CHAPTER 23

URL and Ajax Helper Methods

In this chapter, I am going to complete my coverage of the MVC Framework helper methods by showing you those methods that are able to generate URLs, links, and Ajax-enabled elements. Ajax is a key feature of any rich Web application and the MVC Framework includes some useful features that are based on the jQuery library. I'll show you how this works and demonstrate how you can use it to create Ajax-enabled forms and links. Table 23-1 provides the summary for this chapter.

Table 23-1. *Chapter Summary*

Problem	Solution	Listing
Generate links and URLs	Use the `Url.Content`, `Url.Action`, `Url.RouteUrl`, `Html.ActionLink`, `Html.RouteLink` helpers.	1–3
Submit form data via Ajax	Use the unobtrusive Ajax package and the `Ajax.BeginForm` helper.	4–10
Ensure that non-JavaScript browsers do not display HTML fragments	Set the `Url` Ajax option.	11
Provide the user with feedback during an Ajax request	Use the `LoadingElementId` and `LoadingElementDuration` Ajax options.	12
Prompt the user before making an Ajax request	Use the `Confirm` Ajax option.	13
Create an Ajax enabled link	Use the `Ajax.ActionLink` helper.	14, 15
Receive notifications about the progress and outcome of Ajax requests	Use the Ajax callback options.	16
Use JSON data in Ajax requests	Use the `JsonResult` action result.	17–19
Detect Ajax requests in the controller	Use the `Request.IsAjaxRequest` method.	20, 21

Note You will need to clear the browser history as you go from one example in this chapter to the next. This is just because I build features incrementally and isn't something you would need to worry about in a real project. I have added notes to remind you at key points in the chapter, but if you don't get the result you are expecting from an example then the first thing to try is clearing the history.

Preparing the Example Project

I am going to continue using the HelperMethods project that I created in Chapter 21 and added to in Chapter 22. For this chapter, I have created a new controller called People, shown in Listing 23-1. This controller defines a collection of Person model objects that I will use to demonstrate different helper features.

Listing 23-1. The Contents of the PeopleController.cs File

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        public ActionResult GetPeople() {
            return View(personData);
        }

        [HttpPost]
        public ActionResult GetPeople(string selectedRole) {
            if (selectedRole == null || selectedRole == "All") {
                return View(personData);
            } else {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                return View(personData.Where(p => p.Role == selected));
            }
        }
    }
}
```

I have not used any new techniques in this controller. The Index action method returns the default view. I will use the two GetPeople action methods to handle a simple form. The new features in this chapter appear in the views, which I create as I demonstrate different helper methods.

Defining Additional CSS Styles

I also need to add some new CSS styles to the project, which I have done in the Views/Shared/_Layout.cshtml file, as shown in Listing 23-2. I will define the elements the new styles apply to as I go through the chapter.

Listing 23-2. Adding Styles to 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; }
        table, td, th { border: thin solid black; border-collapse: collapse;
            padding: 5px; background-color: lemonchiffon;
            text-align: left; margin: 10px 0; }
        div.load { color: red; margin: 10px 0; font-weight: bold; }
        div.ajaxLink { margin-top: 10px; margin-right: 5px; float: left; }
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

Installing the NuGet Packages

The MVC Framework relies on the Microsoft *Unobtrusive Ajax* package to make and process Ajax requests. To install this package, select Package Manager Console from the Visual Studio `Tools` ➤ `Library Package Manager` menu and enter the following command:

```
Install-Package jQuery -version 1.10.2
Install-Package Microsoft.jQuery.Unobtrusive.Ajax -version 3.0.0
```

NuGet will install the package—and the jQuery library it depends on—into the project, creating a `Scripts` folder that contains a number of JavaScript files.

Creating Basic Links and URLs

One of the most fundamental tasks in a view is to create a link or URL that the user can follow to another part of the application. In previous chapters, you saw most of the helper methods that you can use to create links and URLs, but I want to take a moment to recap before moving on to some of the more advanced helpers that are available. Table 23-2 describes the available HTML helpers and it shows examples of each of them.

Table 23-2. *HTML Helpers That Render URLs*

Description	Example

Application-relative URL	`Url.Content("~/Content/Site.css")`
	Output:
	`/Content/Site.css`

Link to named action/controller	`Html.ActionLink("My Link", "Index", "Home")`
	Output:
	`<a href="/">My Link</a>`

URL for action	`Url.Action("GetPeople", "People")`
	Output:
	`/People/GetPeople`

URL using route data	`Url.RouteUrl(new {controller = "People", action="GetPeople"})`
	Output:
	`/People/GetPeople`

Link using route data	`Html.RouteLink("My Link", new {controller = "People",`
	`action="GetPeople"})`
	Output:
	`<a href="/People/GetPeople">My Link</a>`

Link to named route	`Html.RouteLink("My Link", "FormRoute", new {controller = "People",`
	`action="GetPeople"})`
	Output:
	`<a href="/app/forms/People/GetPeople">My Link</a>`

■ **Tip** As a reminder, the benefit of using these helpers to generate links and URLs is that the output is derived from the routing configuration, which means that a change in routes is automatically reflected in the links and URLs.

To demonstrate these helpers in action, I added an `Index.cshtml` view file to the `Views/People` folder, the contents of which you can see in Listing 23-3.

Listing 23-3. The Contents of the Index.cshtml File

```
@{
    ViewBag.Title = "Index";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Basic Links & URLs</h2>
<table>
    <thead><tr><th>Helper</th><th>Output</th></tr></thead>
    <tbody>
        <tr>
            <td>Url.Content("~/Content/Site.css")</td>
            <td>@Url.Content("~/Content/Site.css")</td>
        </tr>
```

```
        <tr>
            <td>Html.ActionLink("My Link", "Index", "Home")</td>
            <td>@Html.ActionLink("My Link", "Index", "Home")</td>
        </tr>
        <tr>
            <td>Url.Action("GetPeople", "People")</td>
            <td>@Url.Action("GetPeople", "People")</td>
        </tr>
        <tr>
            <td>Url.RouteUrl(new {controller = "People", action="GetPeople"})</td>
            <td>@Url.RouteUrl(new {controller = "People", action="GetPeople"})</td>
        </tr>
        <tr>
            <td>Html.RouteLink("My Link", new {controller = "People",
                action="GetPeople"})</td>
            <td>@Html.RouteLink("My Link", new {controller = "People",
                action="GetPeople"})</td>
        </tr>
        <tr>
            <td>Html.RouteLink("My Link", "FormRoute", new {controller = "People",
                action="GetPeople"})</td>
            <td>@Html.RouteLink("My Link", "FormRoute", new {controller = "People",
                action="GetPeople"})</td>
        </tr>
    </tbody>
</table>
```

This view contains the same set of helper calls that I listed in Table 23-2 and presents the results in an HTML table. You can see the effect by starting the application and navigating to the `/People/Index` URL, as shown in Figure 23-1. I have included this example because it makes it easy to experiment with routing changes and immediately see the effect.

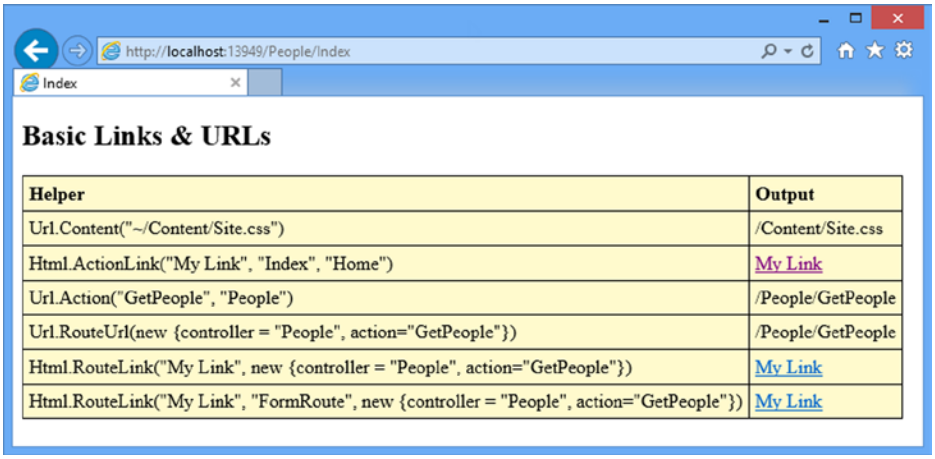

Figure 23-1. *Using helpers to create links and URLs*

Using MVC Unobtrusive Ajax

Ajax (or, if you prefer, AJAX) is shorthand for *Asynchronous JavaScript and XML*. The XML part is not as significant as it used to be, but the asynchronous part is what makes Ajax useful. It is a model for requesting data from the server in the background, without having to reload the Web page. The MVC Framework contains built-in support for *unobtrusive* Ajax, which means that you use helper methods to define your Ajax features, rather than having to add blocks of code throughout your views.

■ **Tip** The MVC Framework unobtrusive Ajax feature is based on jQuery. If you are familiar with the way that jQuery handles Ajax, then you will understand the MVC Ajax features.

Creating the Synchronous Form View

I am going to begin this section by creating the view for the `GetPeople` action in the controller, which I created as the `/Views/People/GetPeople.cshtml` file. You can see the contents of this file in Listing 23-4.

Listing 23-4. The Contents of the GetPeople.cshtml View File

```
@using HelperMethods.Models
@model IEnumerable<Person>
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody>
        @foreach (Person p in Model) {
            <tr>
                <td>@p.FirstName</td>
                <td>@p.LastName</td>
                <td>@p.Role</td>
            </tr>
        }
    </tbody>
</table>

@using (Html.BeginForm()) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

This is a strongly typed view whose model type is `IEnumerable<Person>`. I enumerate the `Person` objects in the model to create rows in an HTML table and use the `Html.BeginForm` helper to create a form that posts back to the action and controller that the view was generated by. The form contains a call to the `Html.DropDownList` helper, which I use to create a `select` element that contains `option` elements for each of the values defined by the `Role` enumeration, plus the value `All`. (I have used LINQ to create the list of values for the `option` elements by concatenating the values in the `enum` with an array that contains a single `All` string.)

The form contains a button that submits the form. The effect is that you can use the form to filter the `Person` objects that I defined in the controller in Listing 23-1, as shown in Figure 23-2. To test, start the application and navigate to the `/People/GetPeople` URL.

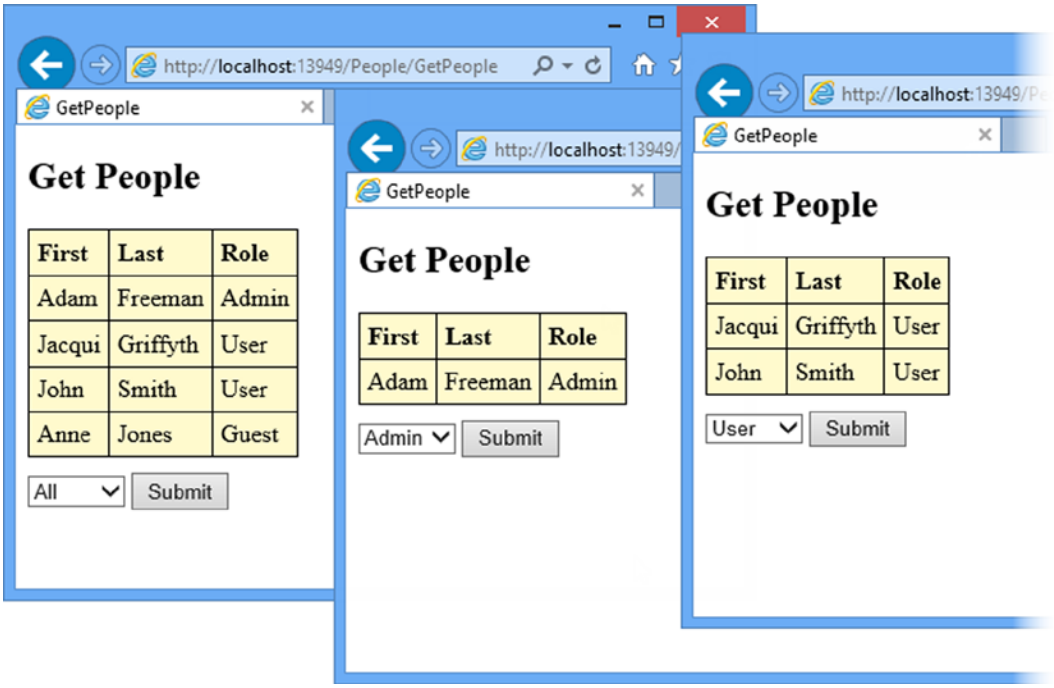

Figure 23-2. *A simple synchronous form*

This is a simple demonstration of a fundamental limitation in HTML forms, which is that the entire page is reloaded when the form is submitted. It means that the entire content of the Web page has to be regenerated and loaded from the server (which can be an expensive operation for complex views) and while this is happening, users cannot perform any other task with the application. They have to wait until the new page is generated, loaded, and then displayed by the browser.

For a simple application like this one, where the browser and server are running on the same machine, the delay is hardly noticeable. But for real applications over real internet connections, synchronous forms can make using a Web application frustrating for the user and expensive in terms of server bandwidth and processing power.

Preparing the Project for Unobtrusive Ajax

The unobtrusive Ajax feature is set up in two places in the application. First, in the `Web.config` file (the one in the root folder of the project) the `configuration/appSettings` element contains an entry for the `UnobtrusiveJavaScriptEnabled` property, which must be set to `true`, as shown in Listing 23-5. (This property is set to `true` by default when Visual Studio creates the project.)

Listing 23-5. Enabling the Unobtrusive Ajax Feature in the Web.config File

```
<?xml version="1.0" encoding="utf-8"?>

<configuration>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
  </system.web>
</configuration>
```

In addition to checking the `Web.config` setting, I need to add references to the jQuery JavaScript libraries that implement the unobtrusive Ajax functionality from the NuGet package I added at the start of the chapter. You can reference the libraries from individual views, but a more common approach is to do this in a layout file so that it affects all of the views that use that layout. In Listing 23-6, you can see how I have added references for two JavaScript libraries to the `/Views/Shared/_Layout.cshtml` file.

Listing 23-6. Adding References for the Unobtrusive Ajax JavaScript Libraries to 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; }
        table, td, th { border: thin solid black; border-collapse: collapse;
            padding: 5px; background-color: lemonchiffon;
            text-align: left; margin: 10px 0; }
        div.load { color: red; margin: 10px 0; font-weight: bold; }
        div.ajaxLink { margin-top: 10px; margin-right: 5px; float: left; }
    </style>
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
</head>
```

```
<body>
    @RenderBody()
</body>
</html>
```

The files that I have referenced with the `script` elements were added to the `Scripts` folder by the NuGet package. The `jquery-1.10.2.js` file contains the core jQuery library and the `jquery.unobtrusive-ajax.js` file contains the Ajax functionality (which relies on the main jQuery library).

Creating an Unobtrusive Ajax Form

I am now ready to start applying unobtrusive Ajax features to the example application, starting with an unobtrusive Ajax form. In the sections that follow, I go through the process of replacing a regular synchronous form with an Ajax equivalent and explain how the unobtrusive Ajax feature works.

Preparing the Controller

The goal is that only the data in the HTML `table` element is replaced when the user clicks on the `Submit` button in the example application. That means that the first thing that I need to do is refactor the action methods in the `People` controller so that I can get just the data I want through a child action. You can see the changes I have made to the `People` controller in Listing 23-7.

Listing 23-7. Refactoring the Action Methods in the PeopleController.cs File

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        public PartialViewResult GetPeopleData(string selectedRole = "All") {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return PartialView(data);
        }
```

```
        public ActionResult GetPeople(string selectedRole = "All") {
            return View((object)selectedRole);
        }
    }
}
```

I have added a `GetPeopleData` action that selects the `Person` objects that I need to display, and passes them to the `PartialView` method to generate the table rows that are required. Because the selection of the data is handled in the `GetPeopleData` action method, I have been able to simplify the `GetPeople` action method and remove the `HttpPost` version entirely. The purpose of this method is to pass the selected role as a `string` to its view.

I created a new partial view file, `/Views/People/GetPeopleData.cshtml`, for the new `GetPeopleData` action method. You can see the contents of the view in Listing 23-8. This view is responsible for generating the `tr` elements that will populate the table using the enumeration of `Person` objects that are passed from the action method.

Listing 23-8. The Contents of the GetPeopleData.cshtml File

```
@using HelperMethods.Models
@model IEnumerable<Person>

@foreach (Person p in Model) {
    <tr>
        <td>@p.FirstName</td>
        <td>@p.LastName</td>
        <td>@p.Role</td>
    </tr>
}
```

I also had to update the `/Views/People/GetPeople.cshtml` view, which you can see in Listing 23-9.

Listing 23-9. Updating the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody>
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Html.BeginForm()) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

I have changed the view model type to `string`, which I pass to the `Html.Action` helper method to invoke the `GetPeopleData` child action. This renders the partial view and generates the table rows.

Creating the Ajax Form

I still have a synchronous form in the application after these changes, but I have separated out the functionality in the controller so that I can request just the table rows through the `GetPeopleData` action. This new action method will be the target of the Ajax request and the next step is to update the `GetPeople.cshtml` view so that posting the form is handled through Ajax, as shown in Listing 23-10.

Listing 23-10. Creating an Unobtrusive Ajax Form in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody"
    };
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm("GetPeopleData", ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

At the heart of the MVC Framework support for Ajax forms is the `Ajax.BeginForm` helper method, which takes an `AjaxOptions` object as its argument. I like to create the `AjaxOptions` objects at the start of the view in a Razor code block, but you can create them inline when you call `Ajax.BeginForm` if you prefer.

The `AjaxOptions` class, which is in the `System.Web.Mvc.Ajax` namespace, defines properties that let me configure how the asynchronous request to the server is made and what happens to the data that comes back. These properties are described in Table 23-3.

Table 23-3. *AjaxOptions Properties*

Property	Description
`Confirm`	Sets a message to be displayed to the user in a confirmation window before making the Ajax request
`HttpMethod`	Sets the HTTP method that will be used to make the request—must be either `Get` or `Post`
`InsertionMode`	Specifies the way in which the content retrieved from the server is inserted into the HTML. The three choices are expressed as values from the `InsertionMode` enum: `InsertAfter`, `InsertBefore` and `Replace` (which is the default).
`LoadingElementId`	Specifies the ID of an HTML element that will be displayed while the Ajax request is being performed
`LoadingElementDuration`	Specifies the duration of the animation used to reveal the element specified by `LoadingElementId`
`UpdateTargetId`	Sets the ID of the HTML element into which the content retrieved from the server will be inserted
`Url`	Sets the URL that will be requested from the server

In the listing, I have set the `UpdateTargetId` property to `tableBody`. This is the `id` I assigned to the `tbody` HTML element in the view in Listing 23-10. When the user clicks the `Submit` button, an asynchronous request will be made to the `GetPeopleData` action method and the HTML fragment that is returned is used to replace the existing elements in the `tbody`.

■ **Tip** The `AjaxOptions` class also defines properties that specify callbacks for different stages in the request life cycle. See the "Working with Ajax Callbacks" section later in this chapter for details.

That's all there is to it: I replace the `Html.BeginForm` method with `Ajax.BeginForm` and ensure that I have a target for the new content. Everything else happens automatically and the result is an asynchronous form.

It can be hard to detect when you are testing with the browser and the server on the same machine, but you can tell that the browser is making Ajax requests for fragments of HTML by using the browser F12 tools. These tools allow you to monitor the network requests that the browser makes and in Figure 23-3, you can see the Internet Explorer tools showing a call to the `GetPeopleData` action method.

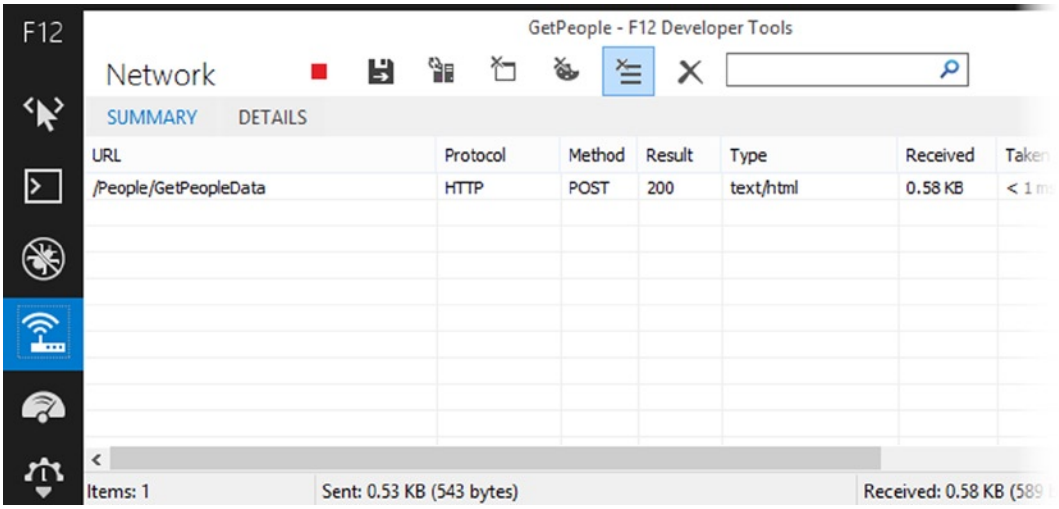

Figure 23-3. *Confirming that Ajax requests are being made*

Understanding How Unobtrusive Ajax Works

When I call the `Ajax.BeginForm` helper method, the options that I specify using the `AjaxOptions` object are transformed into attributes applied to the `form` element. The view in Listing 23-10 produces the following `form` element:

```
...
<form action="/People/GetPeopleData" data-ajax="true" data-ajax-mode="replace"
    data-ajax-update="#tableBody" id="form0" method="post">
...
```

When the HTML page rendered from the `GetPeople.cshtml` view is loaded by the browser, the JavaScript in the `jquery.unobtrusive-ajax.js` library scans the HTML elements and identifies the Ajax form by looking for elements that have a `data-ajax` attribute with a value of `true`.

The other attributes whose names start with `data-ajax` contain the values I specified using the `AjaxOptions` class. These configuration options are used to configure jQuery, which has built-in support for managing Ajax requests.

■ **Tip** You don't have to use the MVC Framework support for unobtrusive Ajax. There are plenty of alternatives available, including using jQuery directly. That said, pick a technique and stick to it. I recommend against mixing the MVC Framework unobtrusive Ajax support with other techniques and libraries in the same view, as there can be some unfortunate interactions, such as duplicated or dropped Ajax requests.

Setting Ajax Options

I can fine-tune the behavior of the Ajax requests by setting values for the properties of the `AjaxOptions` object that I pass to the `Ajax.BeginForm` helper method. In the following sections, I explain what each of these options does and why they can be useful.

Ensuring Graceful Degradation

When I set up the Ajax-enabled form in Listing 23-10, I passed in the name of the action method that I wanted to be called asynchronously. In the example, this was the `GetPeopleData` action, which generates a partial view containing a fragment of HTML.

One problem with this approach is that it doesn't work well if the user has disabled JavaScript (or is using a browser that doesn't support it). In such cases, when the user submits the form, the browser display discards the current HTML page and replaces it with the fragment returned by the target action method. The effect can be seen in Figure 23-4.

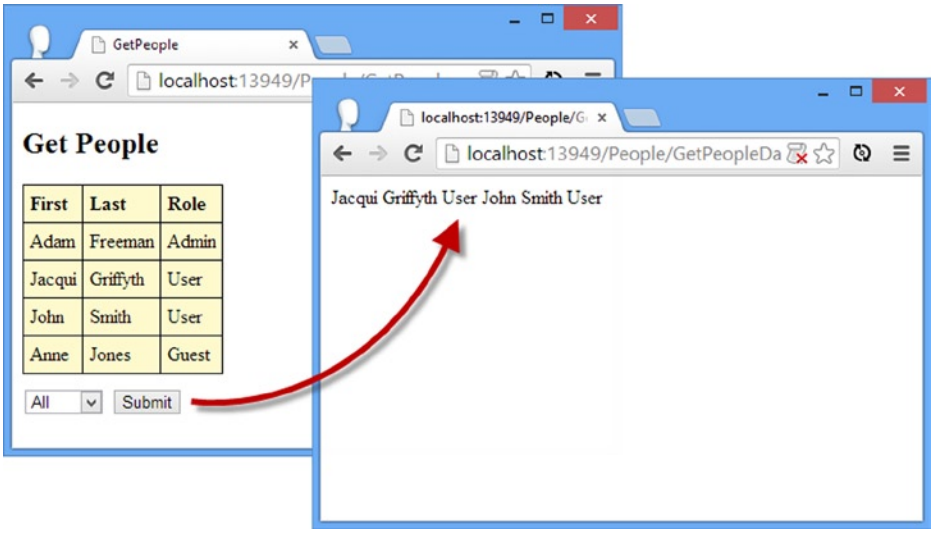

Figure 23-4. *The effect of using the Ajax.BeginForm helper without browser JavaScript support*

■ **Note** I used Google Chrome for this figure because it makes it easy to toggle JavaScript.

The simplest way to address this problem is to use the `AjaxOptions.Url` property to specify the target URL for the asynchronous request rather than specifying the action name as an argument to the `Ajax.BeginForm` method, as shown in Listing 23-11.

Listing 23-11. Ensuring Gracefully Degrading Forms in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData")
    };

}
```

```
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

I have used the `Url.Action` helper method to create a URL that will invoke the `GetPeopleData` action, and used the version of the `Ajax.BeginForm` method that takes only an `AjaxOptions` parameter. This has the effect of creating a `form` element that posts back to the originating action method if JavaScript isn't enabled, like this:

```
...
<form action="/People/GetPeople" data-ajax="true" data-ajax-mode="replace"
    data-ajax-update="#tableBody" data-ajax-url="/People/GetPeopleData" id="form0"
    method="post">
...
```

If JavaScript is enabled, then the unobtrusive Ajax library will take the target URL from the `data-ajax-url` attribute, which refers to the child action. If JavaScript is disabled, then the browser will use the regular form posting technique, which takes the target URL from the `action` attribute, which points back at the action method that will generate a complete HTML page.

■ **Caution** You might be wondering why I am making such a big deal about users who have disabled JavaScript. After all, who does that? In fact, it is prevalent in two groups of users. The first group consists of those users who take their IT security seriously and disable anything that could be used as the basis for an attack, something that JavaScript has been known for over the years. The second group is users in large corporations, which apply incredibly restrictive policies in the name of IT security (although, in my experience, corporate PCs are so poorly set up that security is nonexistent and the restrictions just annoy the users). You can ignore graceful degradation if you feel you can ignore IT security experts and people who work for big companies. But, since these can be affluent and tech-savvy users, I always take the time to make sure I support them.

Providing the User with Feedback While Making an Ajax Request

One drawback of using Ajax is that it isn't obvious to the user that something is happening, because the request to the server is made in the background. I can inform the user that a request is being performed by using the `AjaxOptions.LoadingElementId` and `AjaxOptions.LoadingElementDuration` properties. Listing 23-12 shows how I have applied these properties in the `GetPeople.cshtml` view file.

Listing 23-12. Giving Feedback to the User in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000
    };

}
<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

The `AjaxOptions.LoadingElementId` property specifies the `id` attribute value of a hidden HTML element that will be shown to the user while an Ajax request is performed. To demonstrate this feature, I added a `div` element to the view that I hid from the user by setting the CSS `display` property to `none`. I gave this `div` element an `id` attribute of `loading` and used this `id` as the value for the `LoadingElementId` property and the unobtrusive Ajax feature will display the element to the user for the duration of the request, as shown in Figure 23-5. The `LoadingElementDuration` property specifies the duration of the animation that is used to reveal the `loading` element to the user. I specified a value of `1000`, which denotes one second.

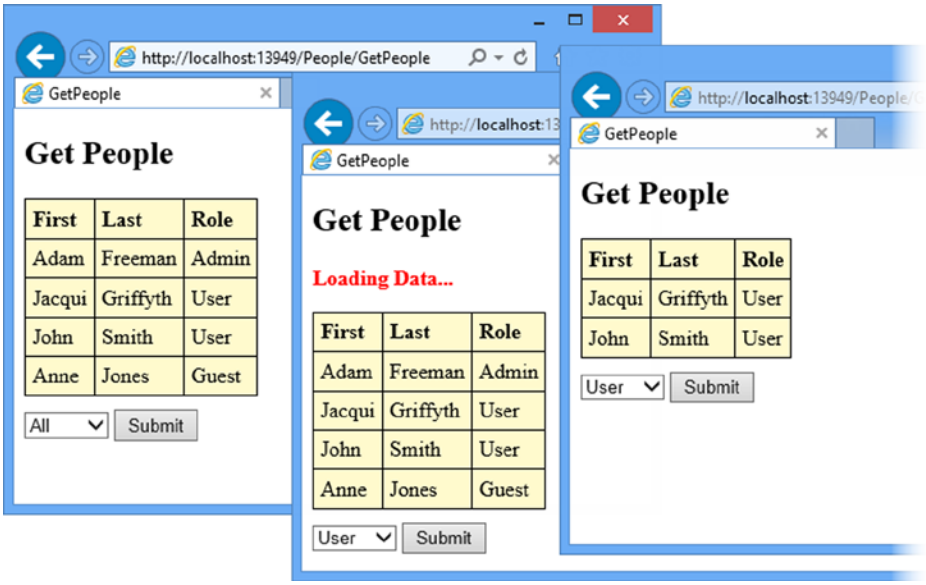

Figure 23-5. *Providing the user with feedback during an Ajax request*

Prompting the User Before Making a Request

The `AjaxOptions.Confirm` property lets me specify a message that will be used to prompt the user before each asynchronous request. The user can elect to proceed with or cancel the request. Listing 23-13 shows how I have applied this property to the `GetPeople.cshtml` file.

Listing 23-13. Promoting the User Before Making an Asynchronous Request in the GetPeople.cshtml File

```
...
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
...
```

With this addition, the user is prompted each time they submit the form, as shown in Figure 23-6. The user is prompted for *every* request, which means that this feature should be used sparingly to avoid irritating the user.

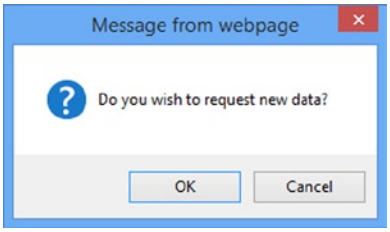

Figure 23-6. *Prompting the user before making a request*

Creating Ajax Links

In addition to forms, unobtrusive Ajax can be used to create a elements that will be followed asynchronously. The mechanism for this is similar to the way that Ajax forms work. You can see how I have added Ajax links to the `GetPeople.cshtml` view in Listing 23-14.

Listing 23-14. Creating Ajax-Enabled Links in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

```
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeopleData",
                new {selectedRole = role},
                new AjaxOptions {UpdateTargetId = "tableBody"})
        </div>
    }
</div>
```

I have used a `foreach` loop to call the `Ajax.ActionLink` helper for each of the values defined by the `Role` enumeration, creating a set of Ajax-enabled a elements. The a elements that are produced have the same kind of `data` attributes you saw when working with forms, like this:

```
...
<a data-ajax="true" data-ajax-mode="replace" data-ajax-update="#tableBody"
    href="/People/GetPeopleData?selectedRole=Guest">Guest</a>
...
```

The routing configuration in the project does not have an entry for the `selectedRole` variable, so the URL that has been generated for the `href` attribute specifies the role that the link represents using the query string component of the URL.

You can see the links I added to the view in Figure 23-7. Clicking one of these links will call the `GetPeopleData` action method and replace the contents of the `tbody` element with the HTML fragment that is returned. This creates the same effect of filtering the data that I achieved using the Ajax-enabled form earlier in the chapter.

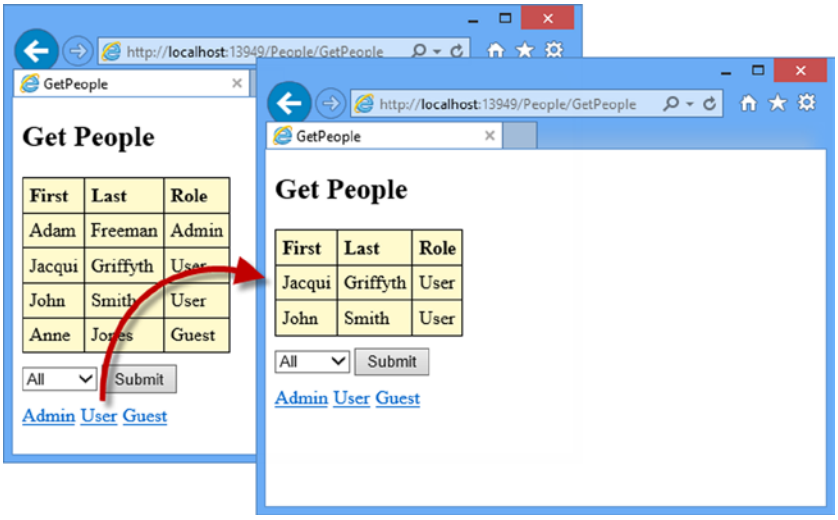

Figure 23-7. *Adding Ajax-enabled links to a view*

■ **Tip** You may have to clear your browser history to see the changes for this example.

Ensuring Graceful Degradation for Links

I face the same problem with the Ajax-enabled links as I did with the forms. When there is no JavaScript support on the browser, clicking one of the links will just display the HTML fragment that the `GetPeopleData` action method generates.

I can address this using the `AjaxOptions.Url` property to specify the URL for the Ajax request. For this example, I have specified the `GetPeople` action to the `Ajax.ActionLink` helper method, as shown in Listing 23-15.

Listing 23-15. Creating Graceful Ajax-Enabled Links in the GetPeople.cshtml File

```
...
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                    UpdateTargetId = "tableBody",
                    Url = Url.Action("GetPeopleData", new {selectedRole = role})
                })
        </div>
    }
</div>
...
```

This is why I created a new `AjaxOptions` object for each of the links rather than using the one I created in the Razor code block for the `form` element. Independent `AjaxOptions` allow me to specify a different value for the `Url` property for each link and support graceful degradation for non-JavaScript browsers.

Working with Ajax Callbacks

The `AjaxOptions` class defines a set of properties that specify JavaScript functions that will be called at various points in the Ajax request life cycle. These properties are described in Table 23-4.

Table 23-4. *AjaxOptions Callback Properties*

Property	jQuery Event	Description
`OnBegin`	`beforeSend`	Called immediately prior to the request being sent
`OnComplete`	`complete`	Called if the request is successful
`OnFailure`	`error`	Called if the request fails
`OnSuccess`	`success`	Called when the request has completed, irrespective of whether the request succeeded or failed

Each of the `AjaxOptions` callback properties correlates to an Ajax event supported by the jQuery library. I have listed the jQuery events in Table 23-4 for those readers who have used jQuery before. You can get details on each of these events and the parameters that will be passed to your functions at `http://api.jquery.com/jQuery.ajax` or in my book, *Pro jQuery 2.0*, also published by Apress.

In Listing 23-16, you can see how I have used a `script` element to define some basic JavaScript functions that will report on the progress of the Ajax request and use the properties shown in Table 23-4 to specify the functions as handlers for the Ajax events.

Listing 23-16. Using the Ajax Callbacks in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}

<script type="text/javascript">
    function OnBegin() {
        alert("This is the OnBegin Callback");
    }

    function OnSuccess(data) {
        alert("This is the OnSuccessCallback: " + data);
    }

    function OnFailure(request, error) {
        alert("This is the OnFailure Callback:" + error);
    }

    function OnComplete(request, status) {
        alert("This is the OnComplete Callback: " + status);
    }
</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>
```

```
@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}

<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                    UpdateTargetId = "tableBody",
                    Url = Url.Action("GetPeopleData", new {selectedRole = role}),
                    OnBegin = "OnBegin",
                    OnFailure = "OnFailure",
                    OnSuccess = "OnSuccess",
                    OnComplete = "OnComplete"
                })
        </div>
    }
</div>
```

I have defined four functions, one for each of the callbacks. For this example, I have kept things simple and simply display a message to the user in each of the functions. With these changes, clicking one of the links will display a sequence of alters that report on the progress of the Ajax request, as shown in Figure 23-8.

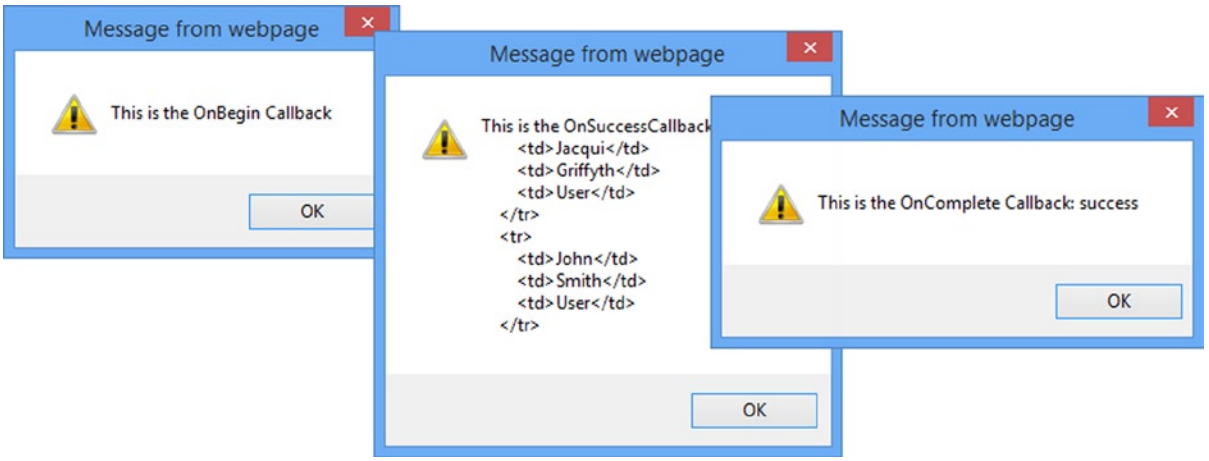

Figure 23-8. *The series of dialog boxes shown in response to the Ajax callbacks*

Displaying dialog boxes to the user for each callback isn't the most useful thing to do with the Ajax callbacks, but it demonstrates the sequence in which they are called. These JavaScript functions can be used for any purpose: manipulate the HTML DOM, trigger additional requests, and so forth. One of the most useful things to do with the callbacks is handle JSON data, which I describe in the next section.

Working with JSON

In the Ajax examples so far, the server has rendered fragments of HTML and sent them to the browser. This is a perfectly acceptable technique, but it is verbose (because the server is sending the HTML elements along with the data) and it limits what can be done with the data at the browser.

One way to address both of these issues is to use the *JavaScript Object Notation* (JSON) format, which is a language-independent way of expressing data. It emerged from the JavaScript language, but has long since taken on a life of its own and is widely used. In this section, I'll show you how to create an action method that returns JSON data, as well as how to process that data in the browser.

■ **Tip** In Chapter 27, I describe the Web API feature, which is an alternative approach for creating Web services.

Adding JSON Support to the Controller

The MVC Framework makes creating an action method that generates JSON data simple. You can see how I have added such an action method to the `People` controller in Listing 23-17.

Listing 23-17. An Action Method That Returns JSON Data in the PeopleController.cs File

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

namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        private IEnumerable<Person> GetData(string selectedRole) {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return data;
        }
```

```
        public JsonResult GetPeopleDataJson(string selectedRole = "All") {
            IEnumerable<Person> data = GetData(selectedRole);
            return Json(data, JsonRequestBehavior.AllowGet);
        }

        public PartialViewResult GetPeopleData(string selectedRole = "All") {
            return PartialView(GetData(selectedRole));
        }

        public ActionResult GetPeople(string selectedRole = "All") {
            return View((object)selectedRole);
        }
    }
}
```

Since I want to present the same data in two different formats (HTML and JSON), I have refactored the controller so that there is a common (and `private`) `GetData` method that is responsible for performing the filtering.

I have added a new action method called `GetPeopleDataJson`, which returns a `JsonResult` object. This is a special kind of `ActionResult` that tells the view engine that I want to return JSON data to the client, rather than HTML. (You can learn more about the `ActionResult` class and the role it plays in the MVC Framework in Chapter 17.)

I create a `JsonResult` by calling the `Json` method in the action method, passing in the data that I want converted to the JSON format, like this:

```
...
return Json(data, JsonRequestBehavior.AllowGet);
...
```

In this case, I have also passed in the `AllowGet` value from the `JsonRequestBehavior` enumeration. By default, JSON data will only be sent in response to `POST` requests, but by passing this value as a parameter to the `Json` method, I tell the MVC Framework to respond to `GET` requests as well.

■ **Caution** You should only use `JsonRequestBehavior.AllowGet` if the data you are returning is not private. Due to a security issue in many Web browsers, it's possible for third-party sites to intercept JSON data that you return in response to a GET request, which is why `JsonResult` will not respond to GET requests by default. In most cases, you will be able to use POST requests to retrieve the JSON data instead, avoiding the problem. For more information, see http://haacked.com/archive/2009/06/25/json-hijacking.aspx.

Processing JSON in the Browser

To process the JSON I retrieve from the MVC Framework application server, I specify a JavaScript function using the `OnSuccess` callback property in the `AjaxOptions` class. In Listing 23-18, you can see how I have updated the `GetPeople.cshtml` view file to remove the handler functions I defined in the last section and use the `OnSuccess` callback to process the JSON data.

Listing 23-18. Working with JSON Data in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}

<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><td>"
                + person.LastName + "</td><td>" + person.Role + "</td></tr>");
        }
    }
</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

```
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                    Url = Url.Action("GetPeopleDataJson", new {selectedRole = role}),
                    OnSuccess = "processData"
                })
        </div>
    }
</div>
```

I have defined a new function called `processData`, which contains some basic jQuery code that processes the JSON objects and uses them to create the `tr` and `td` elements needed to populate the `table`.

■ **Tip** I don't go into jQuery in this book because it is a topic in and of itself. I love jQuery, though, and if you want to learn more about it, then I have written *Pro jQuery* 2.0 (Apress, 2013).

Notice that I have removed the value for the `UpdateTargetId` property from the `AjaxOptions` objects I created for the links. If you forget to do this, the unobtrusive Ajax feature will try and treat the JSON data it retrieves from the server as HTML. You can usually tell this is happening because the contents of the target element will be removed but not replaced with any new data.

You can see the result of the switch to JSON by starting the application, navigating to the `/People/GetPeople` URL, and clicking one of the links. As Figure 23-9 shows, I don't get quite the right result. In particular, the information displayed in the `Role` column of the table isn't correct. I will explain why this happens and show you how to make it right in the next section.

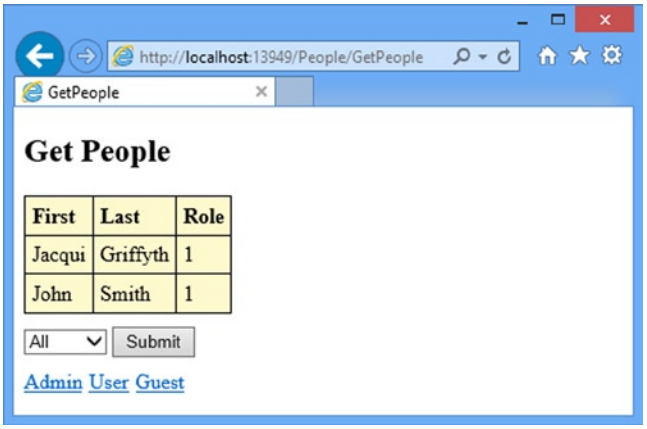

Figure 23-9. *Working with JSON data instead of HTML fragments*

Preparing Data for Encoding

When I called the Json method from within the GetPeopleDataJson action method, I left the MVC Framework to figure out how to encode People objects in the JSON format. The MVC Framework doesn't have any special insights into the model types in an application, and so it makes a best-effort guess about what it needs to do. Here is how the MVC Framework expresses a single Person object in JSON:

```
...
{"PersonId":0,"FirstName":"Adam","LastName":"Freeman",
   "BirthDate":"\/Date(62135596800000)\/","HomeAddress":null,"IsApproved":false,"Role":0}
...
```

It looks like a bit of a mess, but the result is actually pretty clever—it just isn't quite what I need. First, all the properties defined by the Person class are represented in the JSON, even though I did not assign values to some of them in the People controller. In some cases, the default value for the type has been used (false is used for IsApproved, for example), and in others null has been used (such as for HomeAddress). Some values are converted into a form that can be readily interpreted by JavaScript, such as the BirthDate property, but others are not handled as well, such as using 0 for the Role property rather than Admin.

VIEWING JSON DATA

It can be useful to see what JSON data your action methods return and the easiest way to do this is to enter a URL that targets the action in the browser, like this:

http://localhost:13949/People/GetPeopleDataJson?selectedRole=Guest

You can do this in pretty much any browser, but most will force you to save and open a text file before you can see the JSON content. I like to use the Google Chrome browser for this because it helpfully displays the JSON data in the main browser window, which makes the process quicker and means you don't end up with dozens of open text file windows. I also recommend Fiddler (www.fiddler2.com), which is an excellent Web debugging proxy that allows you to dig right into the details of the data sent between the browser and the server.

The MVC Framework has made a good attempt, but I end up sending properties to the browser that I don't subsequently use and the Role value isn't expressed in a useful way. This is a typical situation when relying on the default JSON encoding, and some preparation of the data you want to send the client is usually required. In Listing 23-19, you can see how I have revised the GetPeopleDataJson action method in the People controller to prepare the data I pass to the Json method.

Listing 23-19. Preparing Data Objects for JSON Encoding in the PeopleController.cs File

```
...
public JsonResult GetPeopleDataJson(string selectedRole = "All") {
    var data = GetData(selectedRole).Select(p => new {
        FirstName = p.FirstName,
        LastName = p.LastName,
        Role = Enum.GetName(typeof(Role), p.Role)
    });
    return Json(data, JsonRequestBehavior.AllowGet);
}
...
```

I have used LINQ to create a sequence of new objects that contain just the `FirstName` and `LastName` properties from the `Person` objects, along with the string representation of the `Role` value. The effect of this change is that I get JSON data that contains just the properties I want, expressed in a way that is more useful to the jQuery code, like this:

```
...
{"FirstName":"Adam","LastName":"Freeman","Role":"Admin"}
...
```

Figure 23-10 shows the change in the output displayed in the browser. You can't tell the unused properties are not sent, of course, but you can see that the `Role` column contains the right values.

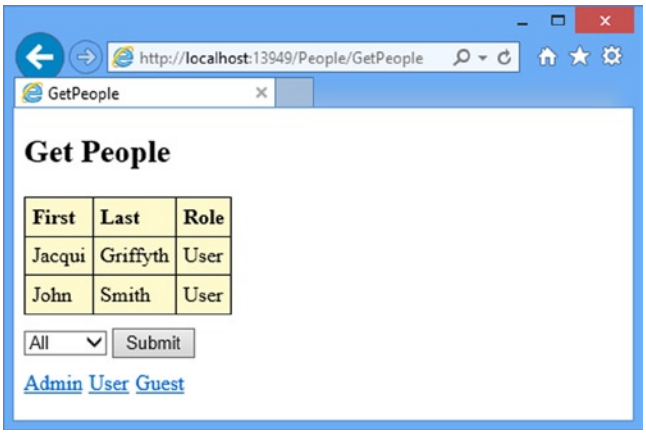

Figure 23-10. *The effect of preparing the data objects for JSON encoding*

■ **Tip** You may have to clear your browser history to see the changes for this example.

Detecting Ajax Requests in the Action Method

The `People` controller presently contains two action methods so that I can support requests for HTML and JSON data. This is usually how I build controllers, because I like lots of short and simple actions, but you don't have to work this way. The MVC Framework provides a simple way of detecting Ajax requests, which means that you can create a single action method that handles multiple data formats. In Listing 23-20, you can see how I have refactored the `Person` controller to contain a single action that handles both JSON and HTML.

Listing 23-20. Creating a Single Action Method in the PersonController.cs File

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

```
namespace HelperMethods.Controllers {
    public class PeopleController : Controller {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };

        public ActionResult Index() {
            return View();
        }

        public ActionResult GetPeopleData(string selectedRole = "All") {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All") {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            if (Request.IsAjaxRequest()) {
                var formattedData = data.Select(p => new {
                    FirstName = p.FirstName,
                    LastName = p.LastName,
                    Role = Enum.GetName(typeof(Role), p.Role)
                });
                return Json(formattedData, JsonRequestBehavior.AllowGet);
            } else {
                return PartialView(data);
            }
        }

        public ActionResult GetPeople(string selectedRole = "All") {
            return View((object)selectedRole);
        }
    }
}
```

I used the `Request.IsAjaxRequest` method to detect Ajax requests and deliver the JSON format if the result is `true`. There are a couple of limitations that you should be aware of before you follow this approach. First, the `IsAjaxRequest` methods returns `true` if the browser has included the `X-Requested-With` header in its request and set the value to `XMLHttpRequest`. This is a widely used convention, but it isn't universal and so you should consider whether your users are likely to make requests that require JSON data without setting this header.

The second limitation is that it assumes that all Ajax requests require JSON data. Your application may be better served by separating the way that a request has been made from the data format that the client seeks. This is my preferred approach and the reason I tend to define separate action methods for data formats.

I also need to make two changes to the `GetPeople.cshtml` view to support the single action method, as shown in Listing 23-21.

Listing 23-21. Supporting a Single Data Action Method in the GetPeople.cshtml File

```
@using HelperMethods.Models
@model string
@{
    ViewBag.Title = "GetPeople";
    Layout = "/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        OnSuccess = "processData"
    };
}

<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><td>"
                + person.LastName + "</td><td>" + person.Role
                + "</td></tr>");
        }
    }
</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display:none">
    <p>Loading Data...</p>
</div>

<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm(ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

```
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new {selectedRole = role},
                new AjaxOptions {
                    Url = Url.Action("GetPeopleData", new {selectedRole = role}),
                    OnSuccess = "processData"
                })
        </div>
    }
</div>
```

The first change is to the `AjaxOptions` object I use for the Ajax-enabled `form`. Because I am no longer able to receive an HTML fragment via an Ajax request, I had to use the same `processData` function to handle the JSON server response that I created for the Ajax-enabled links. The second change is to the value of the `Url` property for the `AjaxOptions` objects created for the links. The `GetPeopleDataJson` action no longer exists and I target the `GetPeopleData` action instead.

Summary

In this chapter, I looked at the MVC Framework's unobtrusive Ajax feature, taking advantage of the functionality of the jQuery library in a simple and elegant way and without needing to add lots of code to views. If you are able to work with HTML fragments, then you don't need to add any JavaScript code to your views at all. But I like working with JSON, which means that I tend to need small JavaScript functions that use jQuery to process the data and generate the HTML elements I require. In the next chapter, I look at one of the most interesting and useful aspects of the MVC Framework: model binding.