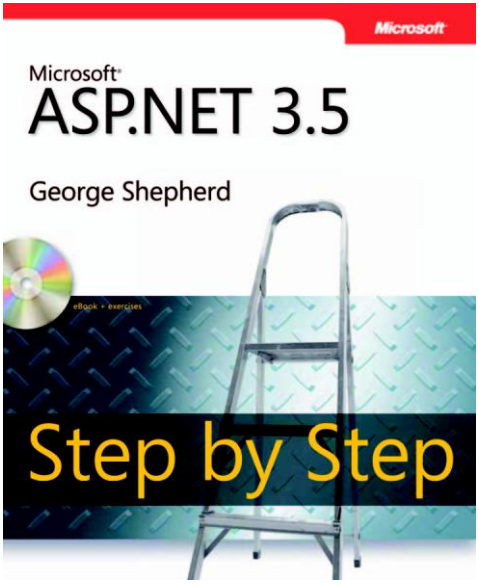


Microsoft® ASP.NET 3.5 Step by Step

George Shepherd

To learn more about this book, visit Microsoft Learning at http://www.microsoft.com/MSPress/books/11239.aspx.

9780735624269

Microsoft® Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2007942085

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental Editor: Devon Musgrave
Project Editor: Kathleen Atkins
Editorial Production: P.M. Gordon Associates
Technical Reviewer: Kenn Scribner; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X14-40155

Table of Contents

Introduction xix
Acknowledgments xxix

Part I **Fundamentals**

1 Web Application Basics 3
HTTP Requests 4
HTTP Requests from a Browser 4
Making HTTP Requests without a Browser 6
HyperText Markup Language 8
Dynamic Content 9
HTML Forms 10
Common Gateway Interface (Very Retro) 12
The Microsoft Platform as a Web Server 12
Internet Information Services 12
Internet Services Application Programming Interface DLLs 13
Internet Information Services 14
Classic ASP (Putting ASP.NET into Perspective) 19
Web Development Concepts 22
ASP.NET 23
Summary 24
Chapter 1 Quick Reference 24

2 ASP.NET Application Fundamentals 25
The Canonical Hello World Application 25
Building the HelloWorld Web Application 26
Mixing HTML with Executable Code 31
Server-Side Executable Blocks 34
The ASP.NET Compilation Model 41

Coding Options. 43
ASP.NET 1.x Style 43
Modern ASP.NET Style. 44
The ASP.NET HTTP Pipeline 46
The IIS 5.x and IIS 6.x Pipeline. 46
The IIS 7.0 Integrated Pipeline 47
Tapping the Pipeline 47
Visual Studio and ASP.NET 50
Local IIS Web Sites 50
File System–Based Web Sites 50
FTP Web Sites 51
Remote Web Sites 51
Hello World and Visual Studio 52
Summary 57
Chapter 2 Quick Reference 58

3 The Page Rendering Model. 59
Rendering Controls as Tags 59
Packaging UI as Components. 62
The Page Using ASP.NET. 63
The Page's Rendering Model 65
The Page's Control Tree. 66
Adding Controls Using Visual Studio 68
Building a Page with Visual Studio 68
Layout Considerations. 76
Summary 77
Chapter 3 Quick Reference 78

4 Custom Rendered Controls 79
The *Control* Class 79
Visual Studio and Custom Controls 81
A Palindrome Checker 88
Controls and Events 92
HtmlTextWriter and Controls 95
Controls and *ViewState* 98
Summary 101
Chapter 4 Quick Reference 101

5 Composite Controls 103
Composite Controls versus Rendered Controls 103
Custom Composite Controls 104
User Controls 112
When to Use Each Type of Control 118
Summary 119
Chapter 5 Quick Reference 119

6 Control Potpourri 121
Validation 121
How Page Validation Works 127
Other Validators 129
Validator Properties 130
Image-Based Controls 130
TreeView 134
MultiView 138
Summary 140
Chapter 6 Quick Reference 141

Part II Advanced Features

7 Web Parts 145
A Brief History of Web Parts 146
What Good Are Web Parts? 146
Developing Web Parts Controls 147
Web Parts Page Development 147
Web Parts Application Development 147
The Web Parts Architecture 147
WebPartManager and *WebZones* 148
Built-in Zones 148
Built-in Web Parts 149
Developing a Web Part 158
Summary 168
Chapter 7 Quick Reference 168

8 A Consistent Look and Feel 169
A Consistent Look and Feel 169
ASP.NET Master Pages 170
Themes 181

Skins 185
Summary 186
Chapter 8 Quick Reference 187

9 Configuration 189
Windows Configuration 190
.NET Configuration 190
Machine.Config 191
Configuration Section Handlers 191
Web.Config 193
Managing Configuration in ASP.NET 1.x 194
Managing Configuration in Later Versions of ASP.NET 195
Configuring ASP.NET from IIS 200
Summary 204
Chapter 9 Quick Reference 205

10 Logging In 207
Web-Based Security 207
Securing IIS 208
Basic Forms Authentication 209
ASP.NET Authentication Services 214
The *FormsAuthentication* Class 214
An Optional Login Page 215
Managing Users 219
ASP.NET Login Controls 225
Authorizing Users 229
Summary 232
Chapter 10 Quick Reference 232

11 Data Binding 233
Representing Collections without Data Binding 233
Representing Collections with Data Binding 234
ListControl-Based Controls 234
TreeView 235
Menu 235
FormView 235
GridView 235
DetailsView 235

DataList 236
Repeater 236
Simple Data Binding 236
Accessing Databases 240
The .NET Database Story 241
Connections 241
Commands 243
Managing Results 244
ASP.NET Data Sources 246
Other Data-bound Controls 251
LINQ 259
Summary 261
Chapter 11 Quick Reference 262

12 Web Site Navigation 263
ASP.NET's Navigation Support 263
The Navigation Controls 263
XML Site Maps 265
The *SiteMapProvider* 265
The *SiteMap* Class 265
The *SiteMapNode* 266
The Navigation Controls 267
The *Menu* and *TreeView* Controls 267
The *SiteMapPath* Control 268
Site Map Configuration 269
Building a Navigable Web Site 270
Trapping the *SiteMapResolve* Event 274
Custom Attributes for Each Node 275
Security Trimming 278
URL Mapping 278
Summary 282
Chapter 12 Quick Reference 283

13 Personalization 285
Personalizing Web Visits 285
Personalization in ASP.NET 286
User Profiles 286
Personalization Providers 286

Using Personalization 287
Defining Profiles in Web.Config 287
Using Profile Information 287
Saving Profile Changes 288
Profiles and Users 289
Summary 294
Chapter 13 Quick Reference 294

Part III Caching and State Management

14 Session State 297
Why Session State? 297
ASP.NET and Session State 298
Introduction to Session State 299
Session State and More Complex Data 304
Configuring Session State 311
Turning Off Session State 312
Storing Session State *inProc* 313
Storing Session State in a State Server 313
Storing Session State in a Database 314
Tracking Session State 314
Tracking Session State with Cookies 314
Tracking Session State with the URL 316
Using AutoDetect 316
Applying Device Profiles 316
Session State Timeouts 317
Other Session Configuration Settings 317
The *Wizard* Control: Alternative to Session State 317
Summary 326
Chapter 14 Quick Reference 327

15 Application Data Caching 329
Using the Data Cache 331
Impact of Caching 333
Managing the Cache 335
DataSets in Memory 336
Cache Expirations 338
Cache Dependencies 341

The SQL Server Dependency 344
Clearing the Cache 345
Summary 348
Chapter 15 Quick Reference 349

16 Caching Output 351
Caching Page Content 351
Managing Cached Content 354
Modifying the *OutputCache* Directive 354
The *HTTPCachePolicy* 360
Caching Locations 361
Output Cache Dependencies 362
Caching Profiles 362
Caching User Controls 363
When Output Caching Makes Sense 366
Summary 367
Chapter 16 Quick Reference 368

Part IV Diagnostics and Plumbing

17 Diagnostics and Debugging 371
Page Tracing 371
Turning on Tracing 372
Trace Statements 375
Application Tracing 379
Enabling Tracing Programmatically 381
The *TraceFinished* Event 382
Piping Other Trace Messages 382
Debugging with Visual Studio 383
Error Pages 386
Unhandled Exceptions 390
Summary 391
Chapter 17 Quick Reference 392

18 The *HttpApplication* Class and HTTP Modules 395
The Application: A Rendezvous Point 395
Overriding *HttpApplication* 397
Application State Caveats 399

Handling Events 399
HttpApplication Events 400
HttpModules 404
Existing Modules 404
Implementing a Module 406
See Active Modules 408
Storing State in Modules 410
Global.asax versus *HttpModules* 414
Summary 414
Chapter 18 Quick Reference 415

19 Custom Handlers 417
Handlers 417
Built-in Handlers 419
IHttpHandler 422
Handlers and Session State 427
Generic Handlers (ASHX Files) 428
Summary 430
Chapter 19 Quick Reference 431

Part V Services, AJAX, Deployment, and Silverlight

20 ASP.NET Web Services 435
Remoting 435
Remoting over the Web 437
SOAP 437
Transporting the Type System 437
Web Service Description Language 438
If You Couldn't Use ASP.NET... 438
A Web Service in ASP.NET 439
Consuming Web Services 446
Asynchronous Execution 451
Evolution of Web Services 454
Other Features 455
Summary 455
Chapter 20 Quick Reference 456

21 Windows Communication Foundation 457

Distributed Computing Redux 457
A Fragmented Communications API 458
WCF for Connected Systems 458
WCF Constituent Elements 459
WCF Endpoints 459
Channels 460
Behaviors 460
Messages 461
How WCF Plays with ASP.NET 462
Side-by-Side Mode 462
ASP.NET Compatibility Mode 462
Writing a WCF Service 463
Building a WCF Client 469
Summary 475
Chapter 21 Quick Reference 476

22 AJAX 477

What Is AJAX? 478
AJAX Overview 479
Reasons to Use AJAX 480
Real-World AJAX 481
AJAX in Perspective 481
ASP.NET Server-Side Support for AJAX 482
ScriptManager Control 482
ScriptManagerProxy Control 482
UpdatePanel Control 483
UpdateProgress Control 483
Timer Control 483
AJAX Client Support 483
ASP.NET AJAX Control Toolkit 484
Other ASP.NET AJAX Community-Supported Stuff 485
AJAX Control Toolkit Potpourri 486
Getting Familiar with AJAX 487
The *Timer* 493
Updating Progress 501

Extender Controls 505
The *AutoComplete* Extender 505
A Modal Pop-up Dialog-Style Component 512
Summary 516
Chapter 22 Quick Reference 517

23 ASP.NET and WPF Content 519
What Is WPF? 519
How Does It Relate to the Web? 521
Loose XAML files 522
XBAP Applications 523
WPF Content and Web Applications 523
What about Silverlight? 529
Summary 529
Chapter 23 Quick Reference 530

24 How Web Application Types Affect Deployment 531
Visual Studio Projects 531
HTTP Project 532
FTP Project 532
File System Project 532
Precompiling 533
Precompiling for Performance 533
Precompiling for Deployment 534
Publishing a Web Site 542
Summary 543
Chapter 24 Quick Reference 544

Glossary 545

Index 547

Chapter 22

AJAX

After completing this chapter, you will be able to

- Understand the problem AJAX solves
- Understand ASP.NET's support for AJAX
- Write AJAX-enabled Web sites
- Take advantage of AJAX as necessary to improve the user's experience

This chapter covers AJAX, possibly the most interesting feature added to ASP.NET recently. AJAX stands for "Asynchronous JavaScript and XML," and it promises to produce an entirely new look and feel for Web sites throughout the world.

Software evolution always seems to happen in this typical fashion: Once a technology is grounded firmly (meaning the connections between the parts work and the architecture is fundamentally sound), upgrading the end user's experience becomes a much higher priority. AJAX's primary reason for existence is to improve on the standard HTTP GET/POST idiom with which Web users are so familiar. That is, the standard Web protocol in which entire forms and pages are sent between the client and the server is getting a whole new addition.

Although standard HTTP is functional and well understood by Web developers, it does have certain drawbacks—the primary one being that the user is forced to wait for relatively long periods while pages refresh. AJAX introduces technology that shields end users from having to wait for a whole page to post. This has been a common problem within all event-driven interfaces (Microsoft Windows being one of the best examples).

Think back to the way HTTP normally works. When you make a request (using GET or POST, for example), the Web browser sends the request to the server, but you can do nothing until the request finishes. That is, you make the request and wait—watching the little thermometer on the browser fill up. Once the request returns to the browser, you may begin using the application again. The application is basically useless until the request returns. In some cases, the browser's window even goes completely blank. Web browsers have to wait for Web sites to finish an HTTP request—in much the same way that Windows programs have to wait for message handlers to complete their processing. (Actually, if the client browser uses a multithreaded user interface such as Microsoft Internet Explorer, you can usually cancel the request—but that's all you can really do.) You can easily demonstrate this problem for yourself by introducing a call to *System.Threading.Thread.Sleep* inside the *Page_Load* method. Putting the thread to sleep will force the end user to wait for the request to finish.

The solution to this problem is to introduce some way to handle the request asynchronously. What if there were a way to introduce asynchronous background processing into a Web site so that the browser would appear much more responsive to the user? What if (for certain applications) making an HTTP request didn't stall the entire browser for the duration of the request, but instead seemed to run the request in the background, leaving the foreground unhindered and changing only the necessary portion of the rendered page? The site would present a much more continuous and smooth look and feel to the user. As another example, what if ASP.NET included some controls that injected script into the rendered pages that modified the HTML Document Object Model, providing more interaction from the client's point of view? Well, that's exactly what ASP.NET's AJAX support was designed to do.

What Is AJAX?

AJAX formalizes a style of programming meant to improve the user interface (UI) responsiveness and visual appeal of Web sites. Many of AJAX's capabilities have been available for a while now. AJAX consolidates several good ideas and uses them to define a style of programming and extends the standard HTTP mechanism that is the backbone of the Internet. Like most Web application development environments, ASP.NET has leveraged HTTP in a very standard way. The browser usually initiates contact with the server using an HTTP GET request, followed by any number of POSTs. The high-level application flow is predicated upon sending a whole request and then waiting for an entire reply from the server. Although ASP .NET's server-side control architecture greatly improves back-end programming, users still get their information a whole page at a time. It's almost like the mainframe/terminal model popular during the 1970s and early 1980s. However, this time the terminal is one of many modern sophisticated browsers and the mainframe is replaced by a Web server (or Web Farm).

The standard HTTP round-trip has been a useful application strategy, and the Web grew up using it. While the Web was growing up in the late 1990s, browsers had widely varying degrees of functionality. For example, browsers ranged all the way from the very rudimentary America Online Browser (which had very limited capabilities) to cell phones and PDAs, to more sophisticated browsers such as Microsoft Internet Explorer and Netscape Navigator that were very rich in capability. For instance, Internet Explorer supports higher level features such as JavaScript and Dynamic HTML. This made striking a balance between usability of your site and the reach of your site very difficult prior to the advent of ASP.NET.

However, being able to run a decent browser that understands how to process client-side scripting is almost a given for the majority of modern computing platforms. These days, most computing platforms run a modern operating system (such as Microsoft Windows XP or Microsoft Vista, or even MAC OS X). These platforms run browsers fully capable of supporting XML and JavaScript. With so many Web client platforms supporting this functionality, it makes sense to take advantage of the capabilities. As we'll see in this chapter, AJAX makes good use of these modern browser features to improve the user's experience.

In addition to extending standard HTTP, AJAX is also a very clever way to use the Web service idiom. Web services are traditionally geared toward enterprise-to-enterprise business communications. However, Web services are also useful on a smaller scale for handling Web requests out of band. ("Out of band" simply means making HTTP requests using means other than the standard page posting mechanism.) AJAX uses Web services behind the scenes to make the client UI more responsive than when using traditional HTTP GETs and POSTs. We'll see how that works in this chapter—especially when we look at the ASP.NET AJAX Control Toolkit Extender controls.

AJAX Overview

One of the primary changes AJAX brings to Web programming is that it depends on the browser taking an even more active role in the process. Instead of the browser simply rendering streams of HTML and executing small custom-written script blocks, AJAX includes some new client-script libraries to facilitate the asynchronous calls back to the server. AJAX also includes some basic server-side components to support these new asynchronous calls coming from the client. There's even a community-supported AJAX Control Toolkit available for ASP.NET's AJAX implementation. Figure 22-1 shows the organization of ASP.NET's AJAX support.

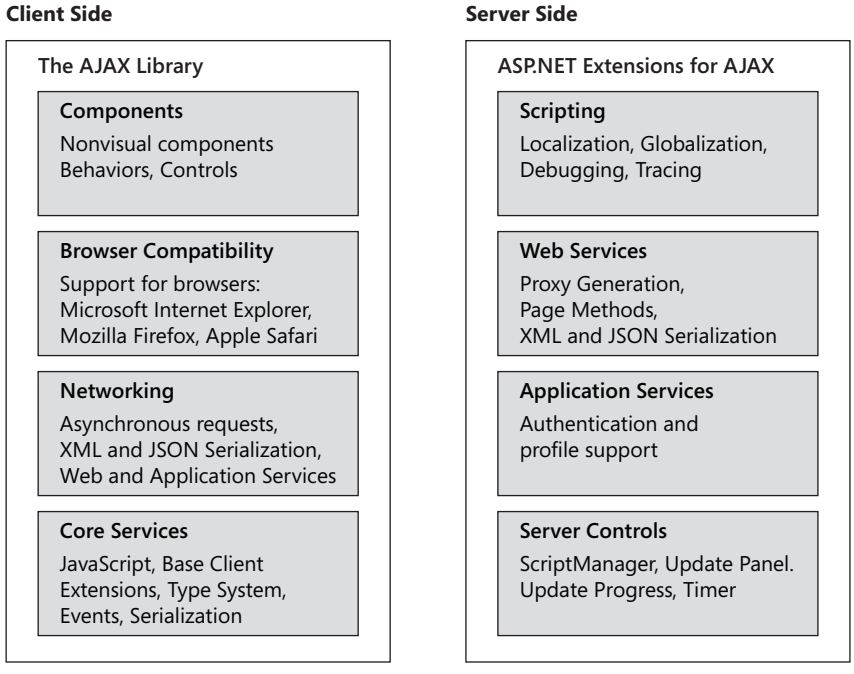


FIGURE 22-1 The conceptual organization of ASP.NET's AJAX support layers

Reasons to Use AJAX

If traditional ASP.NET development is so entrenched and well established, then why would you want to introduce AJAX? At first glance, AJAX seems to introduce some new complexities into the ASP.NET programming picture. In fact, it seems to re-introduce some programming idioms that ASP.NET was designed to deprecate (such as overuse of client-side script). However, AJAX promises to produce a richer experience for the user. Because ASP.NET's support for AJAX is nearly seamless, the added complexities are well mitigated. When building a Web site, there are a few reasons you might choose to AJAX-enable the site.

- AJAX improves the overall efficiency of your site by performing parts of a Web page's processing in the browser when appropriate. Instead of waiting for the entire HTTP protocol to get a response from the browser, pushing certain parts of the page processing to the client helps the client to react much more quickly. Of course, this type of functionality has always been available—as long as you're willing to write the code to make it happen. ASP.NET's AJAX support includes a number of scripts so that you can get a lot of browser-based efficiency by simply using a few server-side controls.
- AJAX introduces UI elements usually found in desktop applications to a Web site. These UI elements include such items as rectangle rounding, callouts, progress indicators, and pop-up windows that work for a wide range of browsers (more browser-side scripting—but most of it's been written for you).
- AJAX introduces partial-page updates. By refreshing only the parts of the Web page that have been updated, the user's wait time is reduced significantly. This brings Web-based applications much closer to desktop applications with regard to perceived UI performance.
- AJAX is supported by most popular browsers—not just Microsoft Internet Explorer. It works for Mozilla Firefox and Apple Safari, too. Although it still requires some effort to strike a balance between UI richness and the ability to reach a wider audience, the fact that AJAX depends on features available in most modern browsers makes this balance much easier to achieve.
- AJAX introduces a huge number of new capabilities. Whereas standard ASP.NET's control and page-rendering model provides great flexibility and extensibility for programming Web sites, AJAX brings in a new concept—the extender control. Extender controls attach to existing server-side controls (such as the *TextBox, ListBox*, and *DropDownList*) at run time and add new client-side appearances and behaviors to the controls. Sometimes extender controls can even call a predefined Web service to get data to populate list boxes and such (for example, the *AutoComplete* extender).
- AJAX improves on ASP.NET's forms authentication and profiles and personalization services. ASP.NET's support for authentication and personalization provided a great boon to Web developers—and AJAX just sweetens the pot.

Today when you browse different Web sites, you'll run into lots of examples of AJAX-style programming. Here are some examples:

- Colorado Geographic: *http://www.coloradogeographic.com/*
- Cyber Homes: *http://www.cyberhomes.com/default.aspx?AspxAutoDetectCookieSupport=1&bhcp=1*
- Component Art: *http://www.componentart.com/*

Real-World AJAX

Throughout the 1990s and into the mid-2000s, Web applications were nearly a throwback to 1970s mainframe and minicomputer architectures. However, instead of finding a single large computer serving dumb terminals, Web applications consist of a Web server (or a Web Farm) connected to smart browsers capable of fairly sophisticated rendering capabilities. Until recently, Web applications took their input via HTTP forms and presented output via HTML pages. The real trick in understanding standard Web applications is to see the disconnected and stateless nature of HTTP. Classic Web applications can only show a snapshot of the state of the application.

As we'll see in this chapter, Microsoft supports standard AJAX idioms and patterns within its ASP.NET framework. However, AJAX is more a style of Web programming involving out-of-band HTTP requests than any specific technology.

You've no doubt seen sites engaging the new interface features and stylings available through AJAX programming. Examples include Microsoft.com, Google.com, and Yahoo.com. Very often while browsing these sites, you'll see modern features such as automatic page updates without you having to generate a postback explicitly. Modal-type dialog boxes requiring your attention will pop up until you dismiss them. These are all features available through AJAX-style programming patterns, and ASP.NET has lots of new support for it.

If you're a long-time Microsoft-platform Web developer, you may be asking yourself whether AJAX is something really worthwhile or whether you might be able to get much of the same type of functionality using a tried and true technology like DHTML.

AJAX in Perspective

Any seasoned Web developer targeting Microsoft Internet Explorer as the browser is undoubtedly familiar with Dynamic HTML (DHTML). DHTML is a technology running at the browser for enabling Windows desktop-style UI elements into the Web client environment. DHTML was a good start, and AJAX brings the promise of more desktop-like capabilities to Web applications.

AJAX makes available wider capabilities than simply using DHTML. DHTML is primarily about being able to change the style declarations of an HTML element through JavaScript. However, that's about as far as it goes. DHTML is very useful for implementing such UI features as having a menu drop down when the mouse is rolled over it. AJAX expands on this idea of client-based UI using JavaScript as well as out-of-band calls to the server. Because AJAX is based on out-of-band server requests (rather than relying *only* on a lot of client script code), AJAX has the potential for much more growth in terms of future capabilities than DHTML.

AJAX represents another level in client-side performance for Web application. Through AJAX, Web sites can now support features such as partial page updates, ToolTips and pop-up windows, and data-driven UI elements (that get their data from Web services).

ASP.NET Server-Side Support for AJAX

Much of ASP.NET's support for AJAX resides in a collection of server-side controls responsible for rendering AJAX-style output to the browser. Recall from Chapter 3 on the page rendering model that the entire page-rendering process of an ASP.NET application is broken down into little bite-sized chunks. Each individual bit of rendering is handled by a class derived from *System.Web.UI.Control*. The entire job of a server-side control is to render output that places HTML elements in the output stream so they appear correctly in the browser. For example, *ListBox* controls render a <*select/*> tag. *TextBox* controls render an <*input type="text" /*> tag. ASP.NET's AJAX server-side controls render AJAX-style script along with HTML to the browser.

ASP.NET's AJAX support consists of these server-side controls along with client code scripts that integrate to produce AJAX-like behavior. Here's a description of the most frequently used official ASP.NET AJAX server controls: *ScriptManager*, *ScriptManagerProxy*, *UpdatePanel*, *UpdateProgress*, and *Timer*.

ScriptManager Control

The *ScriptManager* control manages script resources for the page. The *ScriptManager* control's primary action is to register the AJAX Library script with the page so the client script may use type system extensions. The *ScriptManager* also makes possible partial-page rendering and supports localization as well as custom user scripts. The *ScriptManager* assists with out-of-band calls back to the server. Any ASP.NET site wishing to use AJAX must include an instance of the *ScriptManager* control on any page using AJAX functionality.

ScriptManagerProxy Control

Scripts on a Web page often require a bit of special handling in terms of how the server renders them. Normally, the page uses a *ScriptManager* control to organize the scripts at

the page level. Nested components such as content pages and User controls require the *ScriptManagerProxy* to manage script and service references to pages that already have a *ScriptManager* control.

This is most notable in the case of Master Pages. The Master Page typically houses the *ScriptManager* control. However, ASP.NET will throw an exception if a second instance of *ScriptManager* is found within a given page. So what would content pages do if they needed to access the *ScriptManager* control that the Master Page contains? The answer is that the content page should house the *ScriptManagerProxy* control and work with the true *ScriptManager* control via the proxy. Of course, as mentioned, this also applies to User controls as well.

UpdatePanel Control

The *UpdatePanel* control supports partial page updates by tying together specific server-side controls and events that cause them to render. The *UpdatePanel* control causes only selected parts of the page to be refreshed instead of refreshing the whole page (as happens during a normal HTTP postback).

UpdateProgress Control

The *UpdateProgress* control coordinates status information about partial-page updates as they occur within *UpdatePanel* controls. The *UpdateProgress* control supports intermediate feedback for long-running operations.

Timer Control

The *Timer* control will issue postbacks at defined intervals. Although the *Timer* control will perform a normal postback (posting the whole page), it is especially useful when coordinated with the *UpdatePanel* control to perform periodic partial-page updates.

AJAX Client Support

ASP.NET's AJAX client-side support is centered around a set of JavaScript libraries. The following layers are included in the ASP.NET AJAX script libraries:

- The browser compatibility layer for assisting in managing compatibility across the most frequently used browsers. Whereas ASP.NET by itself implements browser capabilities on the server end, this layer handles compatibility on the client end (the browsers supported include Internet Explorer, Mozilla Firefox, and Apple Safari).

- The ASP.NET AJAX core services layer extends the normal JavaScript environment by introducing classes, namespaces, event handling, data types, and object serialization that are useful in AJAX programming.
- The ASP.NET AJAX base class library for clients includes various components, such as components for string management and for extended error handling.
- The networking layer of the AJAX client-side support manages communication with Web-based services and applications. The networking layer also handles asynchronous remote method calls.

The piece de resistance of ASP.NET's AJAX support is the community-supported Control Toolkit. Although everything mentioned previously provides solid infrastructure for ASP.NET AJAX, AJAX isn't really compelling until you add a rich tool set.

ASP.NET AJAX Control Toolkit

The ASP.NET AJAX Control Toolkit is a collection of components (and samples showing how to use them) encapsulating AJAX's capabilities. When you browse through the samples, you can get an idea of the kind of user experiences available through the controls and extenders. The Control Toolkit also provides a powerful software development kit for creating custom controls and extenders. You can download the ASP.NET AJAX Control Toolkit from the ASP.NET AJAX Web site.

The AJAX Control Toolkit is a separate download and not automatically included with Visual Studio 2008. To use the controls in the toolkit, follow these steps:

1. Download the tool. There are two versions—2.0 and 3.5. Version 3.5 is the most up to date and requires .NET 3.5 on your development machine. (See *http://asp.net/ajax/ajaxcontroltoolkit/* for details.)
2. After unzipping the Toolkit file, open the AjaxControlToolkit solution file in Visual Studio.
3. Build the AjaxControlKit project.
4. The compilation process will produce a file named *AjaxControlToolkit.dll* in the AjaxControlToolkit\bin directory.
5. Click the right mouse button on the Toolbox in Visual Studio, select **Choose Items...** from the menu. Browse to the AjaxControlToolkit.dll file in the AjaxControlToolkit\bin directory and include the DLL. This will bring all the new AJAX Controls from the toolkit into Visual Studio so you may drag and drop them onto forms in your applications.

Other ASP.NET AJAX Community-Supported Stuff

Although not quite officially part of AJAX, you'll find a wealth of AJAX-enabled server-side controls and client-side scripts available through a community-supported effort. The support includes ASP.NET AJAX community-supported controls (mentioned previously) as well as support for client declarative syntax (XML-script) and more.

AJAX Control Toolkit Potpourri

There are a number of other extenders and controls available through a community-supported effort. You can find a link to the AJAX Control Toolkit through *http://asp.net/ajax/.* We'll see a few of the controls available from the toolkit throughout this chapter. Table 22-1 lists the controls and extenders available through this toolkit.

TABLE 22-1 The ASP.NET Control Toolkit

Component	Description
Accordion	This extender is useful for displaying a group of panes one pane at a time. It's similar to using several *CollapsiblePanels* constrained to allow only one to be expanded at a time. The *Accordion* is composed of a group of *AccordionPane* controls.
AlwaysVisibleControl	This extender is useful for pinning a control to the page so its position remains constant while content behind it moves and scrolls.
Animation	This extender provides a clean interface for animating page content.
AutoComplete	This extender is designed to communicate with a Web service to list possible text entries based on what's already in the text box.
Calendar	This extender is targeted for the *TextBox* control providing client-side date-picking functionality in a customizable way.
CascadingDropDown	This extender is targeted toward the *DropDownList* control. It functions to populate a set of related *DropDownList* controls automatically.
CollapsiblePanel	This extender is targeted toward the *Panel* control for adding collapsible sections to a Web page.
ConfirmButton	This extender is targeted toward the *Button* control (and types derived from the *Button* control) useful for displaying messages to the user. The scenarios for which this extender is useful include those requiring confirmation from the user (for example, where linking to another page might cause your end user to lose state).
DragPanel	This is an extender targeted toward *Panel* controls for adding the capability for users to drag the *Panel* around the page.
DropDown	This extender implements a SharePoint-style drop-down menu.
DropShadow	This extender is targeted toward the *Panel* control that applies a drop shadow to the *Panel*.
DynamicPopulate	This extender uses an HTML string returned by a Web service or page method call.

Continued

TABLE 22-1 Continued

Component	Description
FilteredTextBox	This extender is used to ensure that an end user enters only valid characters into a text box.
HoverMenu	This extender is targeted for any *WebControl* that associates that control with a pop-up panel for displaying additional content. It's activated when the user hovers the mouse cursor over the targeted control.
ListSearch	This extender searches items in a designated *ListBox* or *DropDownList* based on keystrokes as they're typed by the user.
MaskedEdit	This extender is targeted toward *TextBox* controls to constrain the kind of text that the *TextBox* will accept by applying a mask.
ModalPopup	This extender mimics the standard Windows modal dialog box behavior. Using the *ModalPopup*, a page may display content of a pop-up window that focuses attention on itself until it is dismissed explicitly by the end user.
MutuallyExclusiveCheckBox	This extender is targeted toward the *CheckBox* control. The extender groups *Checkbox* controls using a key. When a number of *CheckBox* controls all share the same key, the extender ensures that only a single check box will appear checked at a time.
NoBot	This control attempts to provide CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart)-like bot/spam detection and prevention without requiring any user interaction. While using a noninteractive approach may be bypassed more easily than one requiring actual human interaction, this implementation is invisible.
NumericUpDown	This extender is targeted toward the *TextBox* control to create a control very similar to the standard Windows *Edit* control with the *Spin* button. The extender adds "up" and "down" buttons for incrementing and decrementing the value in the *TextBox*.
PagingBulletedList	This extender is targeted toward the *BulletedList* control. The extender enables sorted paging on the client side.
PasswordStrength	This extender is targeted toward the *TextBox* control to help when end users type passwords. While the normal *TextBox* only hides the actual text, the *PasswordStrength* extender also displays the strength of the password using visual cues.
PopupControl	This extender is targeted toward all controls. Its purpose is to open a pop-up window for displaying additional relevant content.
Rating	This control renders a rating system from which end users rate something using images to represent a rating (stars are common).
ReorderList	This ASP.NET AJAX control implements a bulleted, data-bound list with items that can be reordered interactively.
ResizableControl	This extender works with any element on a Web page. Once associated with an element, the *ResizableControl* gives the user the ability to resize that control. The *ResizableControl* puts a handle on the lower right corner of the control.

Continued

TABLE 22-1 Continued

Component	Description
RoundedCorners	The *RoundedCorners* extender may be applied to any Web page element to turn square corners into rounded corners.
Slider	This extender is targeted to the *TextBox* control. It adds a graphical slider that the end user may use to change the numeric value in the *TextBox*.
SlideShow	This extender controls and adds buttons to move between images individually and to play the slide show automatically.
Tabs	This server-side control manages a set of tabbed panels for managing content on a page.
TextBoxWatermark	*TextBoxWatermark* extends the *TextBox* control to display a message while the *TextBox* is empty. Once *TextBox* contains some text, the *TextBox* appears as a normal *TextBox*.
ToggleButton	This extender extends the *CheckBox* to show custom images reflecting the state of the *CheckBox*.
UpdatePanelAnimation	This extender provides a clean interface for animating content associated with an *UpdatePanel*.
ValidatorCallout	*ValidatorCallout* extends the validator controls (such as *RequiredFieldValidator* and *RangeValidator*). The callouts are small pop-up windows that appear near the UI elements containing incorrect data to direct user focus toward them.

Getting Familiar with AJAX

Here's a short example to help get you familiar with AJAX. It's a very simple Web Forms application that shows behind-the-scenes page content updates with the *UpdatePanel* server-side control. In this exercise, you'll create a page with labels showing the date and time that the page loads. One label will be outside the *UpdatePanel*, and the other label will be inside the *UpdatePanel*. You'll be able to see how partial page updates work by comparing the date and time shown in each label.

A simple partial page update

1. Create a new Web site project named *AJAXORama*. Make it a file system Web site. Earlier versions of the AJAX toolkit (for Visual Studio 2005) required a special "AJAX Enabled Web site" template. The template inserted specific entries into the configuration file necessary for AJAX to work. Visual Studio 2008 creates "AJAX Enabled " projects right off the bat. Make sure the default.aspx file is open.
2. Add a *ScriptManager* control to the page. Pick one up off the Toolbox and drop it on the page (you'll find it under a different tab in the toolbox than the normal control tab.). Using the AJAX controls requires a *ScriptManager* to appear prior to any other AJAX controls on the page. By convention, the control is usually placed outside the DIV

Visual Studio creates for you. After placing the script manager control on your page, the <body> element in the *Source* view should look like so:

```
<body>
    <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server">
    </asp:ScriptManager>
    <div>

    </div>
    </form>
</body>
```

3. Drag a *Label* control into the Default.aspx form. From the Properties window, give the *Label* control the name *LabelDateTimeOfPageLoad*.Then drop a *Button* on the form as well. Give it the text *Click Me*. Open the code beside file (default.aspx.cs) and update the *Page_Load* handler to have the label display the current date and time.

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        this.LabelDateTimeOfPageLoad.Text = DateTime.Now.ToString();
    }
}
```

4. Run the page and generate some postbacks by clicking the button a few times. Notice that the label on the page updates with the current date and time each time the button is clicked.

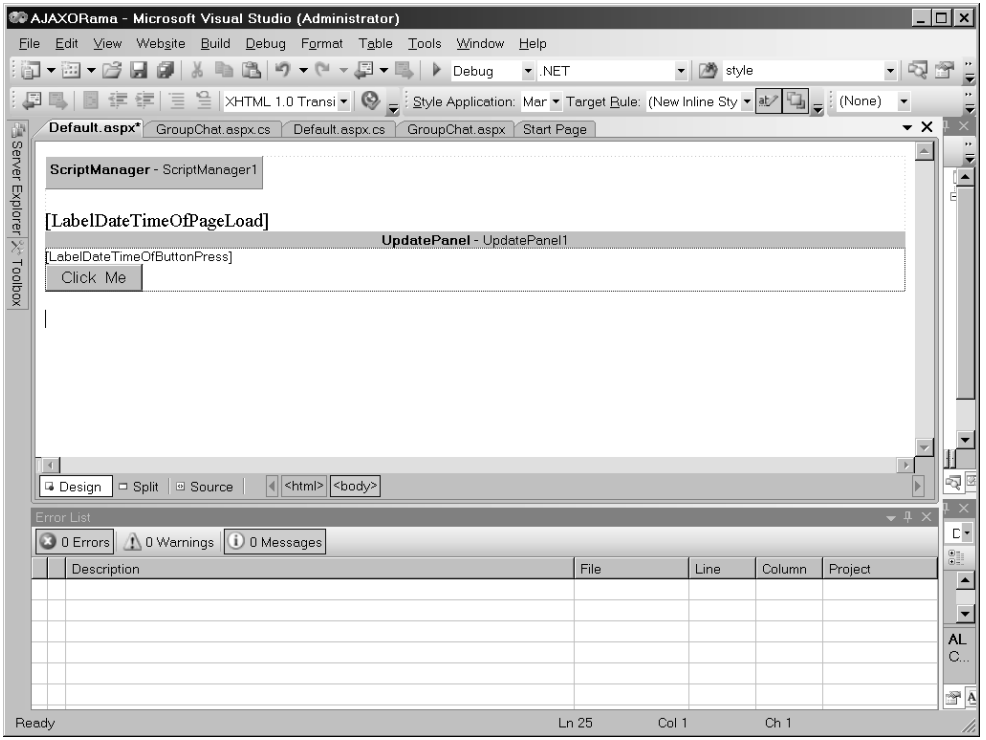
5. Add an *UpdatePanel* control to the page (you'll find this control alongside the *ScriptManager* control in the *AJAX Control Toolkit* tab). Then pick up another *Label* from the Toolbox and drop it into the content area of the *UpdatePanel*. Name the label *LabelDateTimeOfButtonClick*.

6. Add some code to the *Page_Load* method to have the label show the current date and time.

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
```

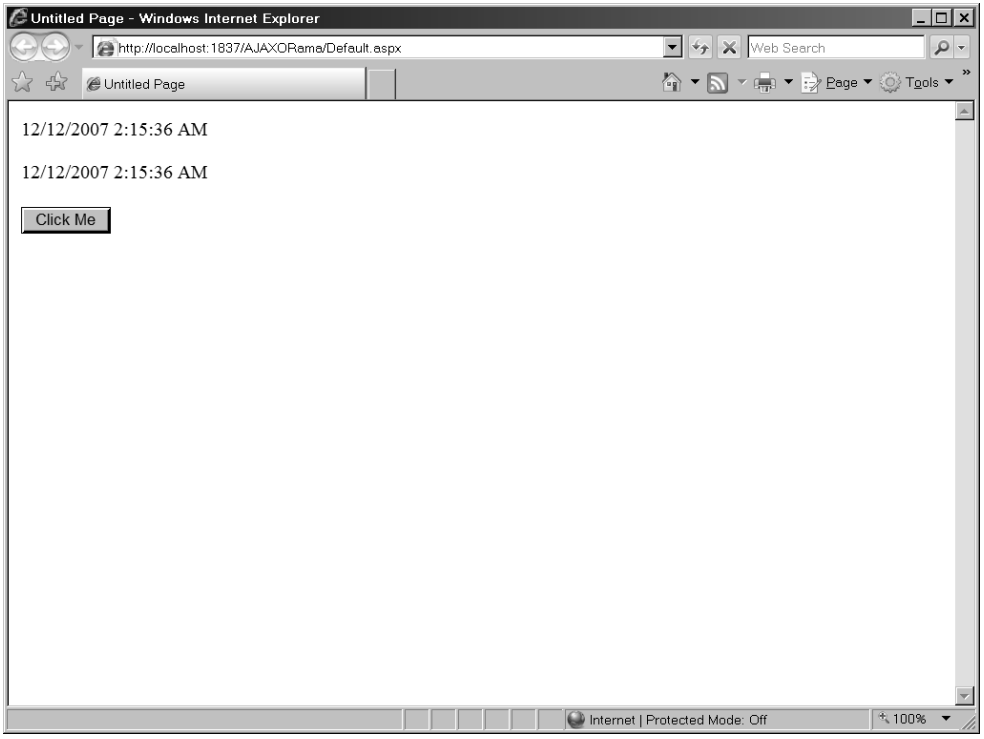
```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        this.LabelDateTimeOfPageLoad.Text = DateTime.Now.ToString();
        this.LabelDateTimeOfButtonClick.Text =
            DateTime.Now.ToString();
    }
}
```

The following graphic shows the *UpdatePanel*, *Button*, and *Labels* as seen within the Visual Studio designer (there are some line breaks in between so that the page is readable):



7. Run the page and generate some postbacks by clicking the button. Both labels should be showing the date and time of the postback (that is, they should show the same time). Although the second label is inside the *UpdatePanel*, the action causing the postback is happening outside the *UpdatePanel*.

The following graphic shows the Web page running without the *Button* being associated with the *UpdatePanel:*



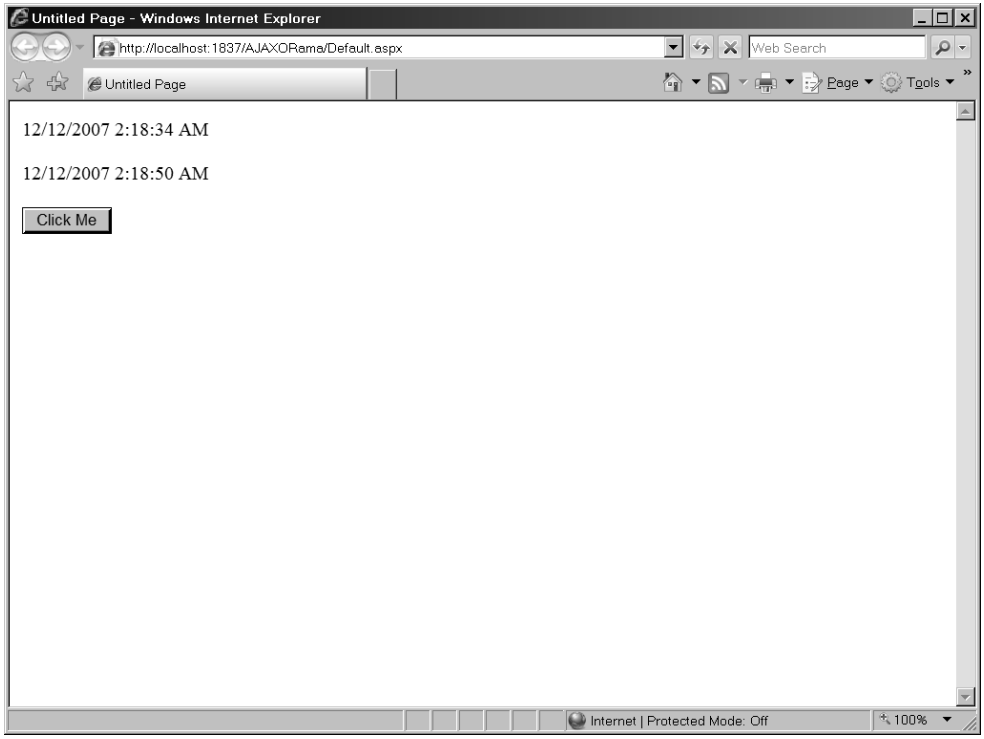
8. Now delete the current button from the form and drop a new button into the *UpdatePanel1* control. Add a *Label* to the *UpdatePanel1* as well. Name the new label *LabelDateTimeOfButtonPress*. Look at the Default.aspx file to see what was produced:

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>

<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <asp:ScriptManager
         ID="ScriptManager1" runat="server" /><br/>
        <asp:Label ID="LabelDateTimeOfPageLoad"
         runat="server"></asp:Label> <br/>
        <asp:UpdatePanel ID="UpdatePanel1" runat="server">
```

```
            <ContentTemplate>
             <asp:Label ID="LabelDateTimeOfButtonPress"
                  runat="server">
             </asp:Label><br/>
             <asp:Button ID="Button1"
                  runat="server" Text="Click Me" />
            </ContentTemplate>
        </asp:UpdatePanel>
    </form>
</body>
</html>
```

The new *Button* should now appear nested inside the *UpdatePanel* along with the new *Label*.

9. Run the page and generate some postbacks by pressing the button. Notice that only the label showing the date and time enclosed in the *UpdatePanel* is updated. This is known as a *partial page update*, since only part of the page is actually updated in response to a page action, such as clicking the button. Partial page updates are also sometimes referred to as *callbacks* rather than postbacks. The following graphic shows the Web page running with the *Button* being associated with the *UpdatePanel*.

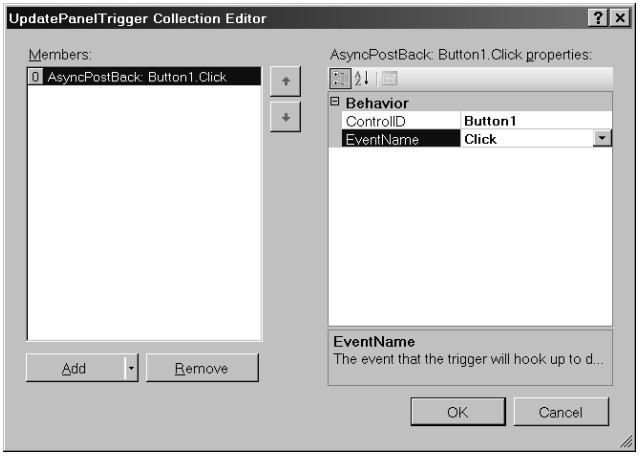


10. Add an UpdatePanel trigger. Because the second label and the button are both associated with the single *UpdatePanel*, only the second *Label* is updated in response to the postback generated by the button. If you could only set up partial page updates

based on elements tied to a single *UpdatePanel*, that would be fairly restrictive. As it turns out, the *UpdatePanel* supports a collection of triggers that will generate partial page updates. To see how this works, you need to first move the button outside the *UpdatePanel* (so that the button generates a full normal postback). The easiest way is to simply drag a button onto the form (making sure it lands outside the *UpdatePanel*).

Because the button is outside the *UpdatePanel* again, postbacks generated by the button are no longer tied solely to the second label, and the partial page update behavior you saw in Step 9 is again non-functional.

11. Update the *UpdatePanel*'s *Triggers* collection to include the *Button*'s *Click* event. With the designer open, select the *UpdatePanel*. Go to the properties Window and choose *Triggers*. This presents a dialog box as shown in the following graphic.



Add a trigger and set the control ID to the button's *ID* and the event to *Click*. (Note that the handy drop-down lists for each property assist you with this selection.) Run the page. Clicking the button should now generate a callback (causing a partial page update) in which the first label continues to show the date and time of the original page load and the second label shows the date and time of the button click. Pretty cool!

Async Callbacks

As you know by this point, standard Web pages require the browser to instigate postbacks. Many times, postbacks are generated by clicking on a *Button* control (in ASP.NET terms). However, most ASP.NET controls may be enabled to generate postbacks as well. For example, if you'd like to receive a postback whenever a user selects an item in a *DropDownList*, just flip the *AutoPostBack* property to *true*, and the control will generate the normal postback whenever the selected item changes.

In some cases, an entire postback is warranted for events such as when the selected item changes. However, in most cases generating postbacks that often will be distracting

for the user and lead to very poor performance for your page. That's because standard postbacks refresh the whole page.

ASP.NET's AJAX support introduces the notion of the "asynchronous" postback. This is done using JavaScript running inside the client page. The *XMLHttpRequest* object posts data to server—making an end run around the normal postback. The server returns data as XML, JSON, or HTML and has to refresh only part of the page. The JavaScript running in the page replaces old HTML within the Document Object Model with new HTML based on the results of the asynchronous postback.

If you've done any amount of client-side script programming, you can imagine how much work doing something like this can be. Performing asynchronous postbacks and updating pages usually requires a lot of JavaScript.

The *UpdatePanel* control you just used in this exercise hides all of the client-side code and also the server-side plumbing. Also, because of ASP.NET's well-architected server-side control infrastructure, the *UpdatePanel* maintains the same server-side control model you're used to seeing in ASP.NET.

The Timer

In addition to causing partial page updates via an event generated by a control (like a button click), AJAX includes a timer to cause regularly scheduled events. You can find the *Timer* control alongside the other standard AJAX controls in the Toolbox. By dropping a *Timer* on a page, you can generate automatic postbacks to the server.

Some uses for the *Timer* include a "shout box"—like an open chat where a number of users type in messages and they appear near the top like a conversation. Another reason you might like an automatic postback is if you wanted to update a live Web camera picture or to refresh some other frequently updated content.

The *Timer* is very easy to use—simply drop it on a page which hosts a *ScriptManager*. The default settings for the timer cause the timer to generate postbacks every minute (every 60,000 milliseconds). The *Timer* is enabled by default and begins firing events as soon as the page loads.

Here's an exercise using the *Timer* to write a simple chat page that displays messages from a number of users who are logged in. The conversation is immediately updated for the user typing in a message. However, users who have not refreshed since the last message don't get to see it—unless they perform a refresh. The page uses a *Timer* to update the conversation automatically. At first, the entire page is refreshed. Then the chat page uses an *UpdatePanel* to update only the chat log (which is the element that has changed).

Using the Timer: Creating a chat page

1. Open the AJAXORama application if it's not already open. The first step is to create a list of chat messages that can be seen from a number of different sessions. Add a global application class to the project by clicking the right mouse button in the Solution Explorer and selecting **Add New Item**. Choose *Global Application Class* as the type of file to add. This will add a file named *Global.asax* to your Web site.

2. Update the *Application_Start* method in Global.asax to create a list for storing messages and add the list to the application cache. Using an *Import* statement at the top makes it more convenient to use the generic *List* collection.

```
<%@ Application Language="C#" %>
<%@ Import Namespace="System.Collections.Generic" %>

<script runat="server">

    void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on application startup
        List<string> messages = new List<string>();
        HttpContext.Current.Cache["Messages"] = messages;
    }

    void Application_End(object sender, EventArgs e)
    {
    }

    void Application_Error(object sender, EventArgs e)
    {
    }

    void Session_Start(object sender, EventArgs e)
    {
    }

    void Session_End(object sender, EventArgs e)
    {
    }
</script>
```

3. Create a chat page by adding a new page to the Web site and calling it GroupChat.aspx. This will hold a text box with messages as they accumulate, and it also gives users a means of adding messages.

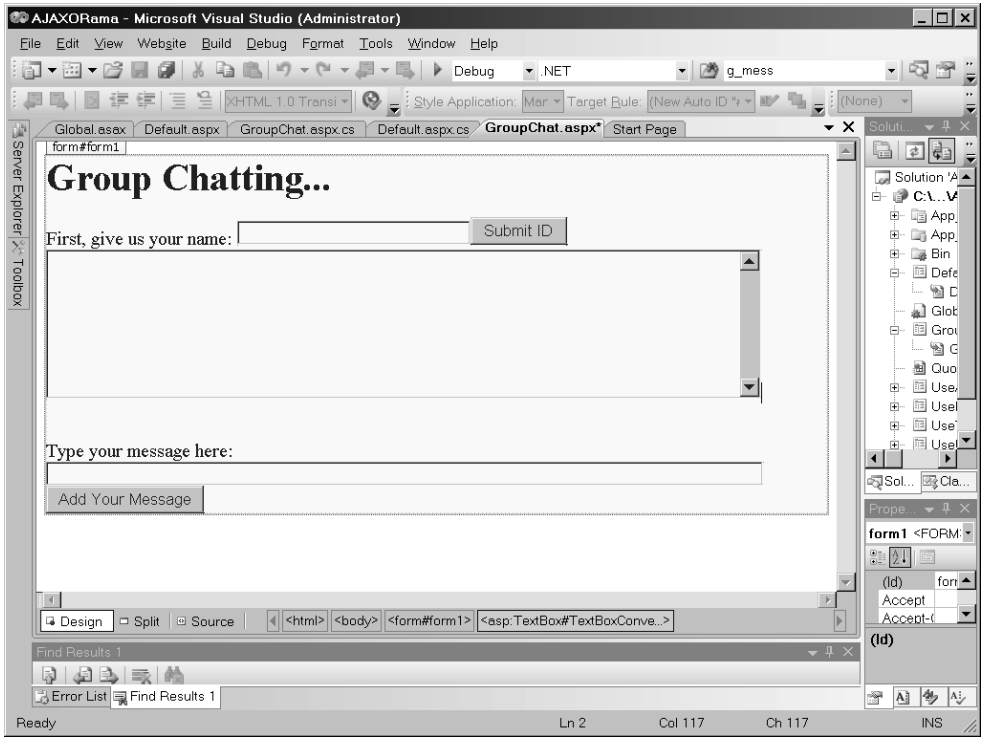
4. When the messages are coming in, it would be very useful to know who sent what messages. This page will force users to identify themselves first—then they can start adding messages. First, type in the text **Group Chatting...** following the *ScriptManager*. Give it a large font style with block display so that it's on its own line. Following that, type in the text **First, give us your name:**. Then, pick up a *TextBox* control from the Toolbox and drop it on the page. Give the *TextBox* the ID **TextBoxUserID.** Drop a

Button on the page so the user can submit his or her name. Give it the text **Submit ID** and the ID **ButtonSubmitID**.

5. Drop another *TextBox* onto the page. This one will hold the messages, so make it large (800 pixels wide by 150 pixels high should do the trick). Set the *TextBox*'s *TextMode* property to *MultiLine*, and set the *ReadOnly* property to *True*. Give the *TextBox* the ID **TextBoxConversation**.

6. Drop one more *TextBox* onto the page. This one will hold the user's current message. Give the *TextBox* the ID **TextBoxMessage**.

7. Add one more *Button* to the page. This one will let the user submit the current message and should have the text **Add Your Message**. Be sure to give the button the ID value **ButtonAddYourMessage**. The following graphic shows a possible layout of these controls.



8. Open the code beside file GroupChat.aspx.cs for editing. Add a method that retrieves the user's name from session state. Note you should also add the *using* clause for *System.Collections.Generic* as later we'll need to access the generic list we placed in the application cache (Step 2):

```
using System;
using System.Data;
using System.Configuration;
```

```
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

    protected string GetUserID()
    {
        string strUserID =
          (string) Session["UserID"];
        return strUserID;
    }
}
```

9. Add a method to update the UI so that users may only type messages after they've identified themselves. If the user has not been identified (that is, the session variable is not there), then *disable* the chat conversation UI elements and *enable* the user identification UI elements. If the user has been identified, then *enable* the chat conversation UI elements and *disable* the user identification UI elements.

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    // other code goes here...
    void ManageUI()
    {
        if (GetUserID() == null)
```

```
        {
            // if this is the first request, then get the user's ID
            TextBoxMessage.Enabled = false;
            TextBoxConversation.Enabled = false;
            ButtonAddYourMessage.Enabled = false;

            ButtonSubmitID.Enabled = true;
            TextBoxUserID.Enabled = true;
        }
        else
        {
            // if this is the first request, then get the user's ID
            TextBoxMessage.Enabled = true;
            TextBoxConversation.Enabled = true;
            ButtonAddYourMessage.Enabled = true;

            ButtonSubmitID.Enabled = false;
            TextBoxUserID.Enabled = false;
        }
    }
}
```

10. Add a *Click* event handler for the *Button* that stores the user ID (ButtonSubmitID). The method should store the user's identity in session state and then call *ManageUI* to enable and disable the correct controls.

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
   // other page code goes here...
    protected void ButtonSubmitID_Click(object sender, EventArgs e)
    {
        Session["UserID"] = TextBoxUserID.Text;
        ManageUI();
    }
}
```

11. Add a method to the page for refreshing the conversation. The code should look up the message list in the application cache and build a string that shows the messages in reverse order (so the most recent is on top). Then the method should set the conversation *TextBox*'s *Text* property to the new string (that is, the text property of the *TextBox* one showing the conversation).

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
   // other page code goes here...
    void RefreshConversation()
    {
        List<string> messages = (List<string>)Cache["Messages"];
        if (messages != null)
        {
            string strConversation = "";

            int nMessages = messages.Count;

            for(int i = nMessages-1; i >=0; i--)
            {
                string s;

                s = messages[i];
                strConversation += s;
                strConversation += "\r\n";
            }

            TextBoxConversation.Text =
                strConversation;
        }
    }
}
```

12. Add a *Click* event handler. Double-click on the *Button* and add a *Click* event handler to respond to the user submitting his or her message (ButtonAddYourMessage). The method should grab the text from the user's message *TextBox*, prepend the user's ID to it, and add it to the list of messages held in the application cache. Then the method should call *RefreshConversation* to make sure the new message appears in the conversation *TextBox*.

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
    // Other code goes here...
    protected void ButtonAddYourMessage_Click(object sender,
                                              EventArgs e)
    {
        // Add the message to the conversation...
        if (this.TextBoxMessage.Text.Length > 0)
        {
            List<string> messages = (List<string>)Cache["Messages"];
            if (messages != null)
            {
                TextBoxConversation.Text = "";

                string strUserID = GetUserID();

                if (strUserID != null)
                {
                    messages.Add(strUserID +
                        ": " +
                        TextBoxMessage.Text);
                    RefreshConversation();
                    TextBoxMessage.Text = "";
                }
            }
        }
    }
}
```

13. Update the *Page_Load* method to call *ManageUI* and *RefreshConversation*.

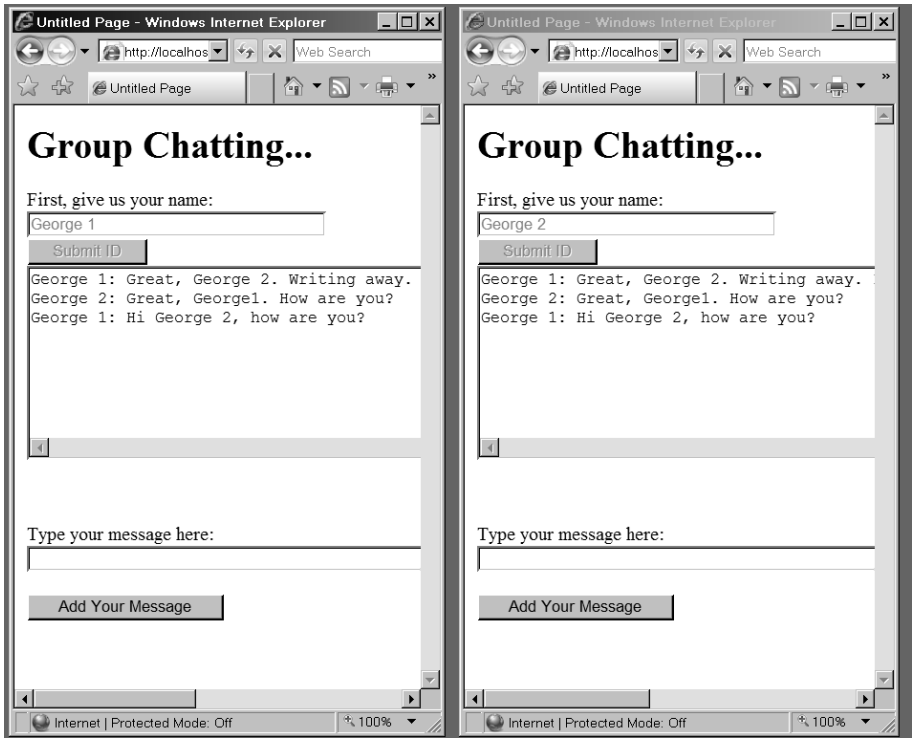
```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
```

```
using System.Xml.Linq;
using System.Collections.Generic;

public partial class GroupChat : System.Web.UI.Page
{
    // Other code goes here...
    protected void Page_Load(object sender, EventArgs e)
    {
        ManageUI();
        RefreshConversation();
    }
}
```

14. Now run the page to see how it works. Once you've identified yourself, you can start typing messages in—and you'll see them appear in the conversation *TextBox*. Try browsing the page using two separate browsers. Do you see an issue? The user typing a message gets to see the message appear in the conversation right away. However, other users involved in the chat don't see any new messages until after they submit messages of their own. Let's solve this issue by dropping an AJAX *Timer* onto the page.

15. Pick up a *ScriptManager* from the AJAX controls and drop it on the page. Then pick up a *Timer* from the AJAX controls and drop it on the page. Although the AJAX *Timer* will start generating postbacks automatically, the default interval is 60,000 milliseconds, or once per minute. Set the *Timer*'s *Interval* property to something more reasonable, such as 10,000 milliseconds (or 10 seconds). Now run both pages and see what happens. You should see the pages posting back automatically every 10 seconds. However, there's still one more issue with this scenario. If you watch carefully enough, you'll see the whole page being refreshed—even though the user name is not changing. During the conversation, you're really only interested in seeing the conversation *TextBox* being updated. Let's fix that by putting in an *UpdatePanel*.

16. Pick up an *UpdatePanel* from the AJAX controls and drop it on the page. Position the *UpdatePanel* so that it can hold the conversation text box. Move the conversation text box so that it's positioned within the *UpdatePanel*. Modify the *UpdatePanel*'s triggers so that it includes the *Timer*'s *Tick* event. Now run the chat pages, and you should see only the conversation text box being updated on each timer tick. The following graphic shows the new layout of the page employing the *UpdatePanel*.



The ASP.NET AJAX *Timer* is useful whenever you need regular, periodic posts back to the server. You can see here how it's especially useful when combined with the *UpdatePanel* doing periodic partial page updates.

Updating Progress

A recurring theme when programming any UI environment is keeping the user updated as to the progress of a long-running operation. If you're programming Windows Forms, you can use the *BackgroundWorker* component and show progress updating using the *Progress* control. Programming for the Web requires a slightly different strategy. ASP.NET's AJAX support includes a component for this—the ASP.NET AJAX *UpdateProgress* control.

UpdateProgress controls display during asynchronous postbacks. All *UpdateProgress* controls on the page become visible when any *UpdatePanel* control triggers an asynchronous postback.

Here's an exercise for using an *UpdateProgress* control on a page.

Using the *UpdateProgress* control

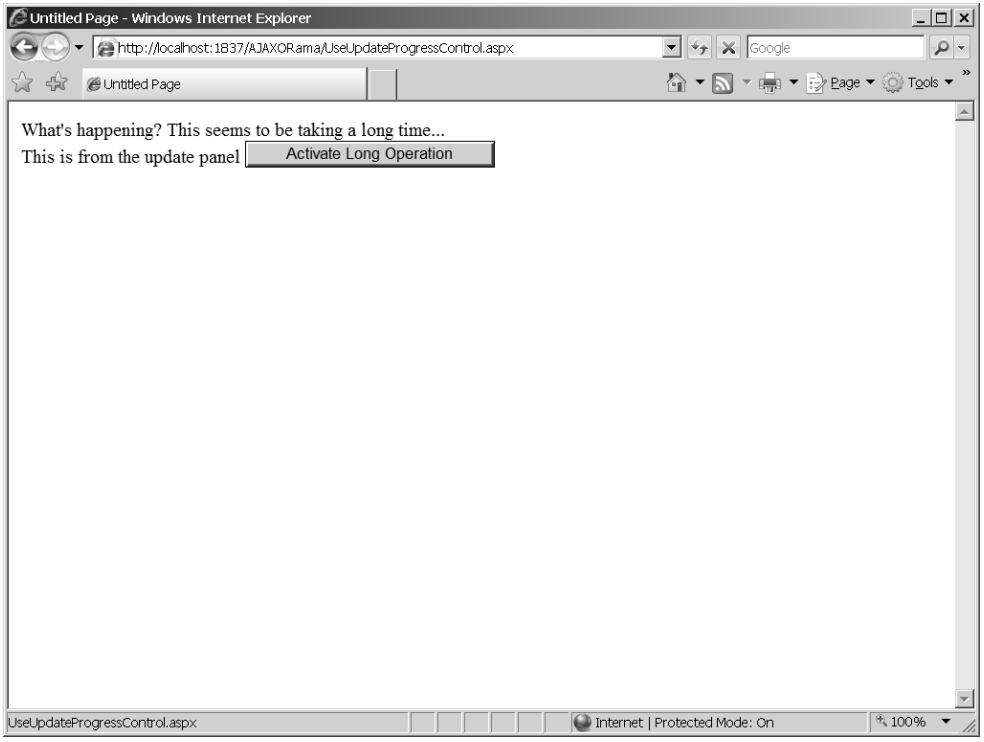
1. Add a new page. Add a new page to the AJAXORama site named *UseUpdateProgressControl.aspx*.
2. Pick up a *ScriptManager* from the Toolbox and drop it on the page.
3. Pick up an *UpdatePanel* and drop it on the page. Give the panel the ID **UpdatePanelForProgress** so you can identify it later. Add a *Button* to the update panel that will begin a long-running operation. Give it the ID **ButtonLongOperation** and the text **Activate Long Operation**.
4. Add a *Click* event handler for the button. The easiest way to create a long-running operation is to put the thread to sleep for a few seconds, as shown here. By introducing a long-running operation here, you'll have a way to test the *UpdateProgress* control and see how it works when the request takes a long time to complete.

```
public partial class UseUpdateProgressControl : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }

    protected void
        ButtonLongOperation_Click(object sender,
                                  EventArgs e)
    {
        // Put thread to sleep for five seconds
        System.Threading.Thread.Sleep(5000);
    }
}
```

5. Now add an *UpdateProgress* control to the page. An *UpdateProgress* control must be tied to a specific *UpdatePanel*. Set the *UpdateProgress* control's *AssociatedUpdatePanelID* property to the *UpdatePanelForProgress* panel you just added.
6. Add a *ProgressTemplate* to the *UpdateProgress* control—this is where the content for the update display will be declared. Add a *Label* to the *ProgressTemplate* so you will be able see it when it appears on the page.

```
<asp:UpdateProgress ID="UpdateProgress1"
    runat="server"
    AssociatedUpdatePanelID="UpdatePanelForProgress"
    DisplayAfter="100">
    <ProgressTemplate>
        <asp:Label ID="Label1" runat="server"
         Text="What's happening? This takes a long time...">
        </asp:Label>
    </ProgressTemplate>
</asp:UpdateProgress>
```

7. Run the page to see what happens. When you press the button that executes the long-running operation, you should see the *UpdateProgress* control show its content automatically. This graphic shows the *UpdateProgress* control in action.



8. Finally, no asynchronous progress updating UI technology is complete without a means to cancel the long-running operation. If you wish to cancel the long-running operation, you may do so by inserting a little of your own JavaScript into the page. You'll need to do this manually because there's no support for this using the Wizards. Write a client-side script block and place it near the top of the page—just before the <*html*> tag. The script block should get the instance of the *Sys.WebForms.PageRequestManager*. The *PageRequestManager* is a class that's available to the client as part of the script injected by the ASP.NET AJAX server-side controls. The *PageRequestManager* has a method named *get_isInAsyncPostBack()* that you can use to figure out whether the page is in the middle of an asynchronous callback (generated by the *UpdatePanel*). If the page is in the middle of an asynchronous callback, use the *PageRequestManager*'s *abortPostBack()* method to quit the request. Add a *Button* to the *ProgressTemplate* and then assign its *OnClientClick* property to make a call to your new *abortAsyncPostback* method. In addition to setting the *OnClientClick* property to the new abort method, insert **return false;** immediately following the call to the abort method, as shown in the following code (inserting "return false;" prevents the browser from issuing a postback).

```
<%@ Page Language="C#"
 AutoEventWireup="true"
CodeFile="UseUpdateProgressControl.aspx.cs"
Inherits="UseUpdateProgressControl" %>

<!DOCTYPE html PUBLIC
"...">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>

<script type="text/javascript">
  function abortAsyncPostback()
  {
    var obj =
        Sys.WebForms.PageRequestManager.getInstance();
    if(obj.get_isInAsyncPostBack())
    {
        obj.abortPostBack();
    }
  }
</script>

</head>
<body>
    <form id="form1" runat="server">
    <div>

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

    </div>
    <asp:UpdateProgress ID="UpdateProgress1"
        runat="server"
        AssociatedUpdatePanelID="UpdatePanelForProgress"
        DisplayAfter="100">
        <ProgressTemplate>
            <asp:Label ID="Label1" runat="server"
             Text="What's happening? This takes a long time...">
            </asp:Label>
            <asp:Button ID="Cancel" runat="server"
              OnClientClick="abortAsyncPostback(); return false;"
              Text="Cancel" />
        </ProgressTemplate>
    </asp:UpdateProgress>
    <asp:UpdatePanel ID="UpdatePanelForProgress" runat="server">
        <ContentTemplate>
            This is from the update panel
            <asp:Button ID="ButtonLongOperation"
              runat="server"
              onclick="ButtonLongOperation_Click"
              Text="Activate Long Operation" />
        </ContentTemplate>
    </asp:UpdatePanel>
```

```
    </form>
</body>
</html>
```

Caution *Caveat Cancel:* As you can see, canceling an asynchronous postback is completely a client-side affair. Canceling a long-running operation on the client end is tantamount to disconnecting the client from the server. Once the client is disconnected from the server, the client will never see the response from the server.

Also, while the client is happy that he or she could cancel the operation, the server may *never know* that the client canceled. So, the big caveat here is to plan for such a cancellation by making sure you program long-running blocking operations carefully so they don't spin out of control. Although IIS 6 and IIS 7 should hopefully refresh the application pool eventually for such runaway threads, it's better to depend on your own good programming practices to make sure long-running operations end reasonably nicely.

ASP.NET's AJAX support provides a great infrastructure for managing partial page updates and for setting up other events such as regular timer ticks. Now let's take a look at ASP.NET's AJAX Extender Controls.

Extender Controls

The *UpdatePanel* provided a way to update only a portion of the page. That's pretty amazing. However, AJAX's compelling features have a very broad reach. One of the most useful features is the Extender Control architecture.

Extender Controls target existing control to extend functionality in the target. While controls such as the *ScriptManager* and the *Timer* do a lot of heavy lifting in terms of injecting lots of script code into the page as it's rendered, the Extender Controls often involve managing the markup (HTML) in the resultant page.

Here are a couple of examples to familiarize you with ASP.NET AJAX Extender Controls. The first one we'll look at is the *AutoComplete* Extender.

The *AutoComplete* Extender

This extender attaches to a standard ASP.NET *TextBox*. As the end user types text into the *TextBox*, the *AutoComplete* Extender calls a Web service to look up candidate entries based on the results of the Web service call. The example borrows a component from the chapter on caching—it's the quotes collection containing a number of famous quotes by various people.

Using the *AutoComplete* extender

1. Add a new page to AJAXORama. Because this page will host the *AutoComplete* Extender, name it *UseAutocompleteExtender*.
2. Add an instance of the *ScriptManager* control to the page you just added.
3. Borrow the *QuotesCollection* class from Chapter 15. Remember, the class derives from *System.Data.Table* and holds a collection of famous quotes and their originators. You can add the component to AJAXORama by creating an App_Code directory under the project node in the Visual Studio Project Explorer, clicking the right mouse button on the *App_Code* directory, selecting **Add Existing Item**, and locating the QuotesCollection.cs file associated with the UseDataCaching example from Chapter 15.
4. Add a method to retrieve the quotes based on the last name. The method should accept the last name of the originator as a string parameter. The *System.Data.DataView* class you'll use for retrieving a specific quote is useful for performing queries on a table in memory. The method should return the quotes as a list of strings. There may be none, one, or many, depending on the selected quote author. You'll use this function shortly.

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Collections.Generic;

/// <summary>
/// Summary description for QuotesCollection
/// </summary>
public class QuotesCollection : DataTable
{
   public QuotesCollection()
   { }

   public void Synthesize()
   {
      this.TableName = "Quotations";
      DataRow dr;

      Columns.Add(new DataColumn("Quote", typeof(string)));
      Columns.Add(new DataColumn("OriginatorLastName", typeof(string)));
      Columns.Add(new DataColumn(@"OriginatorFirstName",
            typeof(string)));

      dr = this.NewRow();
      dr[0] = "Imagination is more important than knowledge.";
      dr[1] = "Einstein";
```

```
        dr[2] = "Albert";
        Rows.Add(dr);
        // Other quotes added here...
    }

    public string[]
    GetQuotesByLastName(string strLastName)
    {
        List<string> list = new List<string>();

        DataView dvQuotes = new DataView(this);
        string strFilter = String.Format("OriginatorLastName = '{0}'", strLastName)
        dvQuotes.RowFilter = strFilter;

        foreach (DataRowView drv in dvQuotes)
        {
            string strQuote =
                drv["Quote"].ToString();

            list.Add(strQuote);
        }

        return list.ToArray();
    }
}
```

5. Add a class named *QuotesManager* to the Web site's *App_Code* directory to manage caching. The Caching example from which this code was borrowed stores and retrieves the *QuotesCollection* during the *Page_Load* event. Because the *QuotesCollection* will be used within a Web service, the caching will have to happen elsewhere. To do this, add a public static method named *GetQuotesFromCache* to retrieve the *QuotesCollection* from the cache.

```
using System;
using System.Data;
using System.Configuration;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;

/// <summary>
/// Summary description for QuotesManager
/// </summary>
public class QuotesManager
{
   public QuotesManager()
   {
   }
```

```
    public static QuotesCollection GetQuotesFromCache()
    {
        QuotesCollection quotes;

        quotes =
            (QuotesCollection)HttpContext.Current.Cache["quotes"];

        if (quotes == null)
        {
            quotes = new QuotesCollection();
            quotes.Synthesize();
        }
        return quotes;
    }
}
```

6. Add an XML Web Service to your application. Click the right mouse button on the project and add an ASMX file to your application. Name the service *QuoteService*. The *WebService* and *WebServiceBinding* attributes may be removed but be sure to adorn the XML Web Service class with the *[System.Web.Script.Services.ScriptService]* attribute. That way, it will be available to the *AutoComplete* extender later on. The *AutoCompleteExtender* will use the XML Web Service to populate its drop-down list box.

7. Add a method to get the last names of the quote originators—that's the method that will populate the drop-down box. The method should take a string representing the text already typed in as the first parameter, an integer representing the maximum number of strings to return. Grab the *QuotesCollection* from the cache using the *QuoteManager*'s static method *GetQuotesFromCache*. Use the *QuotesCollection* to get the rows from the *QuotesCollection*. Finally, iterate through the rows and add the originator's last name to the list of strings to be returned if it starts with the prefix passed in as the parameter. The Common Language Runtime's (CLR) *String* type includes a method named *StartsWith* that's useful to figure out if a string starts with a certain prefix. Note you'll also have to add *using* statements for generic collections and data (as shown).

```
using System;
using System.Linq;
using System.Web;
using System.Collections;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Linq;
using System.Data;
using System.Collections.Generic;

[System.Web.Script.Services.ScriptService]
public class QuoteService : System.Web.Services.WebService
{

    [WebMethod]
    public string[]
    GetQuoteOriginatorLastNames(string prefixText,
                                int count)
```

```
    {
        List<string> list = new List<string>();

        QuotesCollection quotes =
            QuotesManager.GetQuotesFromCache();

        prefixText = prefixText.ToLower();

        foreach (DataRow dr in quotes.Rows)
        {
            string strName =
                dr["OriginatorLastName"].ToString();

            if (strName.ToLower().StartsWith(prefixText))
            {
                if (!list.Contains(strName))
                {
                    list.Add(strName);
                }
            }
        }

        return list.GetRange(0,
            System.Math.Min(count, list.Count)).ToArray();
    }
}
```

8. Now drop a *TextBox* on the *UseAutocompleteExtender* page to hold the originator's last name to be looked up. Give the *TextBox* an ID of **TextBoxOriginatorLastName**.

9. Pick up an *AutoComplete* extender from the AJAX Toolbox and add it to the page. Point the *AutoComplete*'s *TargetControlID* to the *TextBox* holding the originator's last name, **TextBoxOriginatorLastName**. Make the *MinimumPrefix* length **1**, the *ServiceMethod* **GetQuoteOriginatorLastNames**, and the *ServicePath* **quoteservice.asmx**. This wires up the *AutoComplete* extender so that it will take text from the *TextBoxOriginatorLastName TextBox* and use it to feed the XML Web Service *GetQuoteOriginatorLastNames* method.

```
<cc1:AutoCompleteExtender
   ID="AutoCompleteExtenderForOriginatorLastName"
   TargetControlID="TextBoxOriginatorLastName"
   MinimumPrefixLength="1"
   ServiceMethod="GetQuoteOriginatorLastNames"
   ServicePath="quoteservice.asmx"
   runat="server">
</cc1:AutoCompleteExtender>
```

10. Add a *TextBox* to the page to hold the quotes. Name the *TextBox TextBoxQuotes*.

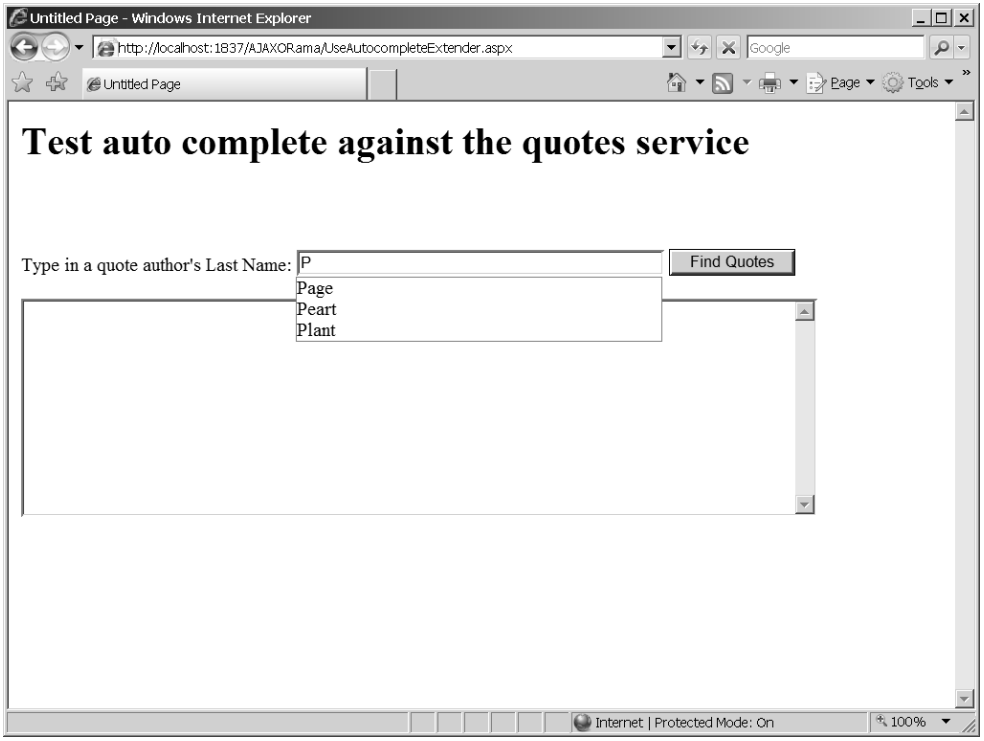
11. Update the *Page_Load* method. It should look up the quotes based on the name in the text box by retrieving the *QuoteCollection* and calling the *QuoteCollection*'s *GetQuotesByLastName* method.

```
using System;
using System.Data;
using System.Configuration;
using System.Collections;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Xml.Linq;
using System.Collections.Generic;
using System.Text;

public partial class UseAutocompleteExtender :
System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        QuotesCollection quotes =
           QuotesManager.GetQuotesFromCache();
        string[] quotesArray =
           quotes.GetQuotesByLastName(TextBoxOriginatorLastName.Text);

        if (quotesArray != null && quotesArray.Length > 0)
        {
            StringBuilder str = new StringBuilder();
            foreach (string s in quotesArray)
            {
                str.AppendFormat("{0}\r\n", s);
            }
            this.TextBoxQuotes.Text = str.ToString();
        }
        else
        {
            this.TextBoxQuotes.Text = "No quotes match your request.";
        }
    }
}
```

12. To make the page updates more efficient, drop an *UpdatePanel* onto the page. Put the *TextBox* for holding the quotes in the *UpdatePanel*. Put a button in the *UpdatePanel*. This will cause only the *TextBox* showing the quotes to be updated (instead of the whole-page refresh).

13. Add two *asynchPostBack* triggers to the *UpdatePanel*. The first trigger should connect the *TextBoxOriginatorLastName TextBox* to the *TextChanged* event. The second trigger should connect the *ButtonFindQuotes* button to the button's *Click* event.

    The following graphic shows the layout of the page using the *AutoComplete* Extender in action.



14. Run the page. As you type originator names into the *TextBox*, you should see a drop-down list appear containing candidate names based on the *QuotesCollection*'s contents.

The *AutoComplete* Extender is an excellent example of the sort of things at which ASP.NET's AJAX support excels. Microsoft Internet Explorer has had its own autocomplete feature built into it for quite a while. Microsoft Internet Explorer remembers often-used names of HTML input text tags and recent values that have been used for them. For example, when you go online to buy an airline ticket at some point and go back to buy another one later, watch what happens as you type in your address. You'll very often see Microsoft Internet Explorer's autocomplete feature show a drop-down list box below the address text box showing the last few addresses you've typed that begin with the text showing in the text box.

The ASP.NET *AutoComplete* Extender works very much like this. However, the major difference is that the end user sees input candidates generated by the Web site rather than simply a history of recent entries. Of course, the Web site could mimic this functionality by tracking a user's profile identity and store a history of what a particular user has typed in to a specific input field on a page. The actual process of generating autocomplete candidates is completely up to the Web server, giving a whole new level of power and flexibility to programming user-friendly Web sites.

A Modal Pop-up Dialog-Style Component

Another interesting feature provided by AJAX making Web applications appear more like desktop applications is the *ModalPopup* Extender. Historically, navigating a Web site involves walking down into the hierarchy of a Web site and climbing back out. When a user provides inputs as he or she works with a page, the only means available to give feedback about the quality of the data has been through the validation controls. In addition, standard Web pages have no facility to focus the users' attention while they type in the information.

The traditional desktop application usually employs modal dialog boxes to focus user attention when gathering important information from the end user. The model is very simple and elegant—the end user is presented with a situation in which he or she must enter some data and Click OK or Cancel before moving on. After dismissing the dialog, the end user sees exactly the same screen he or she saw right before the dialog appeared. There's no ambiguity and no involved process where the end user walks up and down some arbitrary page hierarchy.

This example shows how to use the pop-up dialog extender control. You'll create a page with some standard content and then have a modal dialog-style pop-up show right before submitting the page.

Using a *ModalPopup* extender

1. Add a new page to AJAXORama to host the pop-up extender. Call it *UseModalPopupExtender*.
2. As with all the other examples using AJAX controls, pick up a *ScriptManager* from the toolbox and add it to the page.
3. Add a title to the page (the example here uses "ASP.NET Code of Content"). Give the banner some prominence by surrounding it in *<h1>* and *</h1>* tags.
4. Pick up a *Panel* from the toolbox and add it to the page. It will hold the page's normal content.
5. Add a *Button* to the *Panel* for submitting the content. Give the *Button* the ID **ButtonSubmit** and the text **Submit** and create a button *Click* event handler. You'll need this button later.
6. Place some content on the panel. The content in this sample application uses several check boxes that the modal dialog pop-up will examine before the page is submitted.

```
<h1 >ASP.NET Code Of Conduct </h1>

<asp:Panel ID="Panel1" runat="server"
    style="z-index: 1;left: 10px;top: 70px;
    position: absolute;height: 213px;width: 724px;
    margin-bottom: 0px;">
```

```
    <asp:Label ID="Label1" runat="server"
        Text="Name of Developer:"></asp:Label>
         <asp:TextBox ID="TextBox1"
        runat="server"></asp:TextBox>

    <br />
    <br />
    <br />
    As an ASP.NET developer, I promise to
    <br />
    <input type="checkbox" name="Check" id="Checkbox1"/>
    <label for="Check1">Use Forms Authentication</label>
    <br />
    <input type="checkbox" name="Check" id="Checkbox2"/>
    <label for="Check2">Separate UI From Code</label>
    <br />
    <input type="checkbox" name="Check" id="Checkbox3"/>
    <label for="Check3">Take Advantage of Custom Controls</label>
    <br />
    <input type="checkbox" name="Check" id="Checkbox4"/>
    <label for="Check4">Give AJAX a try</label>
    <br />
    <asp:Button ID="ButtonSubmit" runat="server" Text="Submit"
            onclick="ButtonSubmit_Click" />
    <br />
    </asp:Panel>
```

7. Add another *Panel* to the page to represent the pop-up. Give this *Panel* a light yellow background color so that you'll be able to see it when it comes up. It should also have the ID **PanelModalPopup**.

8. Add some content to the new *Panel* that's going to serve as the modal pop-up. At the very least, the popup should have **OK** and **Cancel** buttons. Give the **OK** and **Cancel** buttons the ID values *ButtonOK* and *ButtonCancel*. You'll need them a bit later.

```
<asp:Panel ID="PanelModalPopup" runat="server"
    BorderColor="Black"
    BorderStyle="Solid"
    BackColor="LightYellow" Height="72px"
    Width="403px">
    <br />
    <asp:Label
       Text="Are you sure these are the correct entries?"
       runat="server">
    </asp:Label>
        
    <asp:Button ID="ButtonOK"
       runat="server"
       Text="OK" />
      
    <asp:Button ID="ButtonCancel"
    runat="server" Text="Cancel" />
    <br />
</asp:Panel>
```

9. Add a script block to the ASPX file. You'll need to do this by hand. Write functions to handle the **OK** and **Cancel** buttons. The example here examines check boxes to see which ones have been checked and then displays an alert to show which features have been chosen. The Cancel handler simply displays an alert saying the **Cancel** button was pressed.

```
<script type="text/javascript">

    function onOk() {
       var optionsChosen;
       optionsChosen = "Options chosen: ";

       if($get('Checkbox1').checked)
       {
         optionsChosen =
            optionsChosen.toString() +
            "Use Forms Authentication ";
       }

       if($get('Checkbox2').checked)
       {
         optionsChosen =
            optionsChosen.toString() +
            "Separate UI From Code ";
       }

       if($get('Checkbox3').checked)
       {
         optionsChosen =
            optionsChosen.toString() +
            "Take Advantage of Custom Controls ";
       }

       if($get('Checkbox4').checked)
       {
         optionsChosen =
            optionsChosen.toString() +
            "Give AJAX a try ";
       }
       alert(optionsChosen);
    }

    function onCancel() {
       alert("Cancel was pressed");
    }
</script>
```

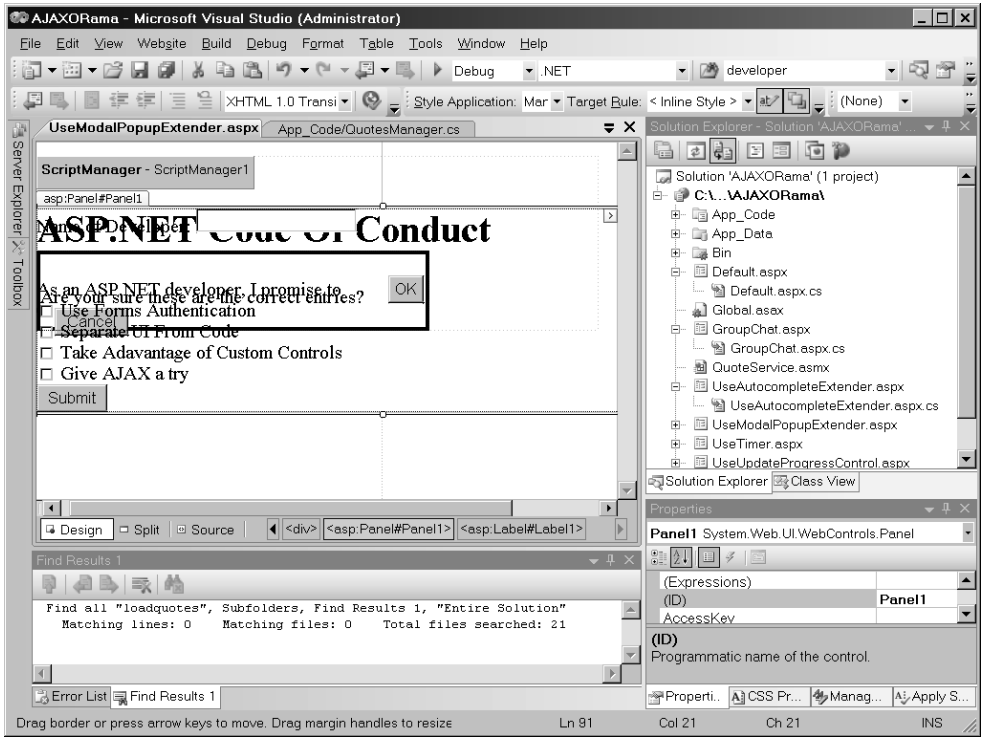
10. Pick up the *ModalPopup* Extender from the toolbox and add it to the page.

11. Add the following markup to the page. This will set various properties on the new *ModalPopup* Extender. It will set the *OkControlID* property to **ButtonOK** and it will set the *CancelControlID* property to **ButtonCancel**. It will also set the *OnCancelScript* property to **onCancel()** (the client-side Cancel script handler you just wrote). Set

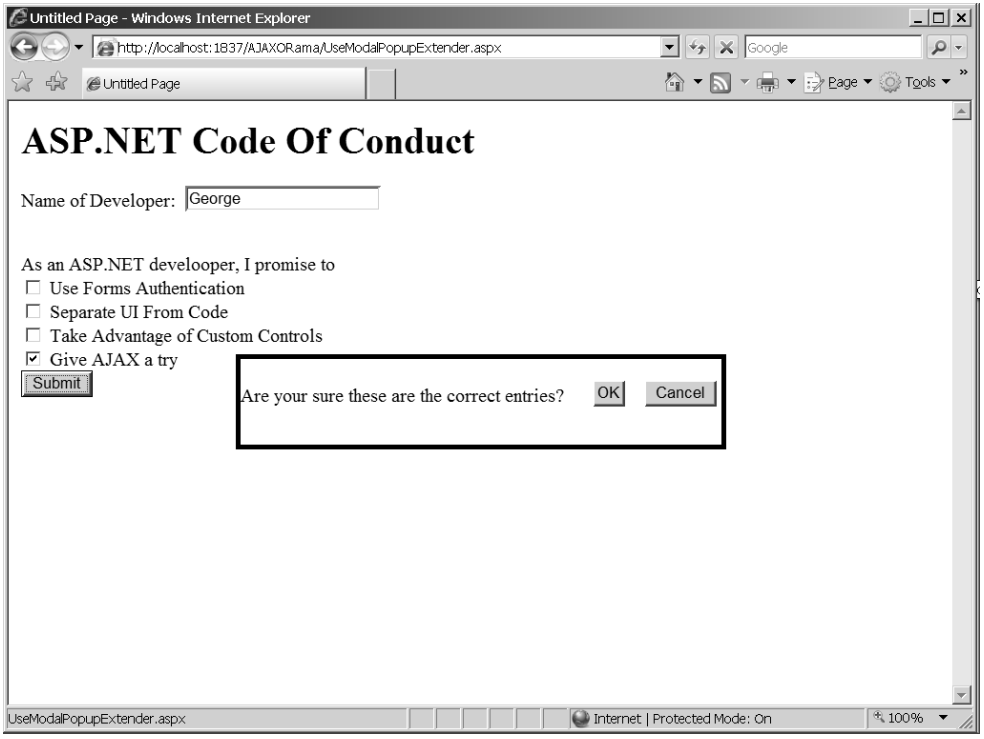
OnOkScript="onOk()" (the client-side OK script handler you just wrote). Finally, the following markup will set the *TargetControlID* property to be **ButtonSubmit**.

```
<cc1:ModalPopupExtender
    ID="ModalPopupExtender1"
    OkControlID="ButtonOK"
    CancelControlID="ButtonCancel"
    OnCancelScript="onCancel()"
    OnOkScript="onOk()"
    TargetControlID="ButtonSubmit"
    PopupControlID="PanelModalPopup">
    runat="server"
    DynamicServicePath="" Enabled="True"
</cc1:ModalPopupExtender>
```

This graphic shows the layout of the page using the *ModalPopup* Extender within Visual Studio 2008.



12. Run the page. When you click the *Submit* button, the *Panel* designated to be the modal popup window will be activated (remember, the *Submit* button is the *TargetControlID* of the *ModalPopup* Extender). When you dismiss the popup using **OK** or **Cancel**, you should see the client-side scripts being executed. The following graphic image shows the *ModalPopup* Extender displaying the modal pop-up panel.



Summary

Without a doubt, supporting AJAX is one of the most important new features of ASP.NET. Using AJAX in your ASP.NET applications helps you improve your Web site's user experience by getting rid of unnecessary postbacks and whole-page refreshes. In addition, AJAX is useful for modifying certain standard server-side controls and HTML elements to change their appearances and behaviors to seem much more "desktop-like." Although many technologies and tricks to improve the user interface experience have been around for a while (DHTML, writing your own client-side script, etc.), AJAX represents the first standard user interface technology available for targeting multiple client platforms. In addition, ASP.NET wraps these capabilities up nice and neatly so they're very convenient to use.

In this chapter, we saw how to use ASP.NET's new *UpdatePanel* to perform partial page updates. We also saw how the *Timer* produces regularly scheduled postbacks and is especially useful in conjunction with the *UpdatePanel*. We saw how the *UpdateProgress* control displays progress information asynchronously. In addition, we got to see how the *AutoComplete* Extender will talk to a Web service to produce an effective "autocomplete" experience, and we saw how the *ModalPopup* Extender allows you to show a *Panel* as though it were a modal dialog box within a desktop application.

If you feel the urge and have the gumption to look at the HTML and script produced by a page using ASP.NET AJAX controls, it's very interesting. You'll also realize the power and convenience of ASP.NET's AJAX support. It's better to have someone else have all that script code packaged within a server-side control than it is to have to write it all by hand.

Chapter 22 Quick Reference

To	Do This
Enable a Web site for AJAX	Normal Web sites generated by Visual Studio 2008's template are AJAX-enabled by default. However, you must add a *ScriptManager* to a page before using any of the AJAX server-side controls.
Implement partial page updating in your page	From within an ASP.NET project, select an *UpdatePanel* from the toolbox. Controls that you place in the *UpdatePanel* will trigger updates for only that panel, leaving the rest of the page untouched.
Assign arbitrary triggers to an *UpdatePanel* (that is, trigger partial page updates using controls and events not related to the panel)	Modify an *UpdatePanel*'s trigger collection to include the new events and controls. Highlight the *UpdatePanel* from within the Visual Studio designer. Select the *Triggers* property from within the property editor. Assign triggers as appropriate.
Implement regularly timed automatic posts from your page	Use the AJAX *Timer* control, which will cause a postback to the server at regular intervals.
Use AJAX to apply special UI nuances to your Web page	After installing Visual Studio 2008, you can create AJAX-enabled sites, and use the new AJAX-specific server-side controls available in the AJAX toolkit. Select the control you need. Most AJAX server-side controls may be programmed completely from the server. However, some controls require a bit of JavaScript on the client end.