bound HTML

The use of prefixes means that I have to design views that take them into account, although the helper methods make this easy to do. In Listing 24-12, you can see how I have updated the `CreatePerson.cshtml` view file so that I capture some of the properties for the `Address` type.

Listing 24-12. Updating the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using(Html.BeginForm()) {
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m=>m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m=>m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m=>m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m=>m.Role)</div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m=> m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m=> m.HomeAddress.Country)
    </div>
    <button type="submit">Submit</button>
}
```

I have used the strongly typed `EditorFor` helper method, and specified the properties I want to edit from the `HomeAddress` property. The helper automatically sets the `name` attributes of the `input` elements to match the format that the default model binder uses, as follows:

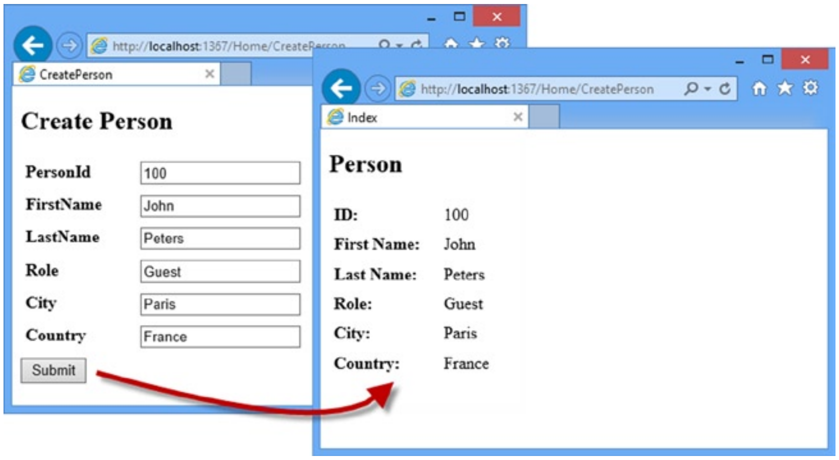
```
...
<input class="text-box single-line" id="HomeAddress_Country" name="HomeAddress.Country"
    type="text" value="" />
...
```

As a consequence of this feature, I don't have to take any special action to ensure that the model binder can create the `Address` object for the `HomeAddress` property. I can demonstrate this by editing the `/Views/Home/Index.cshtml` view to display the `HomeAddress` properties when they are submitted from the form, as shown in Listing 24-13.

Listing 24-13. Displaying the HomeAddress.City and HomeAddress.Country Properties in the Index.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>
<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>
<div><label>City:</label>@Html.DisplayFor(m => m.HomeAddress.City)</div>
<div><label>Country:</label>@Html.DisplayFor(m => m.HomeAddress.Country)</div>
```

If you start the application and navigate to the /Home/CreatePerson URL, you can enter values for the City and Country properties, and check that they are being bound to the model object by submitting the form, as shown in Figure 24-6.


http://localhost:1367/Home/CreatePerson
CreatePerson
Submit
http://localhost:1367/Home/CreatePerson
Index

Create Person

PersonId	100
FirstName	John
LastName	Peters
Role	Guest
City	Paris
Country	France

Person

ID:	100
First Name:	John
Last Name:	Peters
Role:	Guest
City:	Paris
Country:	France

Figure 24-6. *Binding to properties in complex objects*

Specifying Custom Prefixes

There are occasions when the HTML you generate relates to one type of object, but you want to bind it to another. This means that the prefixes containing the view won't correspond to the structure that the model binder is expecting and your data won't be properly processed. To demonstrate this situation, I have created a new class file called AddressSummary.cs in the Models folder. You can see the contents of this file in Listing 24-14.

Listing 24-14. The Contents of the AddressSummary.cs File

```
namespace MvcModels.Models {
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

I have added a new action method in the Home controller that uses the AddressSummary class, as shown in Listing 24-15.

Listing 24-15. Adding a New Action Method in the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;
```

```
namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...

        public ActionResult DisplaySummary(AddressSummary summary) {
            return View(summary);
        }
    }
}
```

The new action method is called `DisplaySummary`. It has an `AddressSummary` parameter, which it passes to the `View` method so that it can be displayed by the default view. I created the `DisplaySummary.cshtml` file in the `/Views/Home` folder and you can see the contents in Listing 24-16.

Listing 24-16. The Contents of the DisplaySummary.cshtml File

```
@model MvcModels.Models.AddressSummary
@{
    ViewBag.Title = "DisplaySummary";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Address Summary</h2>
<div><label>City:</label>@Html.DisplayFor(m => m.City)</div>
<div><label>Country:</label>@Html.DisplayFor(m => m.Country)</div>
```

This view displays the values of the two properties defined by the `AddressSummary` class. To demonstrate the problem with prefixes when binding to different model types, I will change the call to the `BeginForm` helper method in the `/Views/Home/CreatePerson.cshtml` file so that the form is submitted back to the new `DisplaySummary` action method, as shown in Listing 24-17.

Listing 24-17. Changing the Target of the Form in the CreatePerson.cshtml File

```
@model MvcModels.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create Person</h2>
@using(Html.BeginForm("DisplaySummary", "Home")) {
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m=>m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m=>m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m=>m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m=>m.Role)</div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m=> m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m=> m.HomeAddress.Country)
    </div>
    <button type="submit">Submit</button>
}
```

You can see the problem if you start the application and navigate to the /Home/CreatePerson URL. When you submit the form, the values that you entered for the City and Country properties are not displayed in the HTML generated by the DisplaySummary view.

The problem is that the name attributes in the form have the HomeAddress prefix, which is not what the model binder is looking for when it tries to bind the AddressSummary type. I can fix this by applying the Bind attribute to the action method parameter, which tells the binder which prefix to look for, as shown in Listing 24-18.

Listing 24-18. Applying the Bind Attribute in the HomeController.cs File

```
...
public ActionResult DisplaySummary([Bind(Prefix="HomeAddress")]AddressSummary summary) {
    return View(summary);
}
...
```

The syntax is a bit nasty, but the effect is useful. When populating the properties of the AddressSummary object, the model binder will look for HomeAddress.City and HomeAddress.Country data values in the request. In this example, I displayed editors for properties of the Person object, but used the model binder to create an instance of the AddressSummary class when the form data was posted, as shown in Figure 24-7. This may seem like a long setup for a simple problem, but the need to bind to a different kind of object is surprisingly common and you are likely to need this technique in your projects.

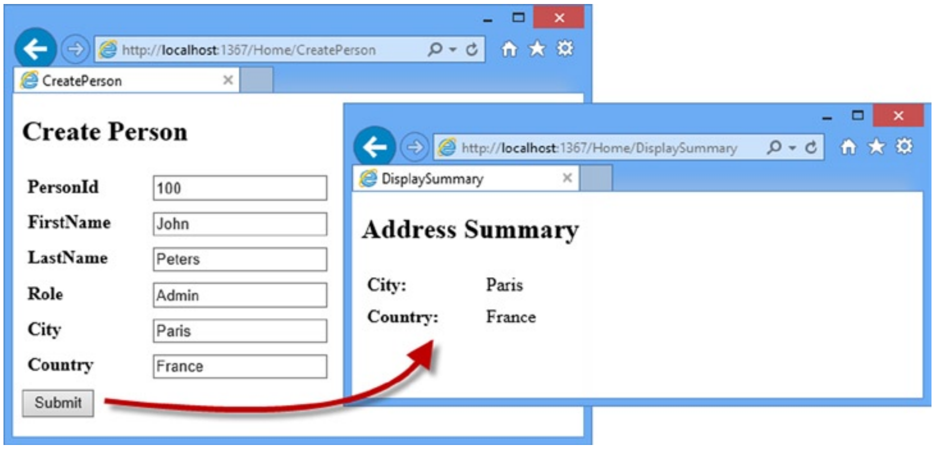


Figure 24-7. *Binding to the properties of a different object type*

Selectively Binding Properties

Imagine that the Country property of the AddressSummary class is especially sensitive and that I don't want the user to be able to specify values for it. The first thing I can do is prevent the user from seeing the property or even prevent the property from being included in the HTML sent to the browser, using the attributes I showed you in Chapter 22, or simply by not adding editors for that property to the view.

However, a nefarious user could simply edit the form data sent to the server when submitting the form data and pick the value for the Country property that suits them. What I really want to do is tell the model binder not to bind a value for the Country property from the request, which I can do by using the Bind attribute on the action method parameter. In Listing 24-19, you can see how I have used the attribute to prevent the user from providing a value for Country property in the DisplaySummary action method in the Home controller.

Listing 24-19. Excluding a Property from Model Binding in the HomeController.cs File

```
...
public ActionResult DisplaySummary(
    [Bind(Prefix="HomeAddress", Exclude="Country")]AddressSummary summary) {
    return View(summary);
}
...
```

The Exclude property of the Bind attribute allows you to exclude properties from the model binding process. You can see the effect by navigating to the /Home/CreatePerson URL, entering some data and submitting the form. You will see that there is no data displayed for the Country property. (As an alternative, you can use the Include property to specify only those properties that should be bound in the model; all other properties will be ignored.)

When the Bind attribute is applied to an action method parameter, it only affects instances of that class that are bound for that action method; all other action methods will continue to try and bind all the properties defined by the parameter type. If you want to create a more widespread effect, then you can apply the Bind attribute to the model class itself, as shown in Listing 24-20, where I have applied the Bind method to the AddressSummary class so that only the City property is included in the bind process.

Listing 24-20. Applying the Bind Attribute in the AddressSummary.cs File

```
using System.Web.Mvc;

namespace MvcModels.Models {
    [Bind(Include="City")]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

■ **Tip** When the Bind attribute is applied to the model class *and* to an action method parameter, a property will be included in the bind process only if neither application of the attribute excludes it. This means that the policy applied to the model class cannot be overridden by applying a less restrictive policy to the action method parameter.

Binding to Arrays and Collections

The default model binder includes some nice support for binding request data to arrays and collections. I demonstrate these features in the following sections, before moving on to show you how to customize the model binding process.

Binding to Arrays

One elegant feature of the default model binder is how it supports action method parameters that are arrays. To demonstrate this, I have added a new method to the Home controller called Names, which you can see in Listing 24-21.

Listing 24-21. Adding the Names Action Method in the HomeController.cs File

```
using System.Linq;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...

        public ActionResult Names(string[] names) {
            names = names ?? new string[0];
            return View(names);
        }
    }
}
```

The Names action method takes a string array parameter called names. The model binder will look for any data item that is called names and create an array that contains those values.

■ **Tip** Notice that I have to check to see if the parameter is null in the action method for this example. You can only use constant or literal values as defaults for parameters.

In Listing 24-22, you can see the /Views/Home/Names.cshtml view file, which I created to show array binding.

Listing 24-22. The Contents of the Names.cshtml File

```
@model string[]
@{
    ViewBag.Title = "Names";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Names</h2>
@if (Model.Length == 0) {
    using(Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <div><label>@(i + 1):</label>@Html.TextBox("names")</div>
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (string str in Model) {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}
```

This view displays different content based on the number of items there are in the view model. If there are no items, then I display a form that contains three identical input elements, like this:

```
...
<form action="/Home/Names" method="post">
    <div><label>1:</label><input id="names" name="names" type="text" value="" /></div>
    <div><label>2:</label><input id="names" name="names" type="text" value="" /></div>
    <div><label>3:</label><input id="names" name="names" type="text" value="" /></div>
    <button type="submit">Submit</button>
</form>
...
```

When I submit the form, the default model binder sees that the action method requires a string array and looks for data items that have the same name as the parameter. For this example, this means the contents of all of the input elements is gathered together to populate an array. You can see how the action method and view operate in Figure 24-8.

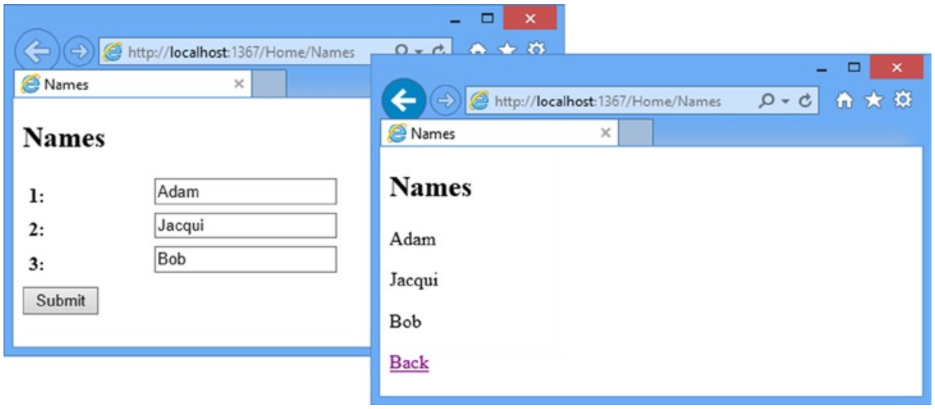

http://localhost:1367/Home/Names
Names
Names
1:
Adam
2:
Jacqui
3:
Bob
Submit
http://localhost:1367/Home/Names
Names
Names
Adam
Jacqui
Bob
Back

Figure 24-8. *Model binding for arrays*

Binding to Collections

It isn't just arrays that I can bind to. I can also use the .NET collection classes. In Listing 24-23, you can see how I have changed the type of the Names action method parameter to be a strongly typed List.

Listing 24-23. Using a Strongly Typed Collection in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...
        public ActionResult Names(IList<string> names) {
            names = names ?? new List<string>();
            return View(names);
        }
    }
}
```

Notice that I have used the `IList` interface. I didn't need to specify a concrete implementation class (although I could have if I preferred). In Listing 24-24, you can see how I have modified the `Names.cshtml` view file to use the new model type.

Listing 24-24. Using a Collection As the Model Type in the Names.cshtml File

```
@model IList<string>
@{
    ViewBag.Title = "Names";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Names</h2>
@if (Model.Count == 0) {
    using(Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <div><label>@(i + 1):</label>@Html.TextBox("names")</div>
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (string str in Model) {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}
```

The functionality of the `Names` action is unchanged, but I am now able to work with a collection class rather than an array.

Binding to Collections of Custom Model Types

I can also bind individual data properties to an array of custom types, such as the `AddressSummary` model class. In Listing 24-25, you can see that I have added a new action method to the `Home` controller called `Address`, which has a strongly typed collection parameter that relies on a custom model class.

Listing 24-25. Defining an Action Method in the HomeController.cs File

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

```
namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...

        public ActionResult Address(IList<AddressSummary> addresses) {
            addresses = addresses ?? new List<AddressSummary>();
            return View(addresses);
        }
    }
}
```

The view that I created for this action method is the `/Views/Home/Address.cshtml` file, which you can see in Listing 24-26.

Listing 24-26. The Contents of the Address.cshtml File

```
@using MvcModels.Models
@model IList<AddressSummary>
@{
    ViewBag.Title = "Address";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Addresses</h2>
@if (Model.Count() == 0) {
    using (Html.BeginForm()) {
        for (int i = 0; i < 3; i++) {
            <fieldset>
                <legend>Address @(i + 1)</legend>
                <div><label>City:</label>@Html.Editor("[" + i + "].City")</div>
                <div><label>Country:</label>@Html.Editor("[" + i + "].Country")</div>
            </fieldset>
        }
        <button type="submit">Submit</button>
    }
} else {
    foreach (AddressSummary str in Model) {
        <p>@str.City, @str.Country</p>
    }
    @Html.ActionLink("Back", "Address");
}
```

This view renders a `form` element if there are no items in the model collection. The `form` consists of pairs of `input` elements whose `name` attributes are prefixed with an array index, like this:

```
...
<fieldset>
    <legend>Address 1</legend>
    <div>
        <label>City:</label>
        <input class="text-box single-line" name="[0].City" type="text" value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[0].Country" type="text" value="" />
    </div>
</fieldset>
<fieldset>
    <legend>Address 2</legend>
    <div>
        <label>City:</label>
        <input class="text-box single-line" name="[1].City" type="text" value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[1].Country" type="text" value="" />
    </div>
</fieldset>
...
```

When the form is submitted, the default model binder realizes that it needs to create a collection of AddressSummary objects and uses the array index prefixes in the name attributes to obtain values for the object properties. The properties prefixed with [0] are used for the first AddressSummary object, those prefixed with [1] are used for the second object, and so on.

The Address.cshtml view defines input elements for three such indexed objects and displays them when the model collection contains items. Before I can demonstrate this, I need to remove the Bind attribute from the AddressSummary model class, as shown in Listing 24-27; otherwise, the model binder will ignore the Country property.

Listing 24-27. Removing the Bind Attribute from the AddressSummary.cs File

```
using System.Web.Mvc;

namespace MvcModels.Models {
    // This attribute has been commented out
    //[Bind(Include="City")]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

You can see how the binding process for custom object collections works by starting the application and navigating to the /Home/Address URL. Enter some cities and countries, and then click the Submit button to post the form to the server. The model binder will find and process the indexed data values and use them to create the collection of AddressSummary objects that are then passed back to the view and displayed to you, as shown in Figure 24-9.

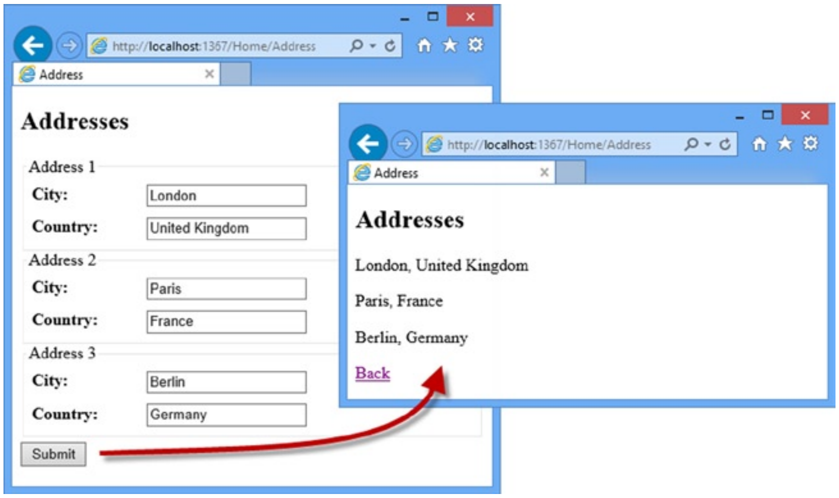


Figure 24-9. *Binding collections of custom objects*

Manually Invoking Model Binding

The model binding process is performed automatically when an action method defines parameters, but I can take direct control of the process if I want to. This gives more explicit control over how model objects are instantiated, where data values are obtained from, and how data parsing errors are handled. Listing 24-28 demonstrates how I have changed the `Address` action method in the `Home` controller to manually invoke the binding process.

Listing 24-28. Manually Invoking the Model Binding Process in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...

        public ActionResult Address() {
            IList<AddressSummary> addresses = new List<AddressSummary>();
            UpdateModel(addresses);
            return View(addresses);
        }
    }
}
```

The `UpdateModel` method takes a model object that I was previously defining as a parameter and tries to obtain values for its public properties using the standard binding process.

When I manually invoke the binding process, I am able to restrict the binding process to a single source of data. By default, the binder looks in four places: form data, route data, the query string, and any uploaded files. Listing 24-29 shows how to restrict the binder to searching for data in a single location—in this case, the form data.

Listing 24-29. Restricting the Binder to the Form Data in the HomeController.cs File

```
...
public ActionResult Address() {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, new FormValueProvider(ControllerContext));
    return View(addresses);
}
...
```

This version of the `UpdateModel` method takes an implementation of the `IValueProvider` interface, which becomes the sole source of data values for the binding process. Each of the four default data locations is represented by an `IValueProvider` implementation, as shown in Table 24-3.

Table 24-3. *The Built-in IValueProvider Implementations*

Source	`IValueProvider` Implementation
`Request.Form`	`FormValueProvider`
`RouteData.Values`	`RouteDataValueProvider`
`Request.QueryString`	`QueryStringValueProvider`
`Request.Files`	`HttpFileCollectionValueProvider`

Each of the classes listed in Table 24-3 takes a `ControllerContext` constructor parameter, which I obtain through the property called `ControllerContext` that is defined by the `Controller` class, as shown in the listing. The most common way of restricting the source of data is to look only at the form values. There is a neat binding trick that I can use so that I don't have to create an instance of `FormValueProvider`, as shown in Listing 24-30.

Listing 24-30. Restricting the Binder Data Source in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, formData);
    return View(addresses);
}
...
```

The `FormCollection` class implements the `IValueProvider` interface, and if I define the action method to take a parameter of this type, the model binder will provide me with an object that I can pass directly to the `UpdateModel` method.

■ **Tip** There are other overloaded versions of the `UpdateModel` method that specify a prefix to search for and which model properties should be included in the binding process.

Dealing with Binding Errors

Users will inevitably supply values that cannot be bound to the corresponding model properties—invalid dates or text for numeric values, for example. When I invoke model binding explicitly, I am responsible for dealing with any errors. The model binder expresses binding errors by throwing an `InvalidOperationException`. Details of the errors can be found through the `ModelState` feature, which I describe in Chapter 25. But when using the `UpdateModel` method, I must be prepared to catch the exception and use the `ModelState` to express an error message to the user, as shown in Listing 24-31.

Listing 24-31. Dealing with Model Binding Errors in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    try {
        UpdateModel(addresses, formData);
    } catch (InvalidOperationException ex) {
        // provide feedback to user
    }
    return View(addresses);
}
...
```

As an alternative approach, I can use the `TryUpdateModel` method, which returns `true` if the model binding process is successful and `false` if there are errors, as shown in Listing 24-32.

Listing 24-32. Using the TryUpdateModel Method in the HomeController.cs File

```
...
public ActionResult Address(FormCollection formData) {
    IList<AddressSummary> addresses = new List<AddressSummary>();
    if (TryUpdateModel(addresses, formData)) {
        // proceed as normal
    } else {
        // provide feedback to user
    }
    return View(addresses);
}
...
```

The only reason to favor `TryUpdateModel` over `UpdateModel` is if you don't like catching and dealing with exceptions. There is no functional difference in the model binding process.

■ **Tip** When model binding is invoked automatically, binding errors are not signaled with exceptions. Instead, you must check the result through the `ModelState.IsValid` property. I explain `ModelState` in Chapter 25.

Customizing the Model Binding System

I have shown you the default model binding process. As you might expect by now, there are some different ways in which the binding system can be customized. I show you some examples in the following sections.

Creating a Custom Value Provider

By defining a custom value provider, I can add my own source of data to the model binding process. Value providers implement the `IValueProvider` interface, which is shown in Listing 24-33.

Listing 24-33. The IValueProvider Interface from the MVC Framework

```
namespace System.Web.Mvc {
    public interface IValueProvider {

        bool ContainsPrefix(string prefix);

        ValueProviderResult GetValue(string key);
    }
}
```

The `ContainsPrefix` method is called by the model binder to determine if the value provider can resolve the data for a given prefix. The `GetValue` method returns a value for a given data key, or `null` if the provider doesn't have any suitable data.

I have added an `Infrastructure` folder to the example project and created a new class file called `CountryValueProvider.cs`, which I will use to provide values for the `Country` property. You can see the contents of this file in Listing 24-34.

Listing 24-34. The Contents of the CountryValueProvider.cs File

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

namespace MvcModels.Infrastructure {
    public class CountryValueProvider : IValueProvider {

        public bool ContainsPrefix(string prefix) {
            return prefix.ToLower().IndexOf("country") > -1;
        }

        public ValueProviderResult GetValue(string key) {
            if (ContainsPrefix(key)) {
                return new ValueProviderResult("USA", "USA",
                    CultureInfo.InvariantCulture);
            } else {
                return null;
            }
        }
    }
}
```

This value provider only responds to requests for values for the Country property and it always returns the value USA. For all other requests, I return null, indicating that I cannot provide data.

I have to return the data value as a ValueProviderResult class. This class has three constructor parameters. The first is the data item that I want to associate with the requested key. The second parameter is a version of the data value that is safe to display as part of an HTML page. The final parameter is the culture information that relates to the value; I have specified the InvariantCulture.

To register the value provider with the application, I need to create a factory class that will create instances of the provider when they are required by the MVC Framework. The factory class must be derived from the abstract ValueProviderFactory class. In Listing 24-35, you can see the contents of the CustomValueProviderFactory.cs class file that I added to the Infrastructure folder.

Listing 24-35. The Contents of the CustomValueProviderFactory.cs File

```
using System.Web.Mvc;

namespace MvcModels.Infrastructure {
    public class CustomValueProviderFactory : ValueProviderFactory {

        public override IValueProvider GetValueProvider(ControllerContext
                controllerContext) {
            return new CountryValueProvider();
        }
    }
}
```

The GetValueProvider method is called when the model binder wants to obtain values for the binding process. This implementation simply creates and returns an instance of the CountryValueProvider class, but you can use the data provided through the ControllerContext parameter to respond to different kinds of requests by creating different value providers.

I need to register the factory class with the application, which I do in the Application_Start method of Global.asax, as shown in Listing 24-36.

Listing 24-36. Registering a Value Provider Factory in the Global.asax File

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

namespace MvcModels {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ValueProviderFactories.Factories.Insert(0, new CustomValueProviderFactory());
        }
    }
}
```

I register the factory class by adding an instance to the static `ValueProviderFactories.Factories` collection. The model binder looks at the value providers in sequence, which means I have to use the `Insert` method to put the custom factory at the first position in the collection if I want to take precedence over the built-in providers.

If I want the custom provider to be a fallback that is used when the other providers cannot supply a data value, then I can use the `Add` method to append the factory class to the end of the collection, like this:

```
...
ValueProviderFactories.Factories.Add(new CustomValueProviderFactory());
...
```

I want the custom value provider to be used before any other provider, and so I used the `Insert` method. I need to modify the `Address` action method before I can test the value provider, so that the model binder doesn't just look at the form data for model property values. In Listing 24-37, you can see how I have removed the restriction on the source for values in the call to the `TryUpdateModel` method.

Listing 24-37. Removing the Restriction on the Sources of Model Property Values in the HomeController.cs File

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

namespace MvcModels.Controllers {
    public class HomeController : Controller {

        // ...other methods and statements omitted for brevity...

        public ActionResult Address() {
            IList<AddressSummary> addresses = new List<AddressSummary>();
            UpdateModel(addresses);
            return View(addresses);
        }
    }
}
```

You can see the custom value provider at work if you start the application and navigate to the `/Home/Address` URL. Enter city and country data, and then press the `Submit` button. You will see that the custom value provider, which has precedence over the built-in providers, has been used to generate values for the `Country` property in each of the `AddressSummary` objects that the model binder has created, as shown in Figure 24-10.

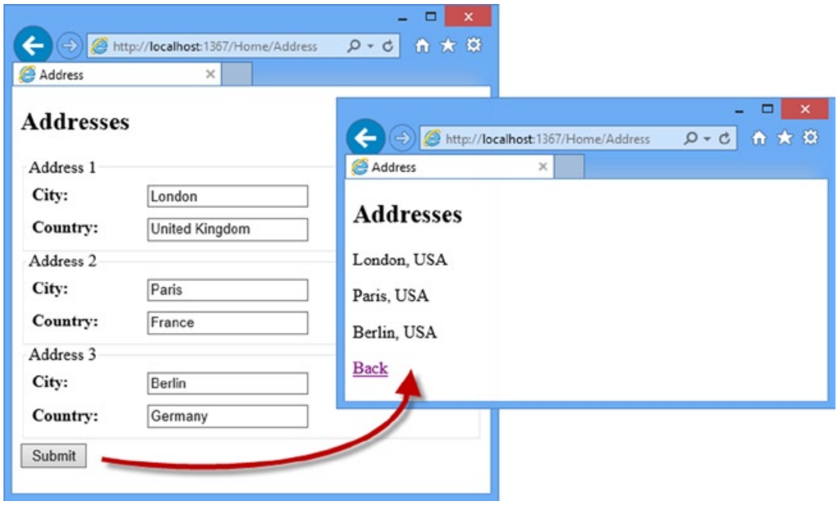


Figure 24-10. *The effect of the custom value provider*

Creating a Custom Model Binder

I can override the default binder's behavior by creating a custom model binder for a specific type. Custom model binders implement the `IModelBinder` interface, which I showed you earlier in the chapter. To demonstrate how to create a custom binder, I have added the `AddressSummaryBinder.cs` class file to the `Infrastructure` folder, the contents of which you can see in Listing 24-38.

Listing 24-38. The Contents of the AddressSummaryBinder.cs File

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

namespace MvcModels.Infrastructure {
    public class AddressSummaryBinder : IModelBinder {

        public object BindModel(ControllerContext controllerContext,
                ModelBindingContext bindingContext) {

            AddressSummary model = (AddressSummary)bindingContext.Model
                ?? new AddressSummary();
            model.City = GetValue(bindingContext, "City");
            model.Country = GetValue(bindingContext, "Country");
            return model;
        }

        private string GetValue(ModelBindingContext context, string name) {
            name = (context.ModelName == "" ? "" : context.ModelName + ".") + name;
            ValueProviderResult result = context.ValueProvider.GetValue(name);
            if (result == null || result.AttemptedValue == "") {
                return "<Not Specified>";
            } else {
                return (string)result.AttemptedValue;
            }
        }
    }
}
```

The MVC Framework will call the BindModel method when it wants an instance of the model type that the binder supports. I will show you how to register a model binder shortly, but the AddressSummaryBinder class will only be used to create instances of the AddressSummary class, which makes the code a lot simpler. (You can create custom binders that support multiple types, but I prefer one binder for each type.)

■ **Tip** I don't perform any input validation in this model binder, meaning that I blithely assume that the user has provided valid values for all of the Person properties. I discuss validation in Chapter 25, but for the moment, I want to focus on the basic model binding process.

The parameters to the BindModel method are a ControllerContext object that you can use to get details of the current request and a ModelBindingContext object, which provides details of the model object that is sought, as well as access to the rest of the model binding facilities in the MVC application. In Table 24-4, I have described the most useful properties defined by the ModelBindingContext class.

Table 24-4. *The Most Useful Properties Defined by the ModelBindingContext Class*

Property	Description
Model	Returns the model object passed to the UpdateModel method if binding has been invoked manually
ModelName	Returns the name of the model that is being bound
ModelType	Returns the type of the model that is being created
ValueProvider	Returns an IValueProvider implementation that can be used to get data values from the request

The custom model binder is simple. When the BindModel method is called, I check to see if the Model property of the ModelBindingContext object has been set. If it has, this is the object that I will generate data value for, and if not, then I create a new instance of the AddressSummary class. I get the values for the City and Country properties by calling the GetValue method and return the populated AddressSummary object.

In the GetValue method, I use the IValueProvider implementation obtained from the ModelBindingContext.ValueProvider property to get values for the model object properties.

The ModelName property tells me if there is a prefix I need to append to the property name I am looking for. You will recall that the action method is trying to create a collection of AddressSummary objects, which means that the individual input elements will have name attribute values that are prefixed [0] and [1]. The values I am looking for in the request will be [0].City, [0].Country, and so on. As a final step, I supply a default value of <Not Specified> if I can't find a value for a property or the property is the empty string (which is what is sent to the server when the user doesn't enter a value in the input elements in the form).

Registering the Custom Model Binder

I have to register the custom model binder so that the MVC application knows which types it can support. I do this in the Application_Start method of Global.asax, as demonstrated by Listing 24-39.

Listing 24-39. Registering a Custom Model Binder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using MvcModels.Infrastructure;
using MvcModels.Models;

namespace MvcModels {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // This statement has been commented out
            //ValueProviderFactories.Factories.Insert(0,
            //    new CustomValueProviderFactory());

            ModelBinders.Binders.Add(typeof(AddressSummary), new AddressSummaryBinder());
        }
    }
}
```

I register the binder through the ModelBinders.Binders.Add method, passing in the type that the binder supports and an instance of the binder class. Notice that I have removed the statement that registers the custom value provider. You can test the custom model binder by starting the application, navigating to the /Home/Address URL, and filling in only some of the form elements. When you submit the form, the custom model binder will use <Not Specified> for all of the properties for which you didn't enter a value, as shown in Figure 24-11.

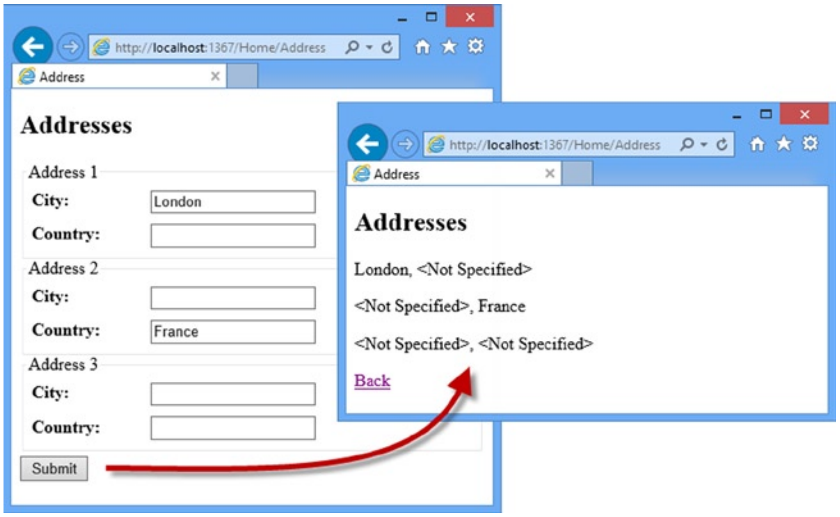


Figure 24-11. *The effect of using a custom model binder*

Registering a Model Binder with an Attribute

You can also register custom model binders by decorating the model class with the `ModelBinder` attribute, which means that you don't need to use the `Global.asax` file. In Listing 24-40, you can see how I have specified `AddressSummaryBinder` as the binder for the `AddressSummary` class.

Listing 24-40. Using the ModelBinder Attribute in the AddressSummary.cs File

```
using System.Web.Mvc;
using MvcModels.Infrastructure;

namespace MvcModels.Models {

    [ModelBinder(typeof(AddressSummaryBinder))]
    public class AddressSummary {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Summary

In this chapter, I introduced you to the workings of the model binding process, showing you how the default model binder operates and the different ways in which the process can be customized. Many MVC Framework applications will only need the default model binder, which works nicely to process the HTML that the helper methods generate. But for more advanced applications, it can be useful to use custom binders that create model objects in a more efficient or specific way. In the next chapter, I show you how to validate model objects and how to present the user with meaningful errors when invalid data is received.