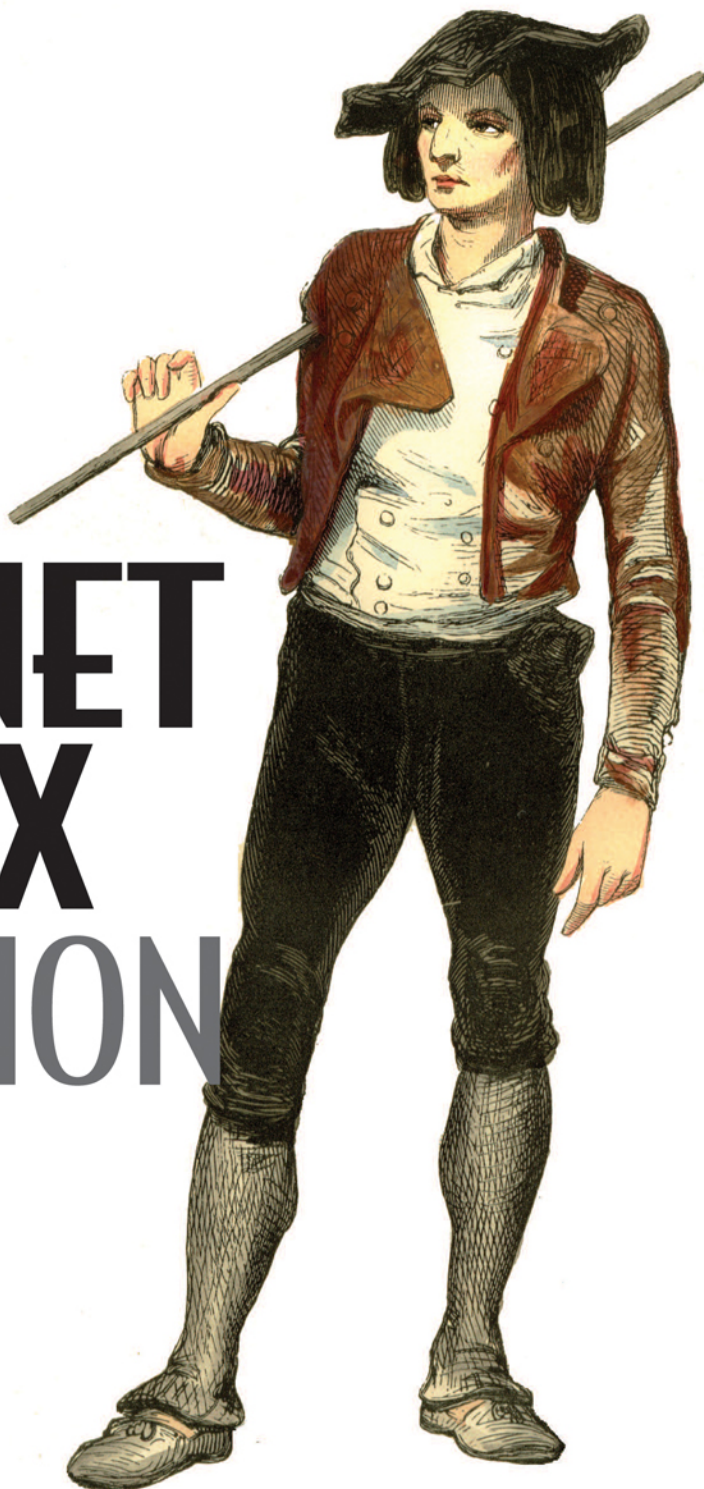


Alessandro Gallo  
David Barkol  
Rama Krishna Vavilala

# ASP.NET AJAX IN ACTION

SAMPLE CHAPTER

 MANNING





***ASP.NET Ajax in Action***

by Alessandro Gallo  
David Barkol  
and Rama Krishna Vavilala

Chapter 4

Copyright 2008 Manning Publications

# *brief contents*

---

## **PART 1 ASP.NET AJAX BASICS .....1**

- 1 ■ Introducing ASP.NET AJAX 3
- 2 ■ First steps with the Microsoft Ajax Library 36
- 3 ■ JavaScript for Ajax developers 73
- 4 ■ Exploring the Ajax server extensions 114
- 5 ■ Making asynchronous network calls 141
- 6 ■ Partial-page rendering with UpdatePanels 194

## **PART 2 ADVANCED TECHNIQUES .....229**

- 7 ■ Under the hood of the UpdatePanel 231
- 8 ■ ASP.NET AJAX client components 264
- 9 ■ Building Ajax-enabled controls 299
- 10 ■ Developing with the Ajax Control Toolkit 332

<b>PART 3</b>	<b>ASP.NET AJAX FUTURES .....</b>	<b>371</b>
11	■ XML Script	373
12	■ Dragging and dropping	410
<b>PART 4</b>	<b>MASTERING ASP.NET AJAX .....</b>	<b>441</b>
13	■ Implementing common Ajax patterns	443

# 4

## *Exploring the Ajax server extensions*

---

### ***In this chapter:***

- Updating an existing ASPNET application
- Performing partial page updates with the UpdatePanel
- Using the ScriptManager
- Working with timers
- Obtaining user feedback

What makes ASP.NET AJAX unique and separates it from other Ajax toolkits and frameworks is the fact that its architecture spans both the client and server. In addition to a rich set of JavaScript libraries, it provides a set of server controls to assist in Ajax development. In the previous two chapters, we revealed the basics of the Microsoft Ajax Library and its ambitions of simplifying Ajax and JavaScript for client-side development. Because most Ajax development originates from the client, these chapters are a pivotal part of the book and will serve as a valuable reference for many of the later chapters.

In this chapter, we continue our discussion of ASP.NET AJAX by delving into the server-side portion of the framework, called the *Ajax server extensions*. If you're familiar with the basics of the server extensions, you may wish to skim this chapter or jump ahead to chapters 6 and 7 to gain a deeper understanding of their inner workings. Nonetheless, the foundation we lay here is important and will be beneficial for even experienced Ajax developers.

As the name implies, Ajax server extensions offer Ajax support for server-side development. To help you understand why this is so valuable, we'll expose some of the issues and challenges of Ajax development from the client perspective.

## 4.1 *Ajax for ASP.NET developers*

---

An Ajax application runs in the browser and is written primarily in JavaScript. This process is initiated when a richer and more intuitive application is delivered from the server to the browser. This includes the logic for rendering and updating the UI, as well as communicating with a server for data needs. The end result is an application that runs more smoothly over time and provides a better user experience. This sounds great and is the recommended approach for Ajax development. However, with this approach comes a new set of issues to address.

For example, what about ASP.NET developers who are unfamiliar with JavaScript or prefer to keep the application logic on the server? What about the rare cases when the browser has JavaScript disabled? What about complex controls like the GridView—does it make sense to rewrite these controls for the client? What about security and exposing the application logic on the client?

These are just a few of the common concerns that surface with Ajax development. Thankfully, the ASP.NET AJAX framework offers an alternative.

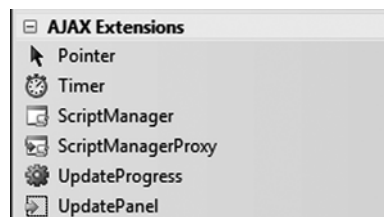
### 4.1.1 *What are the Ajax server extensions?*

Built on top of ASP.NET 2.0, the Ajax server extensions include a new set of server controls and services that simulate Ajax behavior on the client. The extensions

don't adhere to the Ajax model in the traditional sense but respond in a manner that provides that illusion to the end user. In this chapter, we'll focus on the server controls that provide this functionality; the next chapter will give you some insight into how ASP.NET services such as authentication and profile are also supported.

As a quick overview, let's look at the server controls you have at your disposal. Figure 4.1 shows the new controls that are available to the ASP.NET toolbox in Visual Studio. We'll cover each of these controls in this chapter by explaining *when* and *how* they should be used.

Since you're reading this book (and have come this far), chances are you've previously done some ASP.NET development. If you're looking to take an application you wrote previously and add Ajax support to it, then the next few sections should be right up your alley.



**Figure 4.1** The Ajax server extensions are a new set of server controls that complement the already powerful controls in the ASP.NET toolbox.

## 4.2 ***Enhancing an existing ASP.NET site***

---

The goal for the next few sections is straightforward: to take a traditional web application written in ASP.NET and enrich the user experience by adding the Ajax server extensions. In addition to showing how the controls are used, this approach will also demonstrate why and how they're applied in a normal situation. One of the reasons the server extensions are so enticing in this scenario is that they allow you to rapidly integrate Ajax-like behavior into existing applications. It's important to note that without some care and thought, use of the server extensions can be abused and in some cases can even degrade performance.

We'll assess each portion of the application as we reach it; but first, here are some general guidelines to keep in mind:

- *Improve network latency* Do your best to cut back on the amount of data passed between the browser and server. If you can eliminate unnecessary data, network latency and response time will improve.
- *Eliminate full-page refreshes* Keep the interaction between the user and the application as fluid as possible, and avoid a full-page refresh whenever feasible.

- *Keep UI and application logic in code-behind files* Keep any logic used to render or manipulate the UI in the server-side code. This gives you the luxury of supporting browsers that have JavaScript disabled as well as not exposing logic to savvy web users via the client script.
- *Use seamless, transparent integration* Try to keep the existing application intact as much as possible so that future changes will be easy to integrate and few or no changes to the existing logic will be required.
- *Stick to a familiar paradigm* Leverage the server controls so that a typical ASP.NET developer can continue to develop using an already familiar paradigm (server controls and ASP.NET postback mechanism).

If you can meet these goals, you'll have done something rather impressive. Let's begin our journey by examining the existing site you'll be working with for the remainder of the chapter.

#### 4.2.1 A sample ASP.NET site

Figure 4.2 shows the home page for a fictitious and wealthy record company: *Song Unsung Records*. Although it's visually appealing, the application is in desperate need of help in the usability department.

For the sake of a realistic scenario, let's imagine that the site has grown in popularity and that you've been brought aboard as a highly paid consultant to (you hope) improve its usability and performance. After taking a quick look at the interface, you notice immediately that a few areas on the page encourage user interaction: the Artists search at the top, the list of recent feedback items at lower left, and the section for news about a music genre on the right. Unfortunately, interacting with some of the controls in these regions invokes a *postback*, which causes the page to refresh and takes away any interaction the end user has with the site.

We briefly touched on postbacks in chapter 1, but it's worth mentioning again that a postback is costly because of the amount of data sent back and forth to the server and the loss of interaction for the user. Understanding this behavior is important because it's an integral part of how ASP.NET behaves and what the Ajax server extensions are all about.



**Figure 4.2** This application was written for a fictitious record company. Numerous areas on the page encourage user interaction. Each interaction, unfortunately, causes the page to refresh.

### 4.2.2 Configuring an existing ASP.NET site

Creating new sites that are Ajax-enabled is simple: You select the appropriate template from the New Site dialog (see chapter 1 and appendix A) in Visual Studio, and the configuration work is done for you. Taking an existing application and adding Ajax support requires a few more steps. The first involves adding a reference to the library.

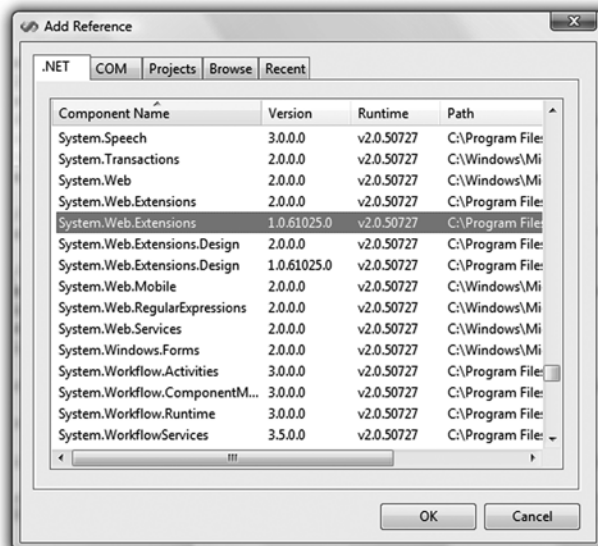
**NOTE** If you're planning to follow along at home, you can download the files from the book's website. If you don't currently have access to the code, the snippets and concepts covered in the following sections are fundamental enough that you can grasp the concepts. The existing sample also requires SQL Server Express—a free version of SQL Server that other samples in the book use as well.

## Postbacks in a nutshell

In ASP.NET, an event—typically a user-driven one such as the clicking of a button—causes a page to send its contents back to the server for processing. This happens principally because pages are stateless, and in order for the server to retrieve the most recent status of a page, the page and all its contents are included in the request back to the server. This is made possible by a hidden field on the page called *ViewState*, which is responsible for storing information about the state of all the server controls in an encoded format. As you can imagine, passing this information back and forth on each postback can become costly over time, not just in terms of bandwidth for the server but also in terms of frustration for the user.

One of the primary objectives of the Ajax server extensions is to find an alternative to some of this undesirable behavior. We'll go deeper into postbacks later in the chapter and with greater detail in the chapters that address the *UpdatePanel* control. You should understand now that postbacks cause a full-page refresh to occur, which is a behavior that Ajax applications seek to suppress or eliminate.

To add a reference to a library in a website or project, you can select the *Add Reference* option from the *Website* or *Project* menu in the menu bar. You can also right-click the site or project in the *Solution Explorer* tab of *Visual Studio* and choose the same option. A dialog similar to the one depicted in figure 4.3 is displayed.



**Figure 4.3**

**The *System.Web.Extensions* library is visible in the .NET tab of the Add Reference dialog. If this isn't visible but the framework has been installed, then you can select the 'Browse' tab to add the dll manually. If you don't see this, you might want to investigate your installation and confirm that the framework has been installed correctly.**

From the dialog, select `System.Web.Extensions` to add a reference to ASP.NET AJAX—this should appear after you’ve successfully installed the framework. If this option isn’t present in the list, take a moment to confirm that you’ve installed the framework correctly, and then select the `Browse` tab to navigate to the `System.Web.Extensions.dll` file on your local machine. Next up is the `web.config` file that defines some of the settings for the application.

The `web.config` file defines the configurations of ASP.NET applications. Items like handling error pages, permissions, and connections strings are placed there for reference and integration with other libraries and components. For ASP.NET AJAX, `web.config` is used to incorporate HTTP handlers, configuration settings, the generating of proxies, and a few other settings that a website needs to leverage the framework.

Most developers create a new Ajax-enabled site and merge the changes between the new `web.config` file and the one on their existing site. We’ll leave this as exercise for you because it entails simple cut-and-paste steps that are too gratuitous to list here. For a detailed explanation of the `web.config` settings, see <http://ajax.asp.net/docs/ConfiguringASPNETAJAX.aspx>. A helpful video is also available on the ASP.NET AJAX homepage at <http://www.asp.net/learn/videos/view.aspx?tabid=63&id=81>.

Assuming you’ve configured the site accordingly, it’s time to add Ajax support by including the most important control in the framework: the `ScriptManager`.

### 4.3 ***ScriptManager: the brains of an Ajax page***

---

The `ScriptManager` control is considered the brains of an Ajax-enabled page and is by far the most important control in the framework. As we move along in this chapter and throughout the book, we’ll demonstrate how to leverage the `ScriptManager` and reveal its intricacies. The important thing to understand at this point is that, as the name suggests, this control is responsible for many of the operations that take place during an Ajax application.

Because you want this control to be present on all the pages of the site, you place it in the master page of the web application rather than in the home page (or content page):

```
<asp:ScriptManager ID="ScriptManager1" runat="server" />
```

You place it in the master page so that any content pages that inherit from it receive the same functionality. This is generally a good practice for similar controls that are used across multiple content pages. Furthermore, this invisible control must be

### More on master pages

Master pages are used to define a consistent look and feel, as well as behavior, for a group of pages in an application. Each page that adopts the look and feel of a master page is called a *content page*. Whenever possible, it's best to place the ScriptManager in a master page so that each content page that inherits from it adopts the same behavior. For more information on master pages, visit <http://msdn2.microsoft.com/en-us/library/wtxbf3hh.aspx>.

declared *before* all other Ajax-enabled server controls in the page hierarchy to ensure that they're loaded and initialized accordingly.

Even though the ScriptManager control isn't declared in the content page, you can easily retrieve an instance of it by calling its static method `GetCurrent` and passing in the current Page instance:

```
ScriptManager scriptManager = ScriptManager.GetCurrent(this.Page);
```

With this instance, you can manage and configure the way the errors, scripts, and other settings on the page behave. We'll explore some of this in a moment; first, let's see what adding the ScriptManager to the page does to the application.

#### 4.3.1 Understanding the ScriptManager

The primary responsibility of the ScriptManager is to deliver scripts to the browser. The scripts it deploys can originate from the ASP.NET AJAX library—embedded resources in the `System.Web.Extensions.dll`, local files on the server, or embedded resources in other assemblies. By default, adding the control to the page, declaratively or programmatically, delivers the required scripts you need for Ajax functionality on the page. To see the evidence, right-click the home page from the browser, and select the View Source option (or select View > Source in IE, or View > Page-Source in Firefox). In the viewed source window, search for an occurrence of `ScriptResource.axd`. You'll find something similar to (but not exactly like) listing 4.1.

#### Listing 4.1 An example of how a script is deployed with the ScriptManager

```
<script src="/04/ScriptResource.axd?d=zQoixCVkx8JK9a1Az_40OriP7
iw9S-TvBA24ugyHeZ8NSIfT6_bRe7yPttg-
sOhCr1udljBUWNQa9KSAugqepLY7DN4cuXzH5ybztCger
rk1&amp;t=633141075498906250"
type="text/javascript">
</script>
```

Let's decode what this tag means; this is at the core of how scripts are delivered to the client.

In ASP.NET 2.0, resources embedded in an assembly are accessed through the `WebResource.axd` HTTP handler. In the ASP.NET AJAX framework, a new HTTP handler called `ScriptResource.axd` replaces it with some additional functionality for localization and browser compression. Listing 4.1 shows a reference to a script assigned by the `ScriptManager` that is eventually downloaded by the new handler.

What about the cryptic text? How does the browser decipher it, and what does it mean? A closer look exposes two parameters: `d` and `t`. They assist the browser in identifying and caching the resource. The first is the encoded resource key, assigned to the `d` parameter. The second is the timestamp, `t`, that signifies the last modification made to the assembly (for example, `t=632962425253593750`). When the page is loaded a second time, the browser recognizes the parameters and spares the user the download by using what's in its cache to retrieve the resources.

**NOTE** Embedding resources in an assembly is a common technique for controls and libraries that require resources like images and scripts. This approach simplifies how controls are packaged and deployed.

Now that you understand how the scripts are downloaded, let's see how you can leverage the `ScriptManager` control to deploy additional scripts.

### 4.3.2 Deploying JavaScript files

Earlier, we examined how the `ScriptManager` control downloads resources to the browser by using a new HTTP handler: `ScriptResource.axd`. You also got a glimpse of this in chapter 2 when we discussed the Microsoft Ajax Library and how the core JavaScript files in the framework are delivered and manipulated with the `ScriptManager`. The next logical step is for you to learn how other scripts can be deployed.

The `ScriptManager` control has a property called `Scripts` that contains a collection of `ScriptReference` objects. A `ScriptReference` is nothing more than a way of registering a JavaScript file for use on a page. Listing 4.2 demonstrates how to include a few scripts on the page using the `ScriptReference` collection.

**Listing 4.2** A `ScriptReference`, which registers files for deployment to a web page

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
  <Scripts>
    <asp:ScriptReference Path="~/scripts/Script1.js" />
    <asp:ScriptReference Path="~/scripts/Script2.js" />
  </Scripts>
</asp:ScriptManager>
```

```
<asp:ScriptReference Assembly="Demo"
    Name="Demo.SuperScript.js" />
</Scripts>
</asp:ScriptManager>
Cueballs in code and text
```

In the first two entries, local JavaScript files are registered as references for the page. In the third entry, an embedded JavaScript file from an assembly is deployed to the site. Each reference added to the collection results in another `ScriptResource.axd` entry in the response's payload to the browser.

Now that you have a general grasp of how scripts are deployed, let's examine another functionality of the ScriptManager: registering service references.

### 4.3.3 Registering services

Working with JavaScript files is an important component of Ajax programming. However, accessing the server for data from JavaScript is what makes Ajax truly possible. In order to be granted this support with the ASP.NET AJAX framework, you must register a service reference for each local web service you wish to interact with.

The ScriptManager has a property called `Services` that contains a collection of `ServiceReference` objects. A `ServiceReference` object is a mechanism for registering services you can access from JavaScript. The end result is a JavaScript proxy that serves as the gateway to the service from the browser. Listing 4.3 demonstrates how to register local services with the ScriptManager.

**Listing 4.3** A `ServiceReference`, which provides a gateway to the service from JavaScript

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
    <Services>
        <asp:ServiceReference Path="~/Services/MainService.asmx" />
        <asp:ServiceReference Path="~/Services/TestService.asmx" />
    </Services>
</asp:ScriptManager>
```

Chapter 5 will take you deeper into how to communicate with services. For now, you can see that the pattern for adding script references is also applied to service references.

Another important feature of the ScriptManager is the ability to support localization for languages and cultures. Let's quickly examine how this works before moving back to the existing ASP.NET application.

### 4.3.4 Localization

The process of supporting specific languages and cultures in an application is commonly referred to as *localization*. You can also consider localization the act of translating the interface. In ASP.NET, this is typically done by embedding localized resources into an organized structure of assemblies, also known as *satellite assemblies*. The ASP.NET AJAX framework supports both this model and a more client-centric model of using static JavaScript files on the server. Let's explore both of these occurrences to gain a general grasp of localization.

#### Localized script files

Localized JavaScript files are nothing more than files mapped to a specific culture. You create this mapping by including the name of the UI culture in the file-name. For instance, a script file that is targeted for the Italian language could be named `SomeScript.it-IT.js`. The *it-IT* stands for the well-known culture identifier of the Italian language in Italy. Proceeding with this pattern, a French version of the file could appropriately be named `SomeScript.fr-FR.js`, and our comrades in the Ukraine could name their file `SomeScript.uk-UA.js` (you get the point).

Rather than implementing logic that gets the current culture on the user's machine and loads the correct file accordingly, you can (and should) use the `ScriptManager` control to do the work for you. The first step in configuring this is to enable script localization:

```
<asp:ScriptManager ID="ScriptManager1" runat="server"
    EnableScriptLocalization="true">
</asp:ScriptManager>
```

Setting the `EnableScriptLocalization` property of the `ScriptManager` to `true` forces the control to retrieve script files for the current culture, if they're available. By default, this property is set to `false`, which means it doesn't perform any localization lookup for you. Consequently, if you now include a script reference for `SomeScript.js`, intentionally omitting the culture name, the appropriate file is downloaded:

```
<asp:ScriptManager ID="ScriptManager1" runat="server"
    EnableScriptLocalization="true">
    <Scripts>
        <asp:ScriptReference Path="SomeScript.js" />
    </Scripts>
</asp:ScriptManager>
```

To reiterate, if the UI culture on your machine were set to Italian, then `SomeScript.it-IT.js` would be downloaded. Under the hood, the `ScriptManager` uses the

naming convention you just exposed to look for a match against the current culture. You can also force a specific culture by setting the `ResourceUICultures` property in the `ScriptReference`:

```
<asp:ScriptReference Path="SomeScript.js"
    ResourceUICultures="it-IT" />
```

Pretty straightforward so far, but what about debug versions of your JavaScript files? In section 2.1.3, you were introduced to the `ScriptMode` property, which determines the version of a script file to load: release, debug, or auto (based on configuration settings on the page or site). Luckily, the same applies to localized scripts—if you had a debug version of the file called `SomeScript.debug.it-IT.js`, you could load it explicitly by setting the `ScriptMode` property:

```
<asp:ScriptReference Path="SomeScript.js" ResourceUICultures="it-IT"
    ScriptMode="Debug" />
```

The result is that the debug version of the Italian resource is loaded. That's all there is to localization on static JavaScript files. Next, let's see how loading script resources from an assembly works with ASP.NET AJAX.

### Using assembly resources

Packaging scripts and resources as embedded assets into an assembly is a common technique that control developers use. This approach is popular primarily because it simplifies how resources are deployed with the control.

In order for a resource to be recognized by the ASP.NET AJAX framework, it must be decorated with the `WebResource` attribute:

```
[assembly: WebResource("ControlNamespace.Control.js",
    "text/javascript")]
```

In this example, `ControlNamespace` represents the default namespace used in the assembly. The remaining portion, `Control.js`, is the name of the resource.

It's highly recommended that the JavaScript file in the assembly not contain any hard-coded string literals. Instead, it should look up values from a resource file that follows the same naming conventions for the scripts. For example, you could use a .NET resource file named `Messages.it-IT.resx` (or `Messages.en-IE` for our Shillelagh-wielding friends in Ireland) to define strings for that culture. The ASP.NET AJAX script loader automatically converts the .NET string resources into a JavaScript object:

```
Messages={
  "SayThankYou":"Grazie mille.",
  "EnjoyMeal":"Buon appetito!"
};
```

The logic on the client can then be UI culture-independent and reference the string easily:

```
alert (Messages.SayThankYou) ;
```

This gives you a general understanding of how to use embedded scripts in a localization context. The topic of how to embed resources into assemblies is slightly out of the scope of this section; for more detailed information, see <http://msdn2.microsoft.com/en-us/library/ms227427.aspx>.

Most of the work of loading and managing localization is handled by the ScriptManager, thus saving you a load of code and time. Using the ScriptManager for localization also comes with additional benefits.

### **ScriptManager localization benefits**

If your application supports multiple cultures, we strongly recommend loading and leveraging the ScriptManager for localization support. Some of the benefits of using the control include:

- *UI culture detection*—When the EnableScriptLocalization property is enabled, the ScriptManager detects and loads the appropriate script resource for you.
- *Custom UI culture support*—The ResourceUICultures property in the ScriptReference object lets you override and determine which UI cultures are supported for a particular script.
- *Avoidance of indefinite caching*—The ScriptManager employs a timestamp to ensure that embedded scripts aren't cached indefinitely by the browser.
- *Encrypted URLs to resources*—As a security measure, the key that directs the browser to the appropriate script is encrypted.

The ScriptManager makes localization very easy and includes an additional set of features that make it an attractive solution for managing localization.

You should now have a general idea of what the ScriptManager is capable of. More instances of how and when it should be used are covered throughout the book as we mentioned. Let's get back to the application and make use of another server control called the *ScriptManagerProxy*.

#### **4.3.5 Using the ScriptManagerProxy**

One and only one ScriptManager can exist on a page. Adding more than one causes an InvalidOperationException to be raised at runtime. But in some situations, a content page may require a reference to a service or script that isn't made

by the ScriptManager in the parent or master page. The additional references may also be required for only a single page and not for others. In these situations, the ScriptManagerProxy control comes to the rescue.

Suppose the customer has requested that you include a certain script on the home page but not on any of the other pages on the site. Adding this script to the master page would deploy it everywhere, which is the undesired result. Instead, you can leverage the ScriptManagerProxy on the target page to ensure that it's included only there.

#### Listing 4.4 Using the ScriptManagerProxy control in a master-page scenario

```
<asp:ScriptManagerProxy ID="ScriptManagerProxy1" runat="server">
  <Scripts>
    <asp:ScriptReference Path="~/scripts/DummyScript.js" />
  </Scripts>
</asp:ScriptManagerProxy>
```

Just like its parent control, the ScriptManagerProxy has a collection of script and service references. Think of the ScriptManagerProxy as an extension of the ScriptManager control: The influence and settings of the ScriptManager control can be extended to content pages, user controls, and more. At runtime, the settings are merged for each page accordingly.

The key purpose of the ScriptManagerProxy is to add references that weren't included with the ScriptManager. This situation occurs most commonly when you're working with master pages.

We've been laying the groundwork by adding support for the Ajax framework and dropping in the ScriptManager control to deploy JavaScript files. Now we can address those dreaded postbacks and start enhancing the overall user experience. We'll return to the ScriptManager later in the chapter when we discuss error handling.

## 4.4 *Partial-page updates*

Earlier in the chapter, we listed the goals for the existing application and mentioned that eliminating complete page refreshes from occurring would greatly enhance the user experience. To reiterate, instead of updating the whole page all at once, as in traditional ASP.NET applications, you should strive to update only portions of the page—dynamically, without changing any of the application logic if possible.

In a conventional Ajax solution, when the UI and application logic reside on the browser, you're responsible for updating the UI with DHTML techniques and a strong grasp of JavaScript. With the UpdatePanel control, the burden of this type of development is abstracted away with all the heavy lifting done for you by the server extensions. The best way to fully understand this is to see it in action.

#### 4.4.1 Introducing the UpdatePanel control

The UpdatePanel is an Ajax-enabled server control that works closely with the ScriptManager to apply partial-page updates to a page. Portions of the page declared by the UpdatePanel can now be updated incrementally rather than as a result of a page refresh. To demonstrate, let's add the UpdatePanel control to the existing application.

The right column on the home page displays news about a selected genre. Using a DropDownList control to display the available genres and a Repeater control to display news about the selected item, the controls work together to inform the user about the latest relevant news. When the user selects a new item in the drop-down list, a postback occurs, and the page is refreshed. Examining the markup for the DropDownList explains this behavior.

In listing 4.5, you can see that the AutoPostBack property of the dropdown list is set to True. As you can probably guess, this setting invokes the postbacks on each selection change. During the postback, the server-side code processes the request by looking up the selected genre and retrieving its relevant news. Listing 4.6 demonstrates how the server code binds the Repeater control on the form with news about the selected genre.

**Listing 4.5** Selecting an item from the music genre list generates a postback to the server.

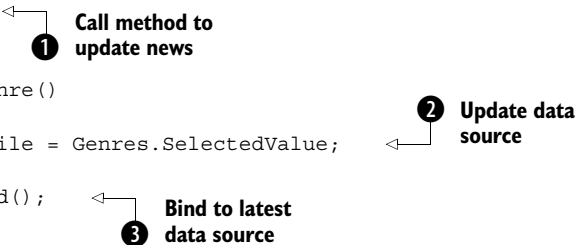
```
<asp:DropDownList ID="Genres" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="Genres_SelectedIndexChanged" >
  <asp:ListItem Text="Rock" Value="~/App_Data/RockFeed.xml"
    Selected="true" />
  <asp:ListItem Text="Jazz" Value="~/App_Data/JazzFeed.xml" />
  <asp:ListItem Text="Blues" Value="~/App_Data/BluesFeed.xml" />
</asp:DropDownList>
```

**Listing 4.6** Selecting an item in the list updates the Repeater's data source.

```
protected void Genres_SelectedIndexChanged(object sender,
                                         EventArgs e)
{
    UpdateGenre();
}

private void UpdateGenre()
{
    GenreSource.DataFile = Genres.SelectedValue;

    GenreNews.DataBind();
}
```



The `Genres_SelectedIndexChanged` method is invoked when the selected genre is changed. It then calls the **1** private `UpdateGenre` method to **2** configure the data source (in this case it's an XML file that represents an RSS feed) and **3** rebind the Repeater.

Because each selection invokes a page refresh, the behavior in the browser isn't appealing to users. Technically, however, it makes a lot of sense. This pattern is common in ASP.NET applications. The postback mechanism is frequently used (and, unfortunately, often abused) to bridge the gap between what is displayed on the UI and the logic on the server.

If you add the `UpdatePanel` control, you can keep everything intact and change the way it behaves for the user. The next time you run the site and select a new genre, the news for a recently selected item is updated without a full-page refresh. For clarity, the relevant source on the page is included in listing 4.7.

**Listing 4.7** Dynamically updating regions during asynchronous postbacks

```
<asp:UpdatePanel ID="GenrePanel" runat="server">
  <ContentTemplate>
    <div class="columnheader">Music News:
      <asp:DropDownList ID="Genres" runat="server"
        AutoPostBack="True"
        OnSelectedIndexChanged="Genres_SelectedIndexChanged" >
        <asp:ListItem Text="Rock" Value="~/App_Data/RockFeed.xml"
          Selected="true" />
        <asp:ListItem Text="Jazz" Value="~/App_Data/JazzFeed.xml" />
        <asp:ListItem Text="Blues" Value="~/App_Data/BluesFeed.xml" />
      </asp:DropDownList>
    </div>
```

```

<asp:Repeater ID="GenreNews" runat="server"
DataSourceID="GenreSource" >
  <ItemTemplate>
    <div class="newshead">
      <asp:HyperLink ID="HyperLink1" runat="server"
        NavigateUrl='<%=XPath("link") %>'
        Text='<%=XPath("title") %>' />
      &nbsp;
      <asp:HyperLink ID="HyperLink2" runat="server"
        NavigateUrl='<%=XPath("link") %>' Text="[read more]" />
    </div>
  </ItemTemplate>
</asp:Repeater>
<hr />
Last Updated: <%= DateTime.Now.ToLongTimeString() %>

<asp:XmlDataSource ID="GenreSource" runat="server"
  DataFile="~/App_Data/RockFeed.xml" XPath="/rss/channel/item">
</asp:XmlDataSource>

</ContentTemplate>
</asp:UpdatePanel>

```

The `ContentTemplate` property of the `UpdatePanel` class defines the regions of the page that are updated dynamically. This time, instead of a normal postback that refreshes the entire page, a new type of postback is introduced: an *asynchronous postback*. An asynchronous postback goes through the page lifecycle and operates like a normal postback, minus the page refresh. This refreshing (pun intended) news means the logic for the UI and application can remain intact.

To demonstrate, let's add code to detect when you're in an asynchronous postback by asking the `ScriptManager` control for more information; see listing 4.8.

**Listing 4.8** During an asynchronous postback, the page goes through the normal page lifecycle.

```

protected void Page_Load(object sender, EventArgs e)
{
  ScriptManager scriptManager = ScriptManager.GetCurrent(this.Page);  ← 1
  if (scriptManager.IsInAsyncPostBack)  ← 2
  {
    // We are doing something cool!
  }
}

```

**Return instance of ScriptManager** 1

**Check for asynchronous postback** 2

You first ❶ retrieve an instance of the `ScriptManager` on the page by calling the static method `GetCurrent` and passing in the parent page. Remember, you declared the `ScriptManager` on the master page, not the content page, so this is the best way to find and retrieve an instance of the `ScriptManager` when in a content page (or child control). An alternative would be to find the control in the `Controls` collection of the master page, but this approach is ostensibly much simpler—under the hood, it does the same thing.

Next, you query the ❷ `IsInAsyncPostBack` property of the `ScriptManager` to determine if you're in the process of handling a normal postback or an asynchronous one. This offers you the option of coding custom logic for each occasion.

**WARNING** Adding an `UpdatePanel` to a page seems so effortless, and the rewards are so great, that many developers entertain the idea of placing the contents of an entire page in a single `UpdatePanel`. This practice is highly discouraged. Although the illusion of Ajax is present, the cost of each postback is significant, and the application's overall performance will suffer greatly over time. As a general rule, try to avoid such solutions, and instead look for portions of the page that can be updated instead of the entire page.

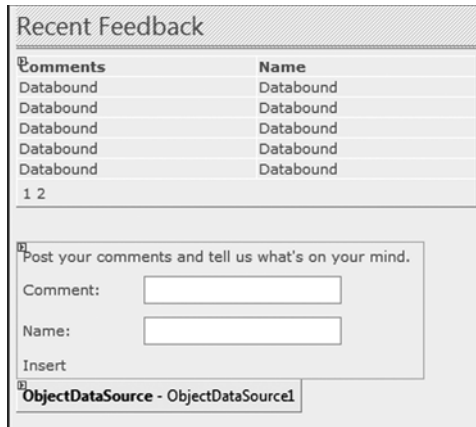
So far, so good—you've added a single `UpdatePanel` to the page and in the process stopped a page refresh from happening each time the user selects a different music genre. Let's now place the focus on the other interactive portion of the site: the feedback area.

#### 4.4.2 More UpdatePanels

Adding multiple `UpdatePanel` controls to a page is not only supported but also encouraged. Doing so means that more regions of the page can be updated dynamically instead of each time a page refreshes. This approach also allows you to take more control of which portions of the page are updated and which ones aren't, thus helping conserve the amount of data passed between the client and server during each postback. To demonstrate, let's add another `UpdatePanel` to address the postbacks that come out of the page's Recent Feedback section.

Figure 4.4 captures Visual Studio's Design view of the related controls before adding the `UpdatePanel`. You see a `GridView` control that is used to display, sort, and page through feedback items. Below the `GridView` is a `DetailsView` control that is used to enter new feedback to the site.

Each time the user attempts to sort, navigate to the next page of results, or add feedback, a postback occurs. Because both controls invoke postbacks and are relatively close to each other in the page layout, you can place a single `UpdatePanel`

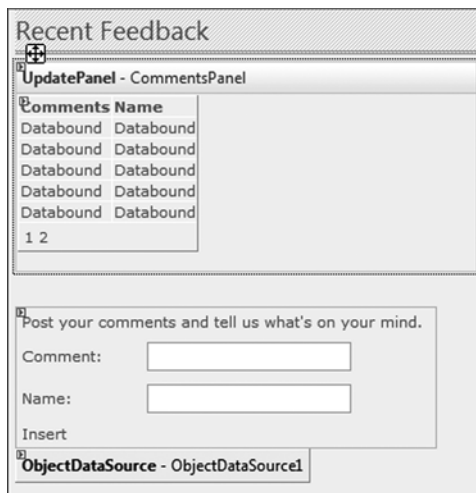
**Figure 4.4**

This snapshot is from the Design view in Visual Studio; it shows the state of the controls before adding Ajax support.

around both of them. After adding the UpdatePanel, the design view of the form resembles figure 4.5.

When you run the site once more, the page refreshes previously invoked from interaction with the news and feedback sections are gone. It's worth mentioning again that postbacks still occur, but now they take place asynchronously and, as a result, update the page incrementally, thus eliminating the flicker.

Unfortunately, you aren't done yet. As you hand over the site to the client for testing, they notice a behavior that they deem undesirable. When a new feedback item is entered, the contents of both UpdatePanel controls are updated instead of just the one for recent feedback. In other words, when a user enters in a new

**Figure 4.5**

Adding the UpdatePanel around the GridView and DetailsView controls replaces their traditional postbacks with asynchronous postbacks.

feedback item or sorts one of the columns, the recent news about a genre (on the right side of the page) also gets updated. You can confirm this by sorting a column like Name on the GridView and watching the Last Updated time in the right column change.

The content of both UpdatePanel controls is updated because by default, every time an asynchronous postback occurs on the page, regardless of which control on the page invoked the postback, an UpdatePanel updates its contents. To solve this, all you need to do is set the UpdateMode property on both panels to Conditional. Doing so tells the UpdatePanel to update its contents only if the postback originates from within itself (from one of its child controls).

**TIP** The default value for the UpdateMode on UpdatePanel controls is Always. However, this setting is rarely needed. A best practice is to always set the mode to Conditional and let the Always condition present itself naturally in your application. Doing so cuts back on the amount of data passed between the server and client and ultimately increases the site's performance. There are limitations to both, which we'll cover in chapters 6 and 7, dedicated to the UpdatePanel control.

This time, when you run the site again, only the contents of the UpdatePanel relative to the user interaction are updated.

Let's take a moment to recap. So far in the chapter, you've eliminated full-page refreshes, you've cut back on the amount of data passed between the client and server, and you've kept all the application and UI logic on the server intact. You've also improved network latency and written no JavaScript code in the process! For extra credit, if the browser had JavaScript disabled, the existing application would continue to function, with normal postbacks and page refreshes—just as it did before you made any updates.

Because most of the initial goals have been met, and given that this is ultimately all about the user experience, perhaps you can find other things to add to the site that add more value and interactivity for the user.

#### 4.4.3 **Insert feedback here**

Adding the UpdatePanel controls to the site radically improves the behavior and overall feel for users. They're spared a page refresh when they interact with the controls, which is a tremendous improvement from the intermittent nature you had before. But as a side effect of this achievement, the user isn't given any indication that something is being updated until it has happened.

Imagine (for the sake of this example) that the source used to retrieve information about a music genre comes from an external RSS feed. This scenario introduces the possibility of slow responses when retrieving a news feed, which means partial-page updates may not happen immediately—and may not happen for quite a while. Previously, the page refresh was an indication that something was happening and that eventually the page would be updated. With partial-page updates, the user is given no visual cue that their actions have been accepted and that work is being done on the other end.

Fortunately, in chapter 1, we introduced the `UpdateProgress` control as a solution for this problem. You'll use the control again here to notify the user that you're retrieving news about the selected genre. Listing 4.9 shows the insertion of the `UpdateProgress` control on the page, right before the `Repeater` control that displays the recent news.

**Listing 4.9** Displaying the `UpdateProgress` control when the `UpdatePanel`'s contents are being updated

```
<asp:UpdateProgress ID="UpdatingNews" runat="server"
  AssociatedUpdatePanelID="GenrePanel" >
  <ProgressTemplate>
    
    &nbsp;&nbsp;&nbsp;Loading ...
  </ProgressTemplate>
</asp:UpdateProgress>
```

← ① Assign to specific UpdatePanel

② Displayed during postback

First, you'd prefer to display the contents of the `UpdateProgress` control only when news about a genre is being retrieved, and not when other `UpdatePanel` controls are being updated. To accomplish this, you set the ① `AssociatedUpdatePanelID` property to the ID of the `UpdatePanel` associated with the music news. This lets you have multiple `UpdateProgress` controls on a single page and gives you added control over how the page is rendered during an asynchronous postback. The `UpdateProgress` control has a ② `ProgressTemplate` property that encapsulates what is to be displayed during an asynchronous postback. For this instance, you use an animated GIF image and some informative text.

Now, when you select a new item from the drop-down list, a subtle but informative message is presented while the data is retrieved. Figure 4.6 displays the `UpdateProgress` control in action.



**Figure 4.6** The `UpdateProgress` control offers a simple and useful tool for keeping the user informed about asynchronous updates on the page.

**TIP** It's easy to get carried away with *Loading* messages. In general, you should try to inform the user with a subtle and informative message that is relevant to the portion of the page being updated. Unless the entire page is being updated, it's usually more considerate and less intrusive to use smaller, useful icons and text to relay messages to the user. Gratuitous messages can have a negative effect on the overall user experience and should generally be avoided.

You're almost finished with the server-extension controls. You've used every one of them except the Timer control, which, when used effectively, can complement the UpdatePanel control and give you the ability to apply partial-page updates at set intervals.

#### 4.4.4 Working with a timer

Included in the Ajax server extensions is a control called the Timer. As its name implies, the control creates a timer on the client that invokes a postback at an interval you specify (in milliseconds). For the existing application, you'll use the Timer control in conjunction with the UpdatePanel to retrieve and display the latest news about the selected genre. Because news about a genre can change often, this subtle addition adds a little extra value to the site because it keeps the user's attention. Listing 4.10 shows how to declare the Timer control on the page.

**Listing 4.10** The Timer runs in the client and invokes a postback at each interval.

```
<asp:Timer ID="NewsTimer" runat="server" Interval="10000"
  OnTick="UpdateNews" />
```

The declaration sets the interval to 10 seconds (or rather, its equivalent, 10000 milliseconds) and also assigns an `UpdateNews` handler to the `OnTick` event. Normally, this would be too frequent of an interval for news updates—we use it here for demonstration purposes only. Also, for reasons we’re about to discuss, you place the `Timer` control *outside* the `UpdatePanel` instead of in the `ContentTemplate` declaration, as in previous examples.

**TIP** It’s important to understand that the ticks for the `Timer` happen on the browser, not the server. Using this control requires you to be mindful of the system resources on the end user’s machine. In general, set the control’s interval to the highest value possible. Setting the interval value to too short an amount may put too much strain on the system and cause unpredictable behavior.

To accompany the declarative code, the server-side code calls the same `UpdateGenre` method used earlier to update the interface. Listing 4.11 shows the code-behind for the `Tick` event handler.

**Listing 4.11 The `Tick` event, which calls the private `UpdateGenre` method**

```
protected void UpdateNews(object sender, EventArgs e)
{
    UpdateGenre();
}

private void UpdateGenre()
{
    GenreSource.DataFile = Genres.SelectedValue;
    GenreNews.DataBind();
}
```

Because the `Timer` control isn’t encapsulated by the `UpdatePanel`, each interval that invokes a postback causes the page to refresh. This happens because the `UpdatePanel` hasn’t been made aware that you’d like to use the `Tick` event of the `Timer` control to invoke an asynchronous postback. To resolve this, you register the `Tick` event with the `UpdatePanel` by adding the event to the control’s `Triggers` collection. The next time you run the application, you’ll notice that the `Last Updated` time is incremented at each interval.

Listing 4.12 shows the entire contents of the music genre section as well as the declaration of the `Timer` control at the end.

**Listing 4.12 Registering the Timer control's Tick event to ensure asynchronous postbacks**

```

<asp:UpdatePanel ID="GenrePanel" runat="server"
UpdateMode="Conditional">
  <ContentTemplate>
    <div class="columnheader">Music News:
      <asp:DropDownList ID="Genres" runat="server"
        AutoPostBack="True"
        OnSelectedIndexChanged="Genres_SelectedIndexChanged" >
        <asp:ListItem Text="Rock" Value="~/App_Data/RockFeed.xml"
          Selected="true" />
        <asp:ListItem Text="Jazz" Value="~/App_Data/JazzFeed.xml" />
        <asp:ListItem Text="Blues" Value="~/App_Data/BluesFeed.xml" />
      </asp:DropDownList>
    </div>

    <asp:UpdateProgress ID="UpdatingNews" runat="server"
      AssociatedUpdatePanelID="GenrePanel" >
      <ProgressTemplate>
        &nbsp;&nbsp;&nbsp;Loading ...
      </ProgressTemplate>
    </asp:UpdateProgress>

    <asp:Repeater ID="GenreNews" runat="server"
      DataSourceID="GenreSource" >
      <ItemTemplate>
        <div class="newshead">
          <asp:HyperLink ID="HyperLink1" runat="server"
            NavigateUrl='<%=XPath("link") %>'
            Text='<%=XPath("title") %>' />
          &nbsp;&nbsp;&nbsp;
          <asp:HyperLink ID="HyperLink2" runat="server"
            NavigateUrl='<%=XPath("link") %>' Text="[read more]" />
        </div>
      </ItemTemplate>
    </asp:Repeater>
    <hr />
    Last Updated: <%= DateTime.Now.ToLongTimeString() %>

  </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="NewsTimer"
      EventName="Tick" />
  </Triggers>
</asp:UpdatePanel>

<asp:Timer ID="NewsTimer" runat="server" Interval="10000"
  OnTick="UpdateNews" />

```

Register async  
postback

Postback  
every 10  
seconds

You've now used every control in the Ajax server extensions, and the result is an application that is far more engaging and responsive than when you started. Along the way, you picked up a collection of best practices for getting the most out of the extensions, and you also got a glimpse into how the ScriptManager works under the hood.

But you're not done yet. Even the best applications contain errors or raise exceptions.

#### 4.4.5 Error handling

Things have been working smoothly so far, but in the real world, errors and exceptions occur. To wrap up this chapter, let's examine what you have at your disposal to make handling these occurrences more manageable. Listing 4.13 shows a snippet of code that purposely throws an exception after the user has selected a new music genre from the drop-down list.

**Listing 4.13** Throwing an exception to see how the page handles it

```
protected void Genres_SelectedIndexChanged(object sender,
                                         EventArgs e)
{
    UpdateGenre();
    throw new Exception("Look out!");
}
```

Earlier, you set the `AutoPostBack` property of this control to `true` and also placed it in an `UpdatePanel`. This means the postback that originates from here is asynchronous, also known as an *Ajax postback*. Typically, depending on the settings of the `web.config` file, an error during a normal postback results in the stack trace and error information being shown on the screen. This time, the browser relays the exception information in a dialog box (see figure 4.7).

This result can be informative for developers, but displaying the same message from the exception back to the user isn't always the best idea. Fortunately, the ScriptManager control throws an event called `AsyncPostBackError` that provides you with an opportunity to update the text in the dialog box before it's presented to the user. Listing 4.14 demonstrates how a handler for the event is registered and the message updated before reaching the user.



**Figure 4.7** By default, exceptions that occur during asynchronous postbacks are displayed in alert dialogs.

**Listing 4.14 Raising the AsyncPostBackError event before the dialog is displayed to the user**

```
protected void Page_Load(object sender, EventArgs e)
{
    ScriptManager scriptManager = ScriptManager.GetCurrent(this.Page);
    scriptManager.AsyncPostBackError += new
        EventHandler<AsyncPostBackEventArgs>(OnAsyncPostBackError);
}

void OnAsyncPostBackError(object sender,
    AsyncPostBackEventArgs e)
{
    ScriptManager.GetCurrent(this.Page).AsyncPostBackErrorMessage =
        "We're sorry, an unexpected error has occurred.";
}
```

Now, when you select another music genre from the list, you're presented with a message box that contains the custom message instead of the one coming from the exception.

Even with the custom error message, it's still considered a best practice to provide a default error page for a website rather than display an alert dialog or stack trace to the user. This way, when an exception occurs, the user is redirected to a friendly page that is informative and useful. The mechanism for handling errors is configurable in the `customErrors` section of `web.config`:

```
<system.web>
  <customErrors mode="On|Off|RemoteOnly"
    defaultRedirect="ErrorPage.aspx">
    ...
  </customErrors>
```

The `mode` property of the `customErrors` section governs how error messages are to be handled. When this property is set to `On`, the user is redirected to the error page defined in the `defaultRedirect` property. The `Off` setting always shows the stack trace—or, in this case, the dialog box with the error message. The `RemoteOnly` value redirects the user to the error page only if they're on a remote machine; otherwise, the same behavior used for the `Off` setting is applied. Due to its flexibility,



**Figure 4.8** You can change the error message during the `AsyncPostBackError` event.

the `RemoteOnly` setting is the most appropriate for developers who wish to debug applications locally and view details about exceptions as they occur.

The `ScriptManager` control provides a property for overriding this mechanism. By default, the `AllowCustomErrorsRedirect` property is set to `true`. This setting honors the values set in the `customErrors` section. Setting this property to `false` forces the dialog to appear when exceptions occur (see listing 4.15).

**Listing 4.15** The `AllowCustomErrorsRedirect` property overrides the `web.config` settings.

```
protected void Page_Load(object sender, EventArgs e)
{
    ScriptManager scriptManager = ScriptManager.GetCurrent(this.Page);
    ...
    scriptManager.AllowCustomErrorsRedirect = false;
}
```

The `AllowCustomErrorsRedirect` value must be set on or before the `Load` event in the ASP.NET page lifecycle. Doing so afterward has no affect on the settings configured in the `customErrors` section. Chapter 7 will show you how to handle errors more elegantly when we examine the events that occur on the client side during asynchronous postbacks.

For now, the lesson is this: always provide a general error page for users. If you have to show the user a dialog box during an exception, handle the `AsyncPostBackError` event to display a friendly and user-centric message as opposed to the message from the exception itself.

## 4.5 Summary

We began this chapter by presenting an alternative to client-side Ajax development. Using the Ajax server extensions, ASP.NET developers can simulate Ajax behavior in the browser. Sometimes a client-centric Ajax solution isn't appropriate for a site. In these cases, you can still use a server-centric solution that leverages these new controls to improve the user experience. In many situations, using both approaches makes sense.

The next chapter will round out your understanding of the core ASP.NET AJAX framework by examining how asynchronous calls are made from the browser. It will also pick up where we left off with the server extensions by exposing how you can use the authentication and profile services in ASP.NET from client script.