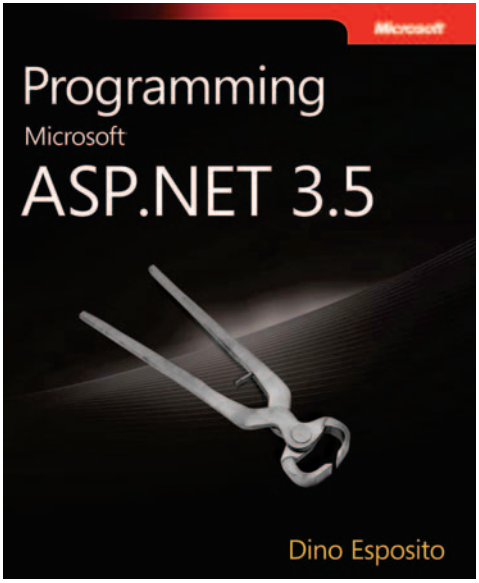

Programming Microsoft® ASP.NET 3.5

Dino Esposito

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12001.aspx

9780735625273

Table of Contents

Acknowledgements xix
Introduction xxi

Part I **Building an ASP.NET Page**

1 The ASP.NET Programming Model 3
What's ASP.NET, Anyway? 4
Programming in the Age of Web Forms 5
Event-Driven Programming over HTTP 6
The HTTP Protocol 8
Structure of an ASP.NET Page 11
The ASP.NET Component Model 15
A Model for Component Interaction 16
The *runat* Attribute 16
ASP.NET Server Controls 20
The ASP.NET Development Stack 21
The Presentation Layer 21
The Page Framework 22
The HTTP Runtime Environment 25
The ASP.NET Provider Model 28
The Rationale Behind the Provider Model 28
A Quick Look at the ASP.NET Implementation 32
Conclusion 37
Just the Facts 37

2 Web Development in Microsoft Visual Studio 2008 39
Introducing Visual Studio 2008 40
Visual Studio Highlights 40
Visual Studio 2008–Specific New Features 45
New Language Features 50

Create an ASP.NET Web Site Project . . . 55
Page Design Features . . . 55
Adding Code to the Project . . . 62
ASP.NET Protected Folders . . . 66
Build the ASP.NET Project . . . 72
Application Deployment . . . 75
XCopy Deployment . . . 75
Site Precompilation . . . 78
Administering an ASP.NET Application . . . 81
The Web Site Administration Tool . . . 82
Editing ASP.NET Configuration Files . . . 85
Conclusion . . . 87
Just the Facts . . . 88

3 Anatomy of an ASP.NET Page . . . 89
Invoking a Page . . . 89
The Runtime Machinery . . . 90
Processing the Request . . . 97
The Processing Directives of a Page . . . 102
The *Page* Class . . . 112
Properties of the *Page* Class . . . 113
Methods of the *Page* Class . . . 117
Events of the *Page* Class . . . 121
The Eventing Model . . . 122
Asynchronous Pages . . . 124
The Page Life Cycle . . . 132
Page Setup . . . 132
Handling the Postback . . . 135
Page Finalization . . . 136
Conclusion . . . 138
Just the Facts . . . 139

4 ASP.NET Core Server Controls . . . 141
Generalities of ASP.NET Server Controls . . . 142
Properties of the *Control* Class . . . 143
Methods of the *Control* Class . . . 146
Events of the *Control* Class . . . 146
Other Features . . . 147

HTML Controls . . . 153
Generalities of HTML Controls . . . 153
HTML Container Controls . . . 156
HTML Input Controls . . . 162
The *HtmlImage* Control . . . 168
Web Controls . . . 169
Generalities of Web Controls . . . 169
Core Web Controls . . . 172
Miscellaneous Web Controls . . . 179
Validation Controls . . . 184
Generalities of Validation Controls . . . 185
Gallery of Controls . . . 187
Special Capabilities . . . 192
Conclusion . . . 198
Just The Facts . . . 199

5 Working with the Page . . . 201
Programming with Forms . . . 202
The *HtmlForm* Class . . . 202
Multiple Forms . . . 205
Cross-Page Postings . . . 209
Dealing with Page Errors . . . 214
Basics of Error Handling . . . 215
Mapping Errors to Pages . . . 220
ASP.NET Tracing . . . 225
Tracing the Execution Flow in ASP.NET . . . 225
Writing Trace Messages . . . 227
The Trace Viewer . . . 229
Page Personalization . . . 230
Creating the User Profile . . . 231
Interacting with the Page . . . 234
Profile Providers . . . 241
Conclusion . . . 244
Just The Facts . . . 245

6 Rich Page Composition . . . 247
Working with Master Pages . . . 248
Authoring Rich Pages in ASP.NET 1.x . . . 248
Writing a Master Page . . . 250

Writing a Content Page 253
Processing Master and Content Pages 258
Programming the Master Page 262
Working with Themes 265
Understanding ASP.NET Themes 266
Theming Pages and Controls 270
Putting Themes to Work 273
Working with Wizards 277
An Overview of the *Wizard* Control 277
Adding Steps to a Wizard 282
Navigating Through the Wizard 285
Conclusion 290
Just the Facts 290

Part II **Adding Data in an ASP.NET Site**

7 ADO.NET Data Providers 295
.NET Data Access Infrastructure 295
.NET Managed Data Providers 296
Data Sources You Access Through ADO.NET 300
The Provider Factory Model 303
Connecting to Data Sources 307
The *SqlConnection* Class 308
Connection Strings 314
Connection Pooling 321
Executing Commands 327
The *SqlCommand* Class 327
ADO.NET Data Readers 331
Asynchronous Commands 337
Working with Transactions 342
SQL Server 2005–Specific Enhancements 347
Conclusion 352
Just The Facts 353

8 ADO.NET Data Containers 355
Data Adapters 355
The *SqlDataAdapter* Class 356
The Table-Mapping Mechanism 362

How Batch Update Works 367
In-Memory Data Container Objects 369
The *DataSet* Object 370
The *DataTable* Object 377
Data Relations 383
The *DataView* Object 386
Conclusion 389
Just The Facts 390

9 The Data-Binding Model 391
Data Source-Based Data Binding 392
Feasible Data Sources 392
Data-Binding Properties 395
List Controls 401
Iterative Controls 407
Data-Binding Expressions 413
Simple Data Binding 413
The *DataBinder* Class 416
Other Data-Binding Methods 418
Data Source Components 422
Overview of Data Source Components 422
Internals of Data Source Controls 424
The *SqlDataSource* Control 427
The *AccessDataSource* Class 433
The *ObjectDataSource* Control 434
The *LinqDataSource* Class 445
The *SiteMapDataSource* Class 456
The *XmlDataSource* Class 460
Conclusion 464
Just the Facts 465

10 The Linq-to-SQL Programming Model 467
LINQ In Brief 468
Language-Integrated Tools for Data Operations 468
A Common Query Syntax 473
The Mechanics of LINQ 482
Working with SQL Server 485
The Data Context 486

Querying for Data 490
Updating Data 498
Other Features 505
Conclusion 507
Just the Facts 508

11 Creating Bindable Grids of Data 509
The *DataGrid* Control 510
The *DataGrid* Object Model 510
Binding Data to the Grid 516
Working with the *DataGrid* 520
The *GridView* Control 524
The *GridView* Object Model 524
Binding Data to a *GridView* Control 530
Paging Data 541
Sorting Data 547
Editing Data 554
Advanced Capabilities 559
Conclusion 565
Just The Facts 566

12 Managing a List of Records 567
The *ListView* Control 567
The *ListView* Object Model 568
Defining the Layout of the List 576
Building a Tabular Layout 577
Building a Flow Layout 582
Building a Tiled Layout 584
Styling the List 590
Working with the *ListView* Control 593
In-Place Editing 594
Conducting the Update 597
Inserting New Data Items 599
Selecting an Item 603
Paging the List of Items 606
Conclusion 610
Just the Facts 610

13 Managing Views of a Record 613
The *DetailsView* Control 613
The *DetailsView* Object Model 614
Binding Data to a *DetailsView* Control 620
Creating Master/Detail Views 624
Working with Data 627
The *FormView* Control 637
The *FormView* Object Model 637
Binding Data to a *FormView* Control 639
Editing Data 642
Conclusion 645
Just The Facts 646

Part III ASP.NET Infrastructure

14 The HTTP Request Context 649
Initialization of the Application 650
Properties of the *HttpApplication* Class 650
Application Modules 651
Methods of the *HttpApplication* Class 652
Events of the *HttpApplication* Class 653
The *global.asax* File 656
Compiling *global.asax* 656
Syntax of *global.asax* 658
Tracking Errors and Anomalies 661
The *HttpContext* Class 662
Properties of the *HttpContext* Class 663
Methods of the *HttpContext* Class 665
The *Server* Object 667
Properties of the *HttpServerUtility* Class 667
Methods of the *HttpServerUtility* Class 668
The *HttpResponse* Object 674
Properties of the *HttpResponse* Class 674
Methods of the *HttpResponse* Class 678
The *HttpRequest* Object 681
Properties of the *HttpRequest* Class 681
Methods of the *HttpRequest* Class 685
Conclusion 686
Just the Facts 687

15 ASP.NET State Management 689

The Application's State 690
Properties of the *HttpApplicationState* Class 691
Methods of the *HttpApplicationState* Class 692
State Synchronization 693
Tradeoffs of Application State 694
The Session's State 695
The Session-State HTTP Module 696
Properties of the *HttpSessionState* Class 700
Methods of the *HttpSessionState* Class 702
Working with a Session's State 702
Identifying a Session 703
Lifetime of a Session 709
Persist Session Data to Remote Servers 711
Persist Session Data to SQL Server 715
Customizing Session State Management 721
Building a Custom Session State Provider 722
Generating a Custom Session ID 725
The View State of a Page 728
The *StateBag* Class 728
Common Issues with View State 730
Programming Web Forms Without View State 733
Changes in the ASP.NET View State 736
Keeping the View State on the Server 741
Conclusion 745
Just the Facts 746

16 ASP.NET Caching 747

Caching Application Data 747
The *Cache* Class 748
Working with the ASP.NET *Cache* 752
Practical Issues 760
Designing a Custom Dependency 766
A Cache Dependency for XML Data 768
SQL Server Cache Dependency 773
Caching ASP.NET Pages 782
The *@OutputCache* Directive 782
The *HttpCachePolicy* Class 788
Caching Multiple Versions of a Page 791

Caching Portions of ASP.NET Pages . . . 794
Advanced Caching Features . . . 800
Conclusion . . . 803
Just the Facts . . . 804

17 ASP.NET Security . . . 805
Where the Threats Come From . . . 806
The ASP.NET Security Context . . . 807
Who Really Runs My ASP.NET Application? . . . 807
Changing the Identity of the ASP.NET Process . . . 810
The Trust Level of ASP.NET Applications . . . 813
ASP.NET Authentication Methods . . . 817
Using Forms Authentication . . . 819
Forms Authentication Control Flow . . . 820
The *FormsAuthentication* Class . . . 825
Configuration of *Forms* Authentication . . . 827
Advanced Forms Authentication Features . . . 831
The Membership and Role Management API . . . 836
The *Membership* Class . . . 836
The Membership Provider . . . 842
Managing Roles . . . 847
Security-Related Controls . . . 853
The *Login* Control . . . 853
The *LoginName* Control . . . 856
The *LoginStatus* Control . . . 856
The *LoginView* Control . . . 858
The *PasswordRecovery* Control . . . 860
The *ChangePassword* Control . . . 862
The *CreateUserWizard* Control . . . 863
Conclusion . . . 865
Just the Facts . . . 866

18 HTTP Handlers and Modules . . . 867
Quick Overview of the IIS Extensibility API . . . 868
The ISAPI Model . . . 868
Changes in IIS 7.0 . . . 872
Writing HTTP Handlers . . . 873
The *IHttpHandler* Interface . . . 873
An HTTP handler for Quick Data Reports . . . 876

The Picture Viewer Handler 882
Serving Images More Effectively 886
Advanced HTTP Handler Programming 894
Writing HTTP Modules 901
The *IHttpModule* Interface 901
A Custom HTTP Module 903
The Page Refresh Feature 906
Conclusion 913
Just the Facts 913

Part IV **ASP.NET AJAX Extensions**

19 Partial Rendering: The Easy Way to AJAX 917
The ASP.NET AJAX Infrastructure 918
The Hidden Engine of AJAX 919
The Microsoft AJAX JavaScript Library 926
The Script Manager Control 939
Selective Page Updates with Partial Rendering 950
The *UpdatePanel* Control 951
Optimizing the Usage of the *UpdatePanel* Control 957
Giving Feedback to the User 962
Light and Shade of Partial Rendering 969
The AJAX Control Toolkit 973
Enhancing Controls with Extenders 973
Improving the User Interface with Input Extenders 981
Adding Safe Popup Capabilities to Web Pages 994
Conclusion 1002
Just the Facts 1003

20 AJAX-Enabled Web Services 1005
Implementing the AJAX Paradigm 1006
Moving Away from Partial Rendering 1006
Designing the –Client Layer of an ASP.NET AJAX Application 1008
Designing the –Server Layer of ASP.NET AJAX Applications 1010
Web Services for ASP.NET AJAX Applications 1013
Web Services as Application-Specific Services 1013
Remote Calls via Web Services 1016
Consuming AJAX Web Services 1020
Considerations for AJAX-Enabled Web Services 1026

WCF Services for ASP.NET AJAX Applications 1028
Building a Simple WCF Service 1028
Building a Less Simple Service 1033
Remote Calls via Page Methods 1036
Introducing Page Methods 1036
Consuming Page Methods 1038
Conclusion 1041
Just the Facts 1042

21 Silverlight and Rich Internet Applications 1043
Silverlight Fast Facts 1044
Versions of Silverlight 1044
Silverlight and Flash 1047
Hosting Silverlight in Web Pages 1048
The Silverlight Engine 1049
Defining XAML Content 1057
The XAML Syntax in Silverlight 1062
The Silverlight Object Model 1074
Silverlight Programming Fundamentals 1074
Introducing Silverlight 2.0 1082
Conclusion 1087

Index 1089

Chapter 18

HTTP Handlers and Modules

In this chapter:

Quick Overview of the IIS Extensability API 868
Writing HTTP Handlers 873
Writing HTTP Modules 901
Conclusion 913

HTTP modules and HTTP handlers are fundamental pieces of the ASP.NET architecture. HTTP handlers and modules are truly the building blocks of the .NET Web platform. Any requests for an ASP.NET managed resource is always resolved by an HTTP handler and passes through a pipeline of HTTP modules. After the handler has processed the request, the request flows back through the pipeline of HTTP modules and is finally transformed into markup for the caller.

An HTTP handler is the component that actually takes care of serving the request. It is an instance of a class that implements the *IHttpHandler* interface. The *ProcessRequest* method of the interface is the central console that governs the processing of the request. For example, the *Page* class—the base class for all ASP.NET run-time pages—implements the *IHttpHandler* interface, and its *ProcessRequest* method is responsible for loading and saving the view state and for firing the well-known set of page events, including *Init*, *Load*, *PreRender*, and the like.

ASP.NET maps each incoming HTTP request to a particular HTTP handler. A special breed of component—named the *HTTP handler factory*—provides the infrastructure for creating the physical instance of the handler to service the request. For example, the *PageHandlerFactory* class parses the source code of the requested *.aspx* resource and returns a compiled instance of the class that represents the page. An HTTP handler is designed to process one or more URL extensions. Handlers can be given an application or machine scope, which means they can process the assigned extensions within the context of the current application or all applications installed on the machine. Of course, this is accomplished by making changes to either the machine-wide *web.config* file or a local *web.config* file, depending on the scope you desire.

HTTP modules are classes that implement the *IHttpModule* interface and handle runtime events. There are two types of public events that a module can deal with. They are the events raised by *HttpApplication* (including asynchronous events) and events raised by other HTTP modules. For example, *SessionStateModule* is one of the built-in modules provided by

ASP.NET to supply session-state services to an application. It fires the *End* and *Start* events that other modules can handle through the familiar *Session_End* and *Session_Start* signatures.

HTTP handlers and HTTP modules have the same functionality as ISAPI extensions and ISAPI filters, respectively, but with a much simpler programming model. ASP.NET allows you to create custom handlers and custom modules. Before we get into this rather advanced aspect of Web programming, a review of the Internet Information Services (IIS) extensibility model is in order because this model determines what modules and handlers can do and what they cannot do.

Note ISAPI stands for Internet Server Application Programming Interface and represents the protocol by means of which IIS talks to external components. The ISAPI model is based on a Microsoft Win32 unmanaged dynamic-link library (DLL) that exports a couple of functions. This model is significantly expanded in IIS 7.0 and largely matches the ASP.NET extensibility model, which is based on HTTP handlers and modules. I'll return to this topic shortly.

Quick Overview of the IIS Extensibility API

A Web server is primarily a server application that can be contacted using a bunch of Internet protocols, such as HTTP, File Transfer Protocol (FTP), Network News Transfer Protocol (NNTP), and the Simple Mail Transfer Protocol (SMTP). IIS—the Web server included with the Microsoft Windows operating system—is no exception.

A Web server generally also provides a documented application programming interface (API) for enhancing and customizing the server's capabilities. Historically speaking, the first of these extension APIs was the Common Gateway Interface (CGI). A CGI module is a new application that is spawned from the Web server to service a request. Nowadays, CGI applications are almost never used in modern Web applications because they require a new process for each HTTP request. As you can easily understand, this approach is rather inadequate for high-volume Web sites and poses severe scalability issues. IIS supports CGI applications, but you will seldom use this feature unless you have serious backward-compatibility issues. More recent versions of Web servers supply an alternate and more efficient model to extend the capabilities of the module. In IIS, this alternative model takes the form of the ISAPI interface.

The ISAPI Model

When the ISAPI model is used, instead of starting a new process for each request, IIS loads an ISAPI component—namely, a Win32 DLL—into its own process. Next, it calls a well-known entry point on the DLL to serve the request. The ISAPI component stays loaded until IIS is shut down and can service requests without any further impact on Web server activity. The

downside to such a model is that because components are loaded within the Web server process, a single faulty component can tear down the whole server and all installed applications. Starting with IIS 4.0, though, some countermeasures have been taken to address this problem. Before the advent of IIS 6.0, you were allowed to set the protection level of a newly installed application choosing from three options: low, medium, and high.

If you choose a low protection, the application (and its extensions) will be run within the Web server process (*inetinfo.exe*). If you choose medium protection, applications will be pooled together and hosted by an instance of a different worker process (*dllhost.exe*). If you choose high protection, each application set to High will be hosted in its own individual worker process (*dllhost.exe*).

Web applications running under IIS 6.0 are grouped in pools, and the choice you can make is whether you want to join an existing pool or create a new one. Figure 18-1 shows the dialog box picking the application pool of choice in IIS 6.0 and Microsoft Windows Server 2003.

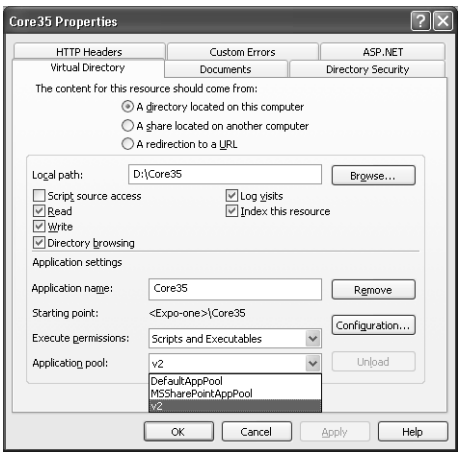

FIGURE 18-1 Configuring the protection level of Web applications in IIS 6.0 under Windows Server 2003.

All applications in a pool share the same run-time settings and the same worker process—*w3wp.exe*.

Illustrious Children of the ISAPI Model

The ISAPI model has another key drawback—the programming model. An ISAPI component represents a compiled piece of code—a Win32 DLL—that retrieves data and writes HTML code to an output console. It has to be developed using C or C++, it should generate multithreaded code, and it must be written with extreme care because of the impact that bugs or runtime failures can have on the application.

A while back, Microsoft attempted to encapsulate the ISAPI logic in the Microsoft Foundation Classes (MFC), but even though the effort was creditable, it didn't pay off very well. MFC tended to bring more code to the table than high-performance Web sites would perhaps like, and worse, the resulting ISAPI extension DLL suffered from a well-documented memory leak.

Active Server Pages (ASP), the predecessor of ASP.NET, is, on the other hand, an example of a well-done ISAPI application. ASP is implemented as an ISAPI DLL (named *asp.dll*) registered to handle HTTP requests with an *.asp* extension. The internal code of the ASP ISAPI extension DLL parses the code of the requested resource, executes any embedded script code, and builds the page for the browser.

As of IIS 6.0, any functionality built on top of IIS must be coded according to the guidelines set by the ISAPI model. ASP and ASP.NET are no exceptions. Today, the whole ASP.NET platform works closely with IIS, but it is not part of it. The *aspnet_isapi.dll* core component is the link between IIS and the ASP.NET runtime environment. When a request for *.aspx* resources comes in, IIS passes the control to *aspnet_isapi.dll*, which in turn hands the request to the ASP.NET pipeline inside an instance of the common language runtime (CLR).

As of this writing, to extend IIS you can write a Win32 DLL only with a well-known set of entry points. This requirement ceases to exist with IIS 7.0, which is scheduled to ship with Windows 2008 Server.

Note A good place to learn about IIS 7.0 and find good scripts and code snippets is *http://www.iis.net*. IIS 7.0 is also part of Windows Vista, but that is not particularly relevant here in the context of an ASP.NET book. Although you can certainly develop part of your Web site on a Windows Vista machine, it is simply out of question that you use Windows Vista as a Web server to host a site. Although fully functional, the IIS 7.0 that has shipped with Windows Vista can be seen as a live tool to experiment and test. The "real" IIS 7.0 for Web developers and administrators will ship in 2008 with Windows 2008 Server.

Structure of ISAPI Components

An ISAPI extension is invoked through a URL that ends with the name of the DLL that implements the function, as shown in the following URL:

```
http://www.contoso.com/apps/hello.dll
```

The DLL must export a couple of functions—*GetExtensionVersion* and *HttpExtensionProc*. The *GetExtensionVersion* function sets the version and the name of the ISAPI server extension. When the extension is loaded, the *GetExtensionVersion* function is the first function to be called. *GetExtensionVersion* is invoked only once and can be used to initialize any needed variables. The function is expected to return *true* if everything goes fine. In the case of errors, the function should return *false* and the Web server will abort loading the DLL and put a message in the system log.

The core of the ISAPI component is represented by the *HttpExtensionProc* function. The function receives basic HTTP information regarding the request (for example, the query string and the headers), performs the expected action, and prepares the response to send back to the browser.

Note Certain handy programming facilities, such as the session state, are abstractions the ISAPI programming model lacks entirely. The ISAPI model is a lower level programming model than, say, ASP or ASP.NET.

The ISAPI programming model is made of two types of components—ISAPI extensions and ISAPI filters.

ISAPI Extensions

ISAPI extensions are the IIS in-process counterpart of CGI applications. As mentioned, an ISAPI extension is a DLL that is loaded in the memory space occupied by IIS or another host application. Because it is a DLL, only one instance of the ISAPI extension needs to be loaded at a time. On the downside, the ISAPI extension must be thread-safe so that multiple client requests can be served simultaneously. ISAPI extensions work in much the same way as an ASP or ASP.NET page. It takes any information about the HTTP request and prepares a valid HTTP response.

Because the ISAPI extension is made of compiled code, it must be recompiled and reloaded at any change. If the DLL is loaded in the Web server's memory, the Web server must be stopped. If the DLL is loaded in the context of a separate process, only that process must be stopped. Of course, when an external process is used, the extension doesn't work as fast as it could when hosted in-process, but at least it doesn't jeopardize the stability of IIS.

ISAPI Filters

ISAPI filters are components that intercept specific server events before the server itself handles them. Upon loading, the filter indicates what event notifications it will handle. If any of these events occur, the filter can process them or pass them on to other filters.

You can use ISAPI filters to provide custom authentication techniques or to automatically redirect requests based on HTTP headers sent by the client. Filters are a delicate gear in the IIS machinery. They can facilitate applications and let them take control of customizable aspects of the engine. For this same reason, though, ISAPI filters can also degrade performance if not written carefully. Filters, in fact, can run only in-process. Filters can be loaded for the Web server as a whole or for specific Web sites.

ISAPI filters can accomplish tasks such as implementing custom authentication schemes, compression, encryption, logging, and request analysis. The ability to examine, and if necessary modify, both incoming and outgoing streams of data makes ISAPI filters very

powerful and flexible. This last sentence shows the strength of ISAPI filters but also indicates their potential weakness, which is that they will hinder performance if not written well.

Changes in IIS 7.0

ASP.NET 1.0 was originally a self-contained, brand new runtime environment bolted onto IIS 5.0. With the simultaneous release of ASP.NET 1.1 and IIS 6.0, the Web development and server platforms have gotten closer and started sharing some services, such as process recycling and output caching. The advent of ASP.NET 2.0 and newer versions hasn't changed anything, but the release of IIS 7.0 will.

A Unified Runtime Environment

In a certain way, IIS 7.0 represents the unification of the ASP.NET and IIS platforms. HTTP handlers and modules, the runtime pipeline, and configuration files will become constituent elements of a common environment. The whole IIS internal pipeline has been componentized to originate a distinct and individually configurable component. A new section will be added to the *web.config* schema of ASP.NET applications to configure the IIS environment.

Put another way, it will be as if the ASP.NET runtime expanded to incorporate and replace the surrounding Web server environment. It's hard to say whether things really went this way or whether it was the other way around. As a matter of fact, the same concepts and instruments you know from ASP.NET are available in IIS 7.0 at the Web server level.

To illustrate, on Windows 2008 Server (and for testing purposes, also on a Windows Vista machine) you can use Forms authentication to protect access to any resources available on the server and not just ASP.NET-specific resources. You might already know that static resources such as HTML pages and JPG images are not served by ASP.NET by default; as such, they're not subject to the authentication rules you set for the application. Where IIS 7.0 is supported, you can now define a handler for some specific and static resources and be sure that IIS will use your code to serve those resources.

Managed ISAPI Extensions and Filters

Today if you want to take control of an incoming request in any version of IIS older than version 7.0, you have no choice other than writing a C or C++ DLL, using either MFC or perhaps the ActiveX Template Library (ATL). More comfortable HTTP handlers and modules are an ASP.NET-only feature, and they can be applied only to ASP.NET-specific resources and only after the request has been authenticated by IIS and handed over to ASP.NET.

In IIS 7.0, you can write HTTP handlers and modules to filter *any* requests and implement any additional features using .NET code for whatever resources the Web server can serve. More precisely, you'll continue writing HTTP handlers and modules as you do today for ASP.NET,

except that you will be given the opportunity to register them for any file type. Needless to say, old-style ISAPI extensions will still be supported, but unmanaged extensions and filters will likely become a thing of the past. I'll demonstrate IIS 7.0 handlers later in the chapter.

Writing HTTP Handlers

ASP.NET comes with a small set of built-in HTTP handlers. There is a handler to serve ASP.NET pages, one for Web services, and yet another to accommodate .NET Remoting requests for remote objects hosted by IIS. Other helper handlers are defined to view the tracing of individual pages in a Web application (*trace.axd*) and to block requests for prohibited resources such as *.config* or *.asax* files. Starting with ASP.NET 2.0, you also find a handler (*webresource.axd*) to inject assembly resources and script code into pages. In ASP.NET 3.5, the *scriptresource.axd* handler has been added as a more refined tool to inject script code and AJAX capabilities into Web pages.

You can write custom HTTP handlers whenever you need ASP.NET to process certain requests in a nonstandard way. The list of useful things you can do with HTTP handlers is limited only by your imagination. Through a well-written handler, you can have your users invoke any sort of functionality via the Web. For example, you could implement click counters and any sort of image manipulation, including dynamic generation of images, server-side caching, or obstructing undesired linking to your images.

Note An HTTP handler can either work synchronously or operate in an asynchronous way. When working synchronously, a handler doesn't return until it's done with the HTTP request. An asynchronous handler, on the other hand, launches a potentially lengthy process and returns immediately after. A typical implementation of asynchronous handlers are asynchronous pages. Later in this chapter, though, we'll take a look at the mechanics of asynchronous handlers, of which asynchronous pages are a special case.

Conventional ISAPI extensions and filters should be registered within the IIS metabase. In contrast, HTTP handlers are registered in the *web.config* file if you want the handler to participate in the HTTP pipeline processing of the Web request. In a manner similar to ISAPI extensions, you can also invoke the handler directly via the URL.

The *IHttpHandler* Interface

Want to take the splash and dive into HTTP handler programming? Well, your first step is getting the hang of the *IHttpHandler* interface. An HTTP handler is just a managed class that implements that interface. More specifically, a synchronous HTTP handler implements the *IHttpHandler* interface; an asynchronous HTTP handler, on the other hand, implements the *IHttpAsyncHandler* interface. Let's tackle synchronous handlers first.

The contract of the *IHttpHandler* interface defines the actions that a handler needs to take to process an HTTP request synchronously.

Members of the *IHttpHandler* Interface

The *IHttpHandler* interface defines only two members—*ProcessRequest* and *IsReusable*, as shown in Table 18-1. *ProcessRequest* is a method, whereas *IsReusable* is a Boolean property.

TABLE 18-1 Members of the *IHttpHandler* Interface

Member	Description
IsReusable	This property gets a Boolean value indicating whether the HTTP runtime can reuse the current instance of the HTTP handler while serving another request.
ProcessRequest	This method processes the HTTP request.

The *IsReusable* property on the *System.Web.UI.Page* class—the most common HTTP handler in ASP.NET—returns *false*, meaning that a new instance of the HTTP request is needed to serve each new page request. You typically make *IsReusable* return *false* in all situations where some significant processing is required that depends on the request payload. Handlers used as simple barriers to filter special requests can set *IsReusable* to *true* to save some CPU cycles. I'll return to this subject with a concrete example in a moment.

The *ProcessRequest* method has the following signature:

```
void ProcessRequest(HttpContext context);
```

It takes the context of the request as the input and ensures that the request is serviced. In the case of synchronous handlers, when *ProcessRequest* returns, the output is ready for forwarding to the client.

A Very Simple HTTP Handler

Again, an HTTP handler is simply a class that implements the *IHttpHandler* interface. The output for the request is built within the *ProcessRequest* method, as shown in the following code:

```
using System.Web;

namespace Core35.Components
{
    public class SimpleHandler : IHttpHandler
    {
        // Override the ProcessRequest method
        public void ProcessRequest(HttpContext context)
        {
            context.Response.Write("<H1>Hello, I'm an HTTP handler</H1>");
        }
```

```
        // Override the IsReusable property
        public bool IsReusable
        {
            get { return true; }
        }
    }
}
```

You need an entry point to be able to call the handler. In this context, an entry point into the handler's code is nothing more than an HTTP endpoint—that is, a public URL. The URL must be a unique name that IIS and the ASP.NET runtime can map to this code. When registered, the mapping between an HTTP handler and a Web server resource is established through the *web.config* file:

```
<configuration>
    <system.web>
        <httpHandlers>
            <add verb="*" path="hello.aspx"
                type="Core35.Components.SimpleHandler" />
        </httpHandlers>
    </system.web>
</configuration>
```

The <*httpHandlers*> section lists the handlers available for the current application. These settings indicate that *SimpleHandler* is in charge of handling any incoming requests for an endpoint named *hello.aspx*. Note that the URL *hello.aspx* doesn't have to be a physical resource on the server; it's simply a public resource identifier. The *type* attribute references the class and assembly that contains the handler. It's canonical format is *type[,assembly]*. You omit the assembly information if the component is defined in the *App_Code* or other reserved folders.

Note If you enter the settings shown earlier in the global *web.config* file, you will register the *SimpleHandler* component as callable from within all Web applications hosted by the server machine.

If you invoke the *hello.aspx* URL, you obtain the results shown in Figure 18-2.

FIGURE 18-2 A sample HTTP handler that answers requests for *hello.aspx*.

The technique discussed here is the quickest and simplest way of putting an HTTP handler to work, but there is more to know about registration of HTTP handlers and there are many more options to take advantage of. Now let's consider a more complex example of an HTTP handler.

An HTTP Handler for Quick Data Reports

With their relatively simple programming model, HTTP handlers give you a means of interacting with the low-level request and response services of IIS. In the previous example, we returned only constant text and made no use of the request information. In the next example, we'll configure the handler to intercept and process only requests of a particular type and generate the output based on the contents of the requested resource.

The idea is to build an HTTP handler for custom *.sqlx* resources. A SQLX file is an XML document that expresses the statements for one or more SQL queries. The handler grabs the information about the query, executes it, and finally returns the result set formatted as a grid. Figure 18-3 shows the expected outcome.

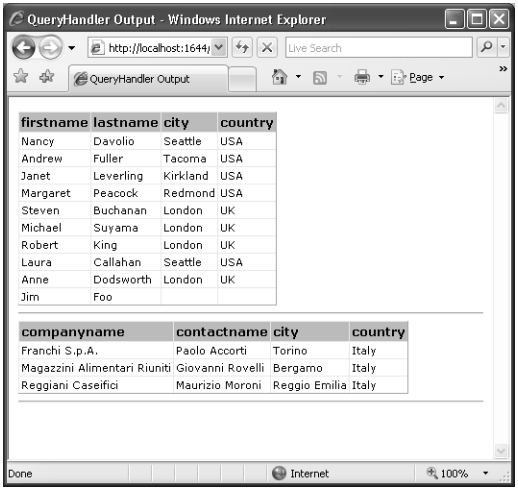

FIGURE 18-3 A custom HTTP handler in action.

To start, let's examine the source code for the *IHttpHandler* class.

Warning Take this example for what it really is—merely a way to process a custom XML file with a custom extension doing something more significant than outputting a "hello world" message. *Do not* take this handler as a realistic prototype for exposing your Microsoft SQL Server databases over the Web.

Building a Query Manager Tool

The HTTP handler should get into the game whenever the user requests an *.sqlx* resource. Assume for now that the system knows how to deal with such a weird extension, and focus on what's needed to execute the query and pack the results into a grid. To execute the query, at a minimum, we need the connection string and the command text. The following text illustrates the typical contents of an *.sqlx* file:

```
<queries>
  <query connString="DATABASE=northwind;SERVER=localhost;UID...;">
    SELECT firstname, lastname, country FROM employees
  </query>
  <query connString="DATABASE=northwind;SERVER=localhost;UID=...;">
    SELECT companyname FROM customers WHERE country='Italy'
  </query>
</queries>
```

The XML document is formed by a collection of <*query*> nodes, each containing an attribute for the connection string and the text of the query.

The *ProcessRequest* method extracts this information before it can proceed with executing the query and generating the output:

```
class SqlxData
{
    public string ConnectionString;
    public string QueryText;
}

public class QueryHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        // Parses the SQLX file
        SqlxData[] data = ParseFile(context);

        // Create the output as HTML
        StringCollection htmlColl = CreateOutput(data);

        // Output the data
        context.Response.Write("<html><head><title>");
        context.Response.Write("QueryHandler Output");
        context.Response.Write("</title></head><body>");
        foreach (string html in htmlColl)
        {
            context.Response.Write(html);
            context.Response.Write("<hr />");
        }
        context.Response.Write("</body></html>");
    }
```

```
    // Override the IsReusable property
    public bool IsReusable
    {
        get { return true; }
    }

    ...
}
```

The *ParseFile* helper function parses the source code of the *.sqlx* file and creates an instance of the *SqlxData* class for each query found:

```
private SqlxData[] ParseFile(HttpContext context)
{
   XmlDocument doc = new XmlDocument();
   string filePath = context.Request.Path;
   using (Stream fileStream = VirtualPathProvider.OpenFile(filePath))  {
       doc.Load(fileStream);
   }

   // Visit the <mapping> nodes
   XmlNodeList mappings = doc.SelectNodes("queries/query");
   SqlxData[] descriptors = new SqlxData[mappings.Count];
   for (int i=0; i < descriptors.Length; i++)
   {
       XmlNode mapping = mappings[i];
       SqlxData query = new SqlxData();
       descriptors[i] = query;

       try {
           query.ConnectionString =
              mapping.Attributes["connString"].Value;
           query.QueryText = mapping.InnerText;
       }
       catch {
           context.Response.Write("Error parsing the input file.");
           descriptors = new SqlxData[0];
           break;
       }
   }
   return descriptors;
}
```

The *SqlxData* internal class groups the connection string and the command text. The information is passed to the *CreateOutput* function, which will actually execute the query and generate the grid:

```
private StringCollection CreateOutput(SqlxData[] descriptors)
{
    StringCollection coll = new StringCollection();

    foreach (SqlxData data in descriptors)
    {
```

```
        // Run the query
        DataTable dt = new DataTable();
        SqlDataAdapter adapter = new SqlDataAdapter(data.QueryText,
            data.ConnectionString);
        adapter.Fill(dt);


        // Error handling
        ...

        // Prepare the grid
        DataGrid grid = new DataGrid();
        grid.DataSource = dt;
        grid.DataBind();

        // Get the HTML
        string html = Utils.RenderControlAsString(grid);
        coll.Add(html);
    }
    return coll;
}
```

After executing the query, the method populates a dynamically created *DataGrid* control. In ASP.NET pages, the *DataGrid* control, like any other control, is rendered to HTML. However, this happens through the care of the special HTTP handler that manages *.aspx* resources. For *.sqlx* resources, we need to provide that functionality ourselves. Obtaining the HTML for a Web control is as easy as calling the *RenderControl* method on an HTML text writer object. This is just what the helper method *RenderControlAsString* does:

```
static class Utils
{
    public static string RenderControlAsString(Control ctl)
    {
        StringWriter sw = new StringWriter();
        HtmlTextWriter writer = new HtmlTextWriter(sw);
        ctl.RenderControl(writer);
        return sw.ToString();
    }
}
```

Note An HTTP handler that needs to access session-state values must implement the *IRequiresSessionState* interface. Like *INamingContainer*, it's a marker interface and requires no method implementation. Note that the *IRequiresSessionState* interface indicates that the HTTP handler requires read and write access to the session state. If read-only access is needed, use the *IReadOnlySessionState* interface instead.

Registering the Handler

An HTTP handler is a class and must be compiled to an assembly before you can use it. The assembly must be deployed to the *Bin* directory of the application. If you plan to make this handler available to all applications, you can copy it to the global assembly cache (GAC). The next step is registering the handler with an individual application or with all the applications running on the Web server. You register the handler in the configuration file:

```
<system.web>
  <httpHandlers>
    <add verb="*"
      path="*.sqlx"
      type= "Core35.Components.QueryHandler,Core35Lib" />
  </httpHandlers>
</system.web>
```

You add the new handler to the <*httpHandlers*> section of the local or global *web.config* file. The section supports three actions: <*add*>, <*remove*>, and <*clear*>. You use <*add*> to add a new HTTP handler to the scope of the *.config* file. You use <*remove*> to remove a particular handler. Finally, you use <*clear*> to get rid of all the registered handlers. To add a new handler, you need to set three attributes—*verb*, *path*, and *type*—as shown in Table 18-2.

TABLE 18-2 Attributes Needed to Register an HTTP Handler

Attribute	Description
Verb	Indicates the list of the supported HTTP verbs—for example, *GET*, *PUT*, and *POST*. The wildcard character (*) is an acceptable value and denotes all verbs.
Path	A wildcard string, or a single URL, that indicates the resources the handler will work on—for example, **.aspx*.
Type	Specifies a comma-separated class/assembly combination. ASP.NET searches for the assembly DLL first in the application's private *Bin* directory and then in the system global assembly cache.

These attributes are mandatory. An optional attribute is also supported—*validate*. When *validate* is set to *false*, ASP.NET delays as much as possible loading the assembly with the HTTP handler. In other words, the assembly will be loaded only when a request for it arrives. ASP.NET will not try to preload the assembly, thus catching any error or problem with it.

So far, you have correctly deployed and registered the HTTP handler, but if you try invoking an *.sqlx* resource, the results you produce are not what you'd expect. The problem lies in the fact that so far you configured ASP.NET to handle only *.sqlx* resources, but IIS still doesn't know anything about them!

A request for an *.sqlx* resource is handled by IIS *before* it is handed to the ASP.NET ISAPI extension. If you don't register *some* ISAPI extension to handle ..sqlx resource requests, IIS will treat each request as a request for a static resource and serve the request by sending

back the source code of the .sqlx file. The extra step required is registering the *.sqlx* extension with the IIS 6.0 metabase such that requests for .sqlx resources are handed off to ASP.NET, as shown in Figure 18-4.

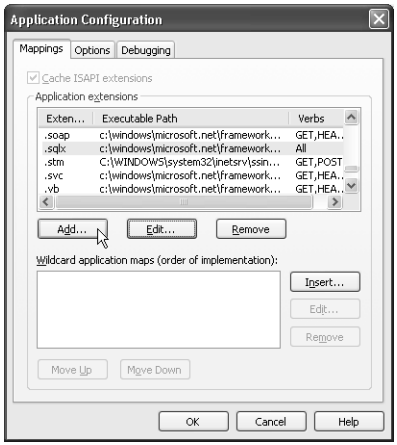

FIGURE 18-4 Registering the *.sqlx* extension with the IIS 6.0 metabase.

The dialog box in the figure is obtained by clicking on the properties of the application in the IIS 6.0 manager and then the configuration of the site. To involve the HTTP handler, you must choose *aspnet_isapi.dll* as the ISAPI extension. In this way, all *.sqlx* requests are handed out to ASP.NET and processed through the specified handler. Make sure you select *aspnet_isapi.dll* from the folder of the ASP.NET version you plan to use.

Caution In Microsoft Visual Studio, if you test a sample *.sqlx* resource using the local embedded Web server, nothing happens that forces you to register the *.sqlx* resource with IIS. This is just the point, though. You're not using IIS! In other words, if you use the local Web server, you have no need to touch IIS; you do need to register any custom resource you plan to use with IIS before you get to production.

Registering the Handler with IIS 7.0

If you run IIS 7.0, you don't strictly need to change anything through the IIS Manager. You can add a new section to the *web.config* file and specify the HTTP handler also for static resources that would otherwise be served directly by IIS. Here's what you need to enter:

```
<system.webServer>
    <add verb="*"
        path="*.sqlx"
        type="Core35.Components.QueryHandler, Core35Lib" />
</system.webServer>
```

The new section is a direct child of the root tag <*configuration*>. Without this setting, IIS can't recognize the page and won't serve it up. The configuration script instructs IIS 7.0 to forward any **.sqlx* requests to your application, which knows how to deal with it.

The Picture Viewer Handler

Let's examine another scenario that involves custom HTTP handlers. Thus far, we have explored custom resources and realized how important it is to register any custom extensions with IIS.

To speed up processing, IIS claims the right of personally serving some resources that typically form a Web application without going down to a particular ISAPI extension. The list includes static files such as images and HTML files. What if you request a GIF or a JPG file directly from the address bar of the browser? IIS retrieves the specified resource, sets the proper content type on the response buffer, and writes out the bytes of the file. As a result, you'll see the image in the browser's page. So far so good.

What if you point your browser to a virtual folder that contains images? In this case, IIS doesn't distinguish the contents of the folder and returns a list of files, as shown in Figure 18-5.

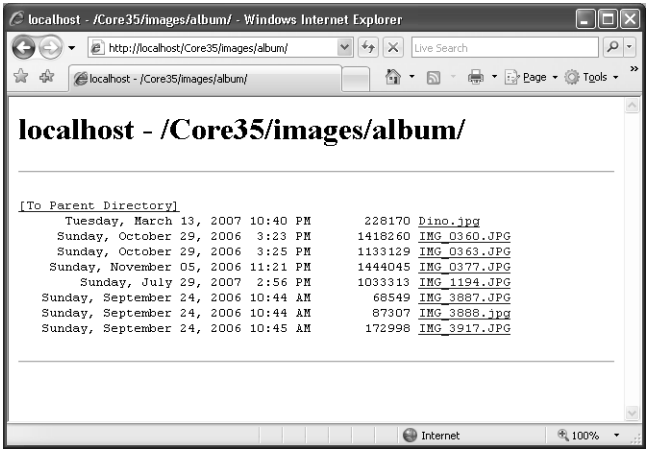

FIGURE 18-5 The standard IIS-provided view of a folder.

Wouldn't it be nice if you could get a preview of the contained pictures, instead?

Designing the HTTP Handler

To start out, you need to decide how you would let IIS know about your wishes. You can use a particular endpoint that, appended to a folder's name, convinces IIS to yield to ASP.NET and provide a preview of contained images. Put another way, the idea is binding our picture

viewer handler to a particular endpoint—say, *folder.axd*. As mentioned earlier in the chapter, a fixed endpoint for handlers doesn't have to be an existing, deployed resource. You make the *folder.axd* endpoint follow the folder name, as shown here:

```
http://www.contoso.com/images/folder.axd
```

The handler will process the URL, extract the folder name, and select all the contained pictures.

Note In ASP.NET, the *.axd* extension is commonly used for endpoints referencing a special service. *Trace.axd* for tracing and *WebResource.axd* for script and resources injection are examples of two popular uses of the extension. In particular, the *Trace.axd* handler implements the same logic described here. If you append its name to the URL, it will trace all requests for pages in that application.

Implementing the HTTP Handler

The picture viewer handler returns a page composed of a multirow table showing as many images as there are in the folder. Here's the skeleton of the class:

```
class PictureViewerInfo
{
    public PictureViewerInfo() {
        DisplayWidth = 200;
        ColumnCount = 3;
    }
    public int DisplayWidth;
    public int ColumnCount;
    public string FolderName;
}

public class PictureViewerHandler : IHttpHandler
{
    // Override the ProcessRequest method
    public void ProcessRequest(HttpContext context)
    {
        PictureViewerInfo info = GetFolderInfo(context);
        string html = CreateOutput(info);

        // Output the data
        context.Response.Write("<html><head><title>");
        context.Response.Write("Picture Web Viewer");
        context.Response.Write("</title></head><body>");
        context.Response.Write(html);
        context.Response.Write("</body></html>");
    }
```

```
    // Override the IsReusable property
    public bool IsReusable
    {
        get { return true; }
    }
    ...
}
```

Retrieving the actual path of the folder is as easy as stripping off the *folder.axd* string from the URL and trimming any trailing slashes or backslashes. Next, the URL of the folder is mapped to a server path and processed using the .NET Framework API for files and folders:

```
private ArrayList GetAllImages(string path)
{
    string[] fileTypes = { "*.bmp", "*.gif", "*.jpg", "*.png" };
    ArrayList images = new ArrayList();
    DirectoryInfo di = new DirectoryInfo(path);
    foreach (string ext in fileTypes)
    {
        FileInfo[] files = di.GetFiles(ext);
        if (files.Length > 0)
            images.AddRange(files);
    }
    return images;
   }
```

The *DirectoryInfo* class provides some helper functions on the specified directory; for example, the *GetFiles* method selects all the files that match the given pattern. Each file is wrapped by a *FileInfo* object. The method *GetFiles* doesn't support multiple search patterns; to search for various file types, you need to iterate for each type and accumulate results in an array list or equivalent data structure.

After you get all the images in the folder, you move on to building the output for the request. The output is a table with a fixed number of cells and a variable number of rows to accommodate all selected images. The image is not downloaded as a thumbnail, but it is more simply rendered in a smaller area. For each image file, a new <*img*> tag is created through the *Image* control. The *width* attribute of this file is set to a fixed value (say, 200 pixels), causing most modern browsers to automatically resize the image. Furthermore, the image is wrapped by an anchor that links to the same image URL. As a result, when the user clicks on an image, the page refreshes and shows the same image at its natural size.

```
string CreateOutputForFolder(PictureViewerInfo info)
{
    ArrayList images = GetAllImages(info.FolderName);
    Table t = new Table();

    int index = 0;
    bool moreImages = true;
```

```
    while (moreImages)
    {
        TableRow row = new TableRow();
        t.Rows.Add(row);
        for (int i = 0; i < info.ColumnCount; i++)
        {
            TableCell cell = new TableCell();
            row.Cells.Add(cell);

            // Create the image
            Image img = new Image();
            FileInfo fi = (FileInfo)images[index];
            img.ImageUrl = fi.Name;
            img.Width = Unit.Pixel(info.DisplayWidth);

            // Wrap the image in an anchor so that a larger image
            // is shown when the user clicks
            HtmlAnchor a = new HtmlAnchor();
            a.HRef = fi.Name;
            a.Controls.Add(img);
            cell.Controls.Add(a);

            // Check whether there are more images to show
            index++;
            moreImages = (index < images.Count);
            if (!moreImages)
                break;
        }
    }
}
```

You might want to make the handler accept some optional query string parameters, such as width and column count. These values are packed in an instance of the helper class *PictureViewerInfo* along with the name of the folder to view. Here's the code to process the query string of the URL to extract parameters if any are present:

```
PictureViewerInfo info = new PictureViewerInfo();
object p1 = context.Request.Params["Width"];
object p2 = context.Request.Params["Cols"];
if (p1 != null)
    Int32.TryParse((string)p1, out info.DisplayWidth);
if (p2 != null)
    Int32.TryParse((string)p2, out info.ColumnCount);
```

Figure 18-6 shows the handler in action.

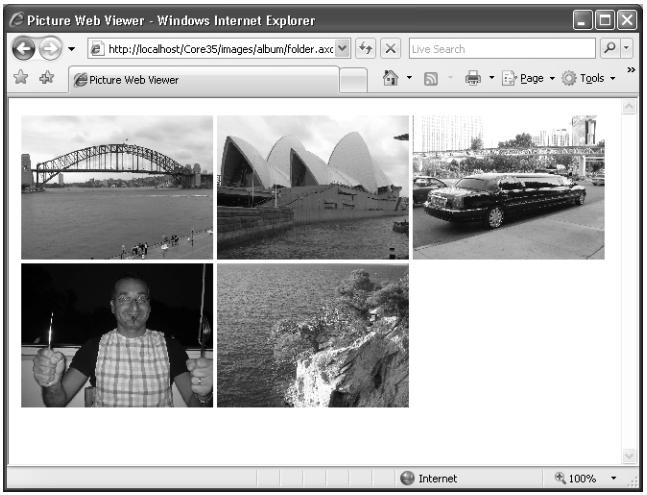

FIGURE 18-6 The picture viewer handler in action with a given number of columns and width.

Registering the handler is easy too. You just add the following script to the *web.config* file:

```
<add verb="*" path="folder.axd"
    type="Core35.Components.PictureViewerHandler,Core35Lib" />
```

You place the assembly in the GAC and move the configuration script to the global *web.config* to extend the settings to all applications on the machine.

Serving Images More Effectively

Any page we get from the Web today is topped with so many images and is so well conceived and designed that often the overall page looks more like a magazine advertisement than an HTML page. Looking at the current pages displayed by portals, it's rather hard to imagine there ever was a time—and it was only seven or eight years ago—when one could create a Web site by using only a text editor and some assistance from a friend who had a bit of familiarity with Adobe PhotoShop.

In spite of the wide use of images on the Web, there is just one way in which a Web page can reference an image—by using the HTML *<img>* tag. By design, this tag points to a URL. As a result, to be displayable within a Web page, an image must be identifiable through a URL and its bits should be contained in the output stream returned by the Web server for that URL.

In many cases, the URL points to a static resource such as a GIF or JPEG file. In this case, the Web server takes the request upon itself and serves it without invoking external components. However, the fact that many *<img>* tags on the Web are bound to a static file does not mean there's no other way to include images in Web pages.

Where else can you turn to get images aside from picking them up from the server file system? For example, you can load images from a database or you can generate or modify them on the fly just before serving the bits to the browser.

Loading Images from Databases

The use of a database as the storage medium for images is controversial. Some people have good reasons to push it as a solution; others tell you bluntly they would never do it and that you shouldn't either. Some people can tell you wonderful stories of how storing images in a properly equipped database was the best experience of their professional life. With no fear that facts could perhaps prove them wrong, other people will confess that they would never use a database again for such a task.

The facts say that all database management systems (DBMS) of a certain reputation and volume have supported binary large objects (BLOB) for quite some time. Sure, a BLOB field doesn't necessarily contain an image—it can contain a multimedia file or a long text file—but overall there must be a good reason for having this BLOB support in SQL Server, Oracle, and similar popular DBMS systems!

To read an image from a BLOB field with ADO.NET, you execute a *SELECT* statement on the column and use the *ExecuteScalar* method to catch the result and save it in an array of bytes. Next, you send this array down to the client through a binary write to the response stream. Let's write an HTTP handler to serve a database-stored image:

```
public class DbImageHandler : IHttpHandler
{
    public void ProcessRequest(HttpContext ctx)
    {
        // Ensure the URL contains an ID argument that is a number
        int id = -1;
        bool result = Int32.TryParse(ctx.Request.QueryString["id"], out id);
        if (!result)
           ctx.Response.End();

        string connString = "...";
        string cmdText = "SELECT photo FROM employees WHERE employeeid=@id";

        // Get an array of bytes from the BLOB field
        byte[] img = null;
        SqlConnection conn = new SqlConnection(connString);
        using (conn)
        {
            SqlCommand cmd = new SqlCommand(cmdText, conn);
            cmd.Parameters.AddWithValue("@id", id);
            conn.Open();
            img = (byte[])cmd.ExecuteScalar();
            conn.Close();
      }
```

```
        // Prepare the response for the browser
        if (img != null)
        {
            ctx.Response.ContentType = "image/jpeg";
            ctx.Response.BinaryWrite(img);
        }
    }

    public bool IsReusable
    {
        get { return true; }
    }
}
```

There are quite a few assumptions made in this code. First, we assume that the field named *photo* contains image bits and that the format of the image is JPEG. Second, we assume that images are to be retrieved from a fixed table of a given database through a predefined connection string. Finally, we're assuming that the URL to invoke this handler includes a query string parameter named *id*.

Notice the attempt to convert the value of the *id* query parameter to an integer before proceeding. This simple check significantly reduces the surface attack for malicious users by verifying that what is going to be used as a numeric ID is really a numeric ID. Especially when you're inoculating user input into SQL query commands, filtering out extra characters and wrong data types is a fundamental measure for preventing attacks.

The *BinaryWrite* method of the *HttpResponse* object writes an array of bytes to the output stream.

Warning If the database you're using is Northwind (as in the preceding example), an extra step might be required to ensure that the images are correctly managed. For some reason, the SQL Server version of the Northwind database stores the images in the *photo* column of the Employees table as OLE objects. This is probably because of the conversion that occurred when the database was upgraded from the Microsoft Access version. As a matter fact, the array of bytes you receive contains a 78-byte prefix that has nothing to do with the image. Those bytes are just the header created when the image was added as an OLE object to the first version of Access. Although the preceding code works like a champ with regular BLOB fields, it must undergo the following modification to work with the *photo* field of the Northwind.Employees database:

```
Response.OutputStream.Write(img, 78, img.Length);
```

Instead of using the *BinaryWrite* call, which doesn't let you specify the starting position, use the code shown here.

A sample page to test BLOB field access is shown in Figure 18-7. The page lets users select an employee ID and post back. When the page renders, the ID is used to complete the URL for the ASP.NET *Image* control.

```
string url = String.Format("dbimage.axd?id={0}",
                           DropDownList1.SelectedValue);
Image1.ImageUrl = url;
```

FIGURE 18-7 Downloading images stored within the BLOB field of a database.

An HTTP handler must be registered in the *web.config* file and bound to a public endpoint. In this case, the endpoint is *dbimage.axd* and the script to enter in the configuration file is shown next:

```
<httpHandlers>
    <add verb="*" path="dbimage.axd"
         type="Core35.Components.DbImageHandler,Core35Lib"/>
</httpHandlers>
```

Note The preceding handler clearly has a weak point: it hard-codes a SQL command and the related connection string. This means that you might need a different handler for each different command or database to access. A more realistic handler would probably use an external and configurable database-specific provider. Such a provider can be as simple as a class that implements an agreed interface. At a minimum, the interface will supply a method to retrieve and return an array of bytes. Alternatively, if you want to keep the ADO.NET code in the handler itself, the interface will just supply members that specify the command text and connection string. The handler will figure out its default provider from a given entry in the *web.config* file.

Serving Dynamically Generated Images

Isn't it true that an image is worth thousands of words? Many financial Web sites offer charts and, more often than not, these charts are dynamically generated on the server. Next, they are served to the browser as a stream of bytes and travel over the classic response out-

put stream. But can you create and manipulate server-side images? For these tasks, Web applications normally rely on ad hoc libraries or the graphic engine of other applications (for example, Microsoft Office applications).

ASP.NET applications are different and, to some extent, luckier. ASP.NET applications, in fact, can rely on a powerful and integrated graphic engine capable of providing an object model for image generation. This server-side system is GDI+, and contrary to what some people might have you believe, GDI+ is fair game for generating images on the fly for ASP.NET applications.

As its name suggests, GDI+ is the successor of GDI, the Graphics Device Interface included with versions of the Windows operating system that shipped before Windows XP. The .NET Framework encapsulates the key GDI+ functionalities in a handful of managed classes and makes those functions available to Web, Windows Forms, and Web service applications.

Most of the GDI+ services belong to the following categories: 2D vector graphics and imaging. 2D vector graphics involve drawing simple figures such as lines, curves, and polygons. Under the umbrella of imaging are functions to display, manipulate, save, and convert bitmap and vector images. Finally, a third category of functions can be identified—typography, which includes the display of text in a variety of fonts, sizes, and styles. Having the goal of creating images dynamically, we are most interested in drawing figures and text and in saving the work as JPEGs or GIFs.

In ASP.NET, writing images to disk might require some security adjustments. Normally, the ASP.NET runtime runs under the aegis of the *NETWORK SERVICE* user account. In the case of anonymous access with impersonation disabled—which are the default settings in ASP.NET—the worker process lends its own identity and security token to the thread that executes the user request of creating the file. With regard to the default scenario, an access denied exception might be thrown if *NETWORK SERVICE* lacks writing permissions on virtual directories—a pretty common situation.

ASP.NET and GDI+ provide an interesting alternative to writing files on disk without changing security settings: in-memory generation of images. In other words, the dynamically generated image is saved directly to the output stream in the needed image format or in a memory stream.

Writing Copyright Notes on Images

GDI+ supports quite a few image formats, including JPEG, GIF, BMP, and PNG. The whole collection of image formats is in the *ImageFormat* structure from the *System.Drawing* namespace. You can save a memory-resident *Bitmap* object to any of the supported formats by using one of the overloads of the *Save* method:

```
Bitmap bmp = new Bitmap(file);
...
bmp.Save(outputStream, ImageFormat.Gif);
```

When you attempt to save an image to a stream or disk file, the system attempts to locate an encoder for the requested format. The encoder is a GDI+ module that converts from the native format to the specified format. Note that the encoder is a piece of unmanaged code that lives in the underlying Win32 platform. For each save format, the *Save* method looks up the right encoder and proceeds.

The next example wraps up all the points we touched on. This example shows how to load an existing image, add some copyright notes, and serve the modified version to the user. In doing so, we'll load an image into a *Bitmap* object, obtain a *Graphics* for that bitmap, and use graphics primitives to write. When finished, we'll save the result to the page's output stream and indicate a particular MIME type.

The sample page that triggers the example is easily created, as shown in the following listing:

```
<html>
<body>
    <img id="picture" src="dynimage.axd?url=images/pic1.jpg" />
</body>
</html>
```

The page contains no ASP.NET code and displays an image through a static HTML <*img*> tag. The source of the image, though, is an HTTP handler that loads the image passed through the query string, and then manipulates and displays it. Here's the source code for the *ProcessRequest* method of the HTTP handler:

```
public void ProcessRequest (HttpContext context)
{
    object o = context.Request["url"];
    if (o == null)
    {
        context.Response.Write("No image found.");
        context.Response.End();
        return;
    }

    string file = context.Server.MapPath((string)o);
    string msg = ConfigurationManager.AppSettings["CopyrightNote"];
    if (File.Exists(file))
    {
        Bitmap bmp = AddCopyright(file, msg);
        context.Response.ContentType = "image/jpeg";
        bmp.Save(context.Response.OutputStream, ImageFormat.Jpeg);
        bmp.Dispose();
    }
    else
    {
        context.Response.Write("No image found.");
        context.Response.End();
    }
}
```

Note that the server-side page performs two different tasks indeed. First, it writes copyright text on the image canvas; next, it converts whatever the original format was to JPEG:

```
Bitmap AddCopyright(string file, string msg)
{
    // Load the file and create the graphics
    Bitmap bmp = new Bitmap(file);
    Graphics g = Graphics.FromImage(bmp);

    // Define text alignment
    StringFormat strFmt = new StringFormat();
    strFmt.Alignment = StringAlignment.Center;

    // Create brushes for the bottom writing
    // (green text on black background)
    SolidBrush btmForeColor = new SolidBrush(Color.PaleGreen);
    SolidBrush btmBackColor = new SolidBrush(Color.Black);

    // To calculate writing coordinates, obtain the size of the
    // text given the font typeface and size
    Font btmFont = new Font("Verdana", 7);
    SizeF textSize = new SizeF();
    textSize = g.MeasureString(msg, btmFont);

    // Calculate the output rectangle and fill
    float x = ((float) bmp.Width-textSize.Width-3);
    float y = ((float) bmp.Height-textSize.Height-3);
    float w = ((float) x + textSize.Width);
    float h = ((float) y + textSize.Height);
    RectangleF textArea = new RectangleF(x, y, w, h);
    g.FillRectangle(btmBackColor, textArea);

    // Draw the text and free resources
    g.DrawString(msg, btmFont, btmForeColor, textArea);
    btmForeColor.Dispose();
    btmBackColor.Dispose();
    btmFont.Dispose();
    g.Dispose();

    return bmp;
}
```

Figure 18-8 shows the results.

Note that the additional text is part of the image the user downloads on her client browser. If the user saves the picture by using the *Save Picture As* menu from the browser, the text (in this case, the copyright note) is saved along with the image.

FIGURE 18-8 A server-resident image has been modified before being displayed.

Note What if the user requests the JPG file directly from the address bar? And what if the image is linked by another Web site or referenced in a blog post? In these cases, the original image is served without any further modification. Why is it so? As mentioned, for performance reasons IIS serves static files, such as JPG images, directly without involving any external module, including the ASP.NET runtime. The HTTP handler that does the trick of adding a copyright note is therefore blissfully ignored when the request is made via the address bar or a hyperlink. What can you do about it?

In IIS 6.0, you must register the JPG extension as an ASP.NET extension for a particular application using the IIS Manager as shown in Figure 18-4. In this case, each request for JPG resources is forwarded to your application and resolved through the HTTP handler.

In IIS 7.0, things are even simpler for developers. All that you have to do is add the following lines to the application's *web.config* file:

```
<system.webServer>
    <handlers>
        <add verb="*"
            path="*.jpg"
            type="Core35.Components.DynImageHandler,Core35Lib" />
    </handlers>
</system.webServer>
```

The *system.webServer* section is a direct child of the root *configuration* node.

Advanced HTTP Handler Programming

HTTP handlers are not a tool for everybody. They serve a very neat purpose: changing the way a particular resource, or set of resources, is served to the user. You can use handlers to filter out resources based on runtime conditions or to apply any form of additional logic to the retrieval of traditional resources such as pages and images. Finally, you can use HTTP handlers to serve certain pages or resources in an asynchronous manner.

For HTTP handlers, the registration step is key. Registration enables ASP.NET to know about your handler and its purpose. Registration is required for two practical reasons. First, it serves to ensure that IIS forwards the call to the correct ASP.NET application. Second, it serves to instruct your ASP.NET application on the class to load to "handle" the request. As mentioned, you can use handlers to override the processing of existing resources (for example, *hello.aspx*) or to introduce new functionalities (for example, *folder.axd*). In both cases, you're invoking a resource whose extension is already known to IIS—the *.axd* extension is registered in the IIS metabase when you install ASP.NET. In both cases, though, you need to modify the *web.config* file of the application to let the application know about the handler.

By using the ASHX extension and programming model for handlers, you can also save yourself the *web.config* update and deploy a new HTTP handler by simply copying a new file in a new or existing application's folder.

Deploying Handlers as ASHX Resources

An alternative way to define an HTTP handler is through an *.ashx* file. The file contains a special directive, named *@WebHandler*, that expresses the association between the HTTP handler endpoint and the class used to implement the functionality. All *.ashx* files must begin with a directive like the following one:

```
<%@ WebHandler Language="C#" Class="Core35.Components.YourHandler" %>
```

When an *.ashx* endpoint is invoked, ASP.NET parses the source code of the file and figures out the HTTP handler class to use from the *@WebHandler* directive. This automation removes the need of updating the *web.config* file. Here's a sample *.ashx* file. As you can see, it is the plain class file plus the special *@WebHandler* directive:

```
<%@ WebHandler Language="C#" Class="MyHandler" %>

using System.Web;

public class MyHandler : IHttpHandler {

    public void ProcessRequest (HttpContext context) {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }
```

```
    public bool IsReusable {
        get {
            return false;
        }
    }
}
```

Note that the source code of the class can either be specified inline or loaded from any of the assemblies referenced by the application. When *.ashx* resources are used to implement an HTTP handler, you just deploy the source file, and you're done. Just as for XML Web services, the source file is loaded and compiled only on demand. Because ASP.NET adds a special entry to the IIS metabase for *.ashx* resources, you don't even need to enter changes to the Web server configuration.

Resources with an *.ashx* extension are handled by an HTTP handler class named *SimpleHandleFactory*. Note that *SimpleHandleFactory* is actually an HTTP handler factory class, not a simple HTTP handler class. We'll discuss handler factories in a moment.

The *SimpleHandleFactory* class looks for the *@WebHandler* directive at the beginning of the file. The *@WebHandler* directive tells the handler factory the name of the HTTP handler class to instantiate once the source code has been compiled.

Important You can build HTTP handlers both as regular class files compiled to an assembly and via *.ashx* resources. There's no significant difference between the two approaches except that *.ashx* resources, like ordinary ASP.NET pages, will be compiled on the fly upon the first request.

Prevent Access to Forbidden Resources

If your Web application manages resources of a type that you don't want to make publicly available over the Web, you must instruct IIS not to display those files. A possible way to accomplish this consists of forwarding the request to *aspnet_isapi* and then binding the extension to one of the built-in handlers—the *HttpForbiddenHandler* class:

```
<add verb="*" path="*.xyz" type="System.Web.HttpForbiddenHandler" />
```

Any attempt to access an *.xyz* resource results in an error message being displayed. The same trick can also be applied for individual resources served by your application. If you need to deploy, say, a text file but do not want to take the risk that somebody can get to them, add the following:

```
<add verb="*" path="yourFile.txt" type="System.Web.HttpForbiddenHandler" />
```

Should It Be Reusable or Not?

In a conventional HTTP handler, the *ProcessRequest* method takes the lion's share of the overall set of functionality. The second member of the *IHttpHandler* interface—the *IsReusable* property—is used only in particular circumstances. If you set the *IsReusable* property to return *true*, the handler is not unloaded from memory after use and is repeatedly used. Put another way, the Boolean value returned by *IsReusable* indicates whether the handler object can be pooled.

Frankly, most of the time it doesn't really matter what you return—be it *true* or *false*. If you set the property to return *false*, you require that a new object be allocated for each request. The simple allocation of an object is not a particularly expensive operation. However, the initialization of the handler might be costly. In this case, by making the handler reusable, you save much of the overhead. If the handler doesn't hold any state, there's no reason for not making it reusable.

In summary, I'd say that *IsReusable* should be always set to *true*, except when you have instance properties to deal with or properties that might cause trouble if used in a concurrent environment. If you have no initialization tasks, it doesn't really matter whether it returns *true* or *false*. As a margin note, the *System.Web.UI.Page* class—the most popular HTTP handler ever—sets its *IsReusable* property to *false*.

The key point to make is the following. Who's really using *IsReusable* and, subsequently, who really cares about its value?

Once the HTTP runtime knows the HTTP handler class to serve a given request, it simply instantiates it—no matter what. So when is the *IsReusable* property of a given handler taken into account? Only if you use an HTTP handler factory—that is, a piece of code that dynamically decides which handler should be used for a given request. An HTTP handler factory can query a handler to determine whether the same instance can be used to service multiple requests and thus optionally create and maintain a pool of handlers.

ASP.NET pages and ASHX resources are served through factories. However, none of these factories ever checks *IsReusable*. Of all the built-in handler factories in the whole ASP.NET platform, very few check the *IsReusable* property of related handlers. So what's the bottom line?

As long as you're creating HTTP handlers for AXD, ASHX, or perhaps ASPX resources, be aware that the *IsReusable* property is blissfully ignored. Do not waste your time trying to figure out the optimal configuration. Instead, if you're creating an HTTP handler factory to serve a set of resources, whether or not to implement a pool of handlers is up to you and *IsReusable* is the perfect tool for the job.

But when should you employ an HTTP handler factory? In all situations in which the HTTP handler class for a request is not uniquely identified. For example, for ASPX pages, you don't know in advance which HTTP handler type you have to use. The type might not even exist (in which case, you compile it on the fly). The HTTP handler factory is used whenever you need to apply some logic to decide which is the right handler to use. In other words, you need an HTTP handler factory when declarative binding between endpoints and classes is not enough.

HTTP Handler Factories

An HTTP request can be directly associated with an HTTP handler or with an HTTP handler factory object. An HTTP handler factory is a class that implements the *IHttpHandlerFactory* interface and is in charge of returning the actual HTTP handler to use to serve the request. The *SimpleHandlerFactory* class provides a good example of how a factory works. The factory is mapped to requests directed at *.ashx* resources. When such a request comes in, the factory determines the actual handler to use by looking at the *@WebHandler* directive in the source file.

In the .NET Framework, HTTP handler factories are used to perform some preliminary tasks on the requested resource prior to passing it on to the handler. Another good example of a handler factory object is represented by an internal class named *PageHandlerFactory*, which is in charge of serving *.aspx* pages. In this case, the factory handler figures out the name of the handler to use and, if possible, loads it up from an existing assembly.

HTTP handler factories are classes that implement a couple of methods on the *IHttpHandlerFactory* interface—*GetHandler* and *ReleaseHandler*, as shown in Table 18-3.

TABLE 18-3 Members of the *IHttpHandlerFactory* Interface

Method	Description
GetHandler	Returns an instance of an HTTP handler to serve the request
ReleaseHandler	Takes an existing HTTP handler instance and frees it up or pools it

The *GetHandler* method has the following signature:

```
public virtual IHttpHandler GetHandler(HttpContext context,
    string requestType, string url, string pathTranslated);
```

The *requestType* argument is a string that evaluates to *GET* or *POST*—the HTTP verb of the request. The last two arguments represent the raw URL of the request and the physical path behind it. The *ReleaseHandler* method is a mandatory override for any class that implements *IHttpHandlerFactory*; in most cases, it will just have an empty body.

The following listing shows a sample HTTP handler factory that returns different handlers based on the HTTP verb (*GET* or *POST*) used for the request:

```
class MyHandlerFactory : IHttpHandlerFactory
{
    public IHttpHandler GetHandler(HttpContext context,
        string requestType, String url, String pathTranslated)
    {
        // Feel free to create a pool of HTTP handlers here
        if(context.Request.RequestType.ToLower() == "get")
            return (IHttpHandler) new MyGetHandler();
        else if(context.Request.RequestType.ToLower() == "post")
            return (IHttpHandler) new MyPostHandler();
        return null;
    }

    public void ReleaseHandler(IHttpHandler handler)
    {
        // Nothing to do
    }
}
```

When you use an HTTP handler factory, it's the factory, not the handler, that needs to be registered with the ASP.NET configuration file. If you register the handler, it will always be used to serve requests. If you opt for a factory, you have a chance to decide dynamically and based on runtime conditions which handler is more appropriate for a certain request. In doing so, you can use the *IsReusable* property of handlers to implement a pool.

Asynchronous Handlers

An asynchronous HTTP handler is a class that implements the *IHttpAsyncHandler* interface. The system initiates the call by invoking the *BeginProcessRequest* method. Next, when the method ends, a callback function is automatically invoked to terminate the call. In the .NET Framework, the sole *HttpApplication* class implements the asynchronous interface. The members of *IHttpAsyncHandler* interface are shown in Table 18-4.

TABLE 18-4 Members of the *IHttpAsyncHandler* Interface

Method	Description
*BeginProcessReques*t	Initiates an asynchronous call to the specified HTTP handler
*EndProcessReques*t	Terminates the asynchronous call

The signature of the *BeginProcessRequest* method is as follows:

```
IAsyncResult BeginProcessRequest(HttpContext context,
    AsyncCallback cb, object extraData);
```

The *context* argument provides references to intrinsic server objects used to service HTTP requests. The second parameter is the *AsyncCallback* object to invoke when the asynchronous method call is complete. The third parameter is a generic cargo variable that contains any data you might want to pass to the handler.

Note An *AsyncCallback* object is a delegate that defines the logic needed to finish processing the asynchronous operation. A delegate is a class that holds a reference to a method. A delegate class has a fixed signature, and it can hold references only to methods that match that signature. A delegate is equivalent to a type-safe function pointer or a callback. As a result, an *AsyncCallback* object is just the code that executes when the asynchronous handler has completed its job.

The *AsyncCallback* delegate has the following signature:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

It uses the *IAsyncResult* interface to obtain the status of the asynchronous operation. To illustrate the plumbing of asynchronous handlers, I'll show you the pseudocode that the HTTP runtime employs when it deals with asynchronous handlers. The HTTP runtime invokes the *BeginProcessRequest* method as illustrated by the following pseudocode:

```
// Sets an internal member of the HttpContext class with
// the current instance of the asynchronous handler
context.AsyncAppHandler = asyncHandler;

// Invokes the BeginProcessRequest method on the asynchronous HTTP handler
asyncHandler.BeginProcessRequest(context, OnCompletionCallback, context);
```

The *context* argument is the current instance of the *HttpContext* class and represents the context of the request. A reference to the HTTP context is also passed as the custom data sent to the handler to process the request. The *extraData* parameter in the *BeginProcessRequest* signature is used to represent the status of the asynchronous operation. The *BeginProcessRequest* method returns an object of type *HttpAsyncResult*—a class that implements the *IAsyncResult* interface. The *IAsyncResult* interface contains a property named *AsyncState* that is set with the *extraData* value—in this case, the HTTP context.

The *OnCompletionCallback* method is an internal method. It gets automatically triggered when the asynchronous processing of the request terminates. The following listing illustrates the pseudocode of the *HttpRuntime* private method:

```
// The method must have the signature of an AsyncCallback delegate
private void OnHandlerCompletion(IAsyncResult ar)
{
    // The ar parameter is an instance of HttpAsyncResult
    HttpContext context = (HttpContext) ar.AsyncState;

    // Retrieves the instance of the asynchronous HTTP handler
    // and completes the request
    IHttpAsyncHandler asyncHandler = context.AsyncAppHandler;
    asyncHandler.EndProcessRequest(ar);

    // Finalizes the request as usual
    ...
}
```

The completion handler retrieves the HTTP context of the request through the *AsyncState* property of the *IAsyncResult* object it gets from the system. As mentioned, the actual object passed is an instance of the *HttpAsyncResult* class—in any case, it is the return value of the *BeginProcessRequest* method. The completion routine extracts the reference to the asynchronous handler from the context and issues a call to the *EndProcessRequest* method:

```
void EndProcessRequest(IAsyncResult result);
```

The *EndProcessRequest* method takes the *IAsyncResult* object returned by the call to *BeginProcessRequest*. As implemented in the *HttpApplication* class, the *EndProcessRequest* method does nothing special and is limited to throwing an exception if an error occurred.

Implementing Asynchronous Handlers

Asynchronous handlers essentially serve one particular scenario—when the generation of the markup is subject to lengthy operations, such as time-consuming database stored procedures or calls to Web services. In these situations, the ASP.NET thread in charge of the request is stuck waiting for the operation to complete. Because the thread is a valuable member of the ASP.NET thread pool, lengthy tasks are potentially the perfect scalability killer. However, asynchronous handlers are here to help.

The idea is that the request begins on a thread-pool thread, but that thread is released as soon as the operation begins. In *BeginProcessRequest*, you typically create your own thread and start the lengthy operation. *BeginProcessRequest* doesn't wait for the operation to complete; therefore, the thread is returned to the pool immediately.

There are a lot of tricky details that this bird's-eye description just omitted. In the first place, you should strive to avoid a proliferation of threads. Ideally, you should use a custom thread pool. Furthermore, you must figure out a way to signal when the lengthy operation has terminated. This typically entails creating a custom class that implements *IAsyncResult* and returning it from *BeginProcessRequest*. This class embeds a synchronization object—typically a *ManualResetEvent* object—that the custom thread carrying the work will signal upon completion.

In the end, building asynchronous handlers is definitely tricky and not for novice developers. Very likely, you are more interested in asynchronous pages than in asynchronous HTTP handlers—that is, the same mechanism but applied to *.aspx* resources. In this case, the "lengthy task" is merely the *ProcessRequest* method of the *Page* class. (Obviously, you configure the page to execute asynchronously only if the page contains code that might start I/O-bound and potentially lengthy operations.)

Starting with ASP.NET 2.0, you find ad hoc support for building asynchronous pages more easily and comfortably. An introductory but still practical chapter on asynchronous pages can be found in my book *Programming ASP.NET Applications—Advanced Topics* (Microsoft Press, 2006).

Warning I've seen several ASP.NET developers using an *.aspx* page to serve markup other than HTML markup. This is not a good idea. An .aspx resource is served by quite a rich and sophisticated HTTP handler—the *System.Web.UI.Page* class. The *ProcessRequest* method of this class entirely provides for the page life cycle as we know it—*Init*, *Load*, and *PreRender* events, as well as rendering stage, view state, and postback management. Nothing of the kind is really required if you only need to retrieve and return, say, the bytes of an image.

Writing HTTP Modules

So we've learned that any incoming requests for ASP.NET resources are handed over to the worker process for the actual processing within the context of the CLR. In IIS 6.0, the worker process is a distinct process from IIS, so if one ASP.NET application crashes, it doesn't bring down the whole server.

ASP.NET manages a pool of *HttpApplication* objects for each running application and picks up one of the pooled instances to serve a particular request. These objects are based on the class defined in your *global.asax* file, or on the base *HttpApplication* class if *global.asax* is missing. The ultimate goal of the *HttpApplication* object in charge of the request is getting an HTTP handler.

On the way to the final HTTP handler, the *HttpApplication* object makes the request pass through a pipeline of HTTP modules. An HTTP module is a .NET Framework class that implements the *IHttpModule* interface. The HTTP modules that filter the raw data within the request are configured on a per-application basis within the *web.config* file. All ASP.NET applications, though, inherit a bunch of system HTTP modules configured in the global *web.config* file.

Generally speaking, an HTTP module can pre-process and post-process a request, and it intercepts and handles system events as well as events raised by other modules. The highly-configurable nature of ASP.NET makes it possible for you to also write and register your own HTTP modules and make them plug into the ASP.NET runtime pipeline, handle system events, and fire their own events.

The *IHttpModule* Interface

The *IHttpModule* interface defines only two methods—*Init* and *Dispose*. The *Init* method initializes a module and prepares it to handle requests. At this time, you subscribe to receive notifications for the events of interest. The *Dispose* method disposes of the resources (all but memory!) used by the module. Typical tasks you perform within the *Dispose* method are closing database connections or file handles.

The *IHttpModule* methods have the following signatures:

```
void Init(HttpApplication app);
void Dispose();
```

The *Init* method receives a reference to the *HttpApplication* object that is serving the request. You can use this reference to wire up to system events. The *HttpApplication* object also features a property named *Context* that provides access to the intrinsic properties of the ASP.NET application. In this way, you gain access to *Response*, *Request*, *Session*, and the like.

Table 18-5 lists the events that HTTP modules can listen to and handle.

TABLE 18-5 *HttpApplication* Events

Event	Description
AcquireRequestState, *PostAcquireRequestState*	Occurs when the handler that will actually serve the request acquires the state information associated with the request. *The post event is not available in ASP.NET 1.x.*
AuthenticateRequest, *PostAuthenticateRequest*	Occurs when a security module has established the identity of the user. *The post event is not available in ASP.NET 1.x.*
AuthorizeRequest, *PostAuthorizeRequest*	Occurs when a security module has verified user authorization. *The post event is not available in ASP.NET 1.x.*
BeginRequest	Occurs as soon as the HTTP pipeline begins to process the request.
Disposed	Occurs when the *HttpApplication* object is disposed of as a result of a call to *Dispose*.
EndRequest	Occurs as the last event in the HTTP pipeline chain of execution.
Error	Occurs when an unhandled exception is thrown.
PostMapRequestHandler	Occurs when the HTTP handler to serve the request has been found. *The event is not available in ASP.NET 1.x.*
PostRequestHandlerExecute	Occurs when the HTTP handler of choice finishes execution. The response text has been generated at this point.
PreRequestHandlerExecute	Occurs just before the HTTP handler of choice begins to work.
PreSendRequestContent	Occurs just before the ASP.NET runtime sends the response text to the client.
PreSendRequestHeaders	Occurs just before the ASP.NET runtime sends HTTP headers to the client.
ReleaseRequestState, *PostReleaseRequestState*	Occurs when the handler releases the state information associated with the current request. *The post event is not available in ASP.NET 1.x.*

Event	Description
ResolveRequestCache, *PostResolveRequestCache*	Occurs when the ASP.NET runtime resolves the request through the output cache. *The post event is not available in ASP.NET 1.x.*
UpdateRequestCache, *PostUpdateRequestCache*	Occurs when the ASP.NET runtime stores the response of the current request in the output cache to be used to serve subsequent requests. *The post event is not available in ASP.NET 1.x.*

All these events are exposed by the *HttpApplication* object that an HTTP module receives as an argument to the *Init* method.

A Custom HTTP Module

Let's begin coming to grips with HTTP modules by writing a relatively simple custom module named *Marker* that adds a signature at the beginning and end of each page served by the application. The following code outlines the class we need to write:

```
using System;
using System.Web;

namespace Core35.Components
{
    public class MarkerModule : IHttpModule
    {
        public void Init(HttpApplication app)
        {
            // Register for pipeline events
        }

        public void Dispose()
        {
            // Nothing to do here
        }
    }
}
```

The *Init* method is invoked by the *HttpApplication* class to load the module. In the *Init* method, you normally don't need to do more than simply register your own event handlers. The *Dispose* method is, more often than not, empty. The heart of the HTTP module is really in the event handlers you define.

Wiring Up Events

The sample *Marker* module registers a couple of pipeline events. They are *BeginRequest* and *EndRequest*. *BeginRequest* is the first event that hits the HTTP application object when the request begins processing. *EndRequest* is the event that signals the request is going to be terminated, and it's your last chance to intervene. By handling these two events, you

can write custom text to the output stream before and after the regular HTTP handler—the *Page*-derived class.

The following listing shows the implementation of the *Init* and *Dispose* methods for the sample module:

```
public void Init(HttpApplication app)
{
    // Register for pipeline events
    app.BeginRequest += new EventHandler(OnBeginRequest);
    app.EndRequest += new EventHandler(OnEndRequest);
}

public void Dispose()
{
}
```

The *BeginRequest* and *EndRequest* event handlers have a similar structure. They obtain a reference to the current *HttpApplication* object from the sender and get the HTTP context from there. Next, they work with the *Response* object to append text or a custom header:

```
public void OnBeginRequest(object sender, EventArgs e)
{
    HttpApplication app = (HttpApplication) sender;
    HttpContext ctx = app.Context;

    // More code here
    ...

    // Add custom header to the HTTP response
    ctx.Response.AppendHeader("Author", "DinoE");

    // PageHeaderText is a constant string defined elsewhere
    ctx.Response.Write(PageHeaderText);
}

public void OnEndRequest(object sender, EventArgs e)
{
    // Get access to the HTTP context
    HttpApplication app = (HttpApplication) sender;
    HttpContext ctx = app.Context;

    // More code here
    ...

    // Append some custom text
    // PageFooterText is a constant string defined elsewhere
    ctx.Response.Write(PageFooterText);
}
```

OnBeginRequest writes standard page header text and also adds a custom HTTP header. *OnEndRequest* simply appends the page footer. The effect of this HTTP module is visible in Figure 18-9.

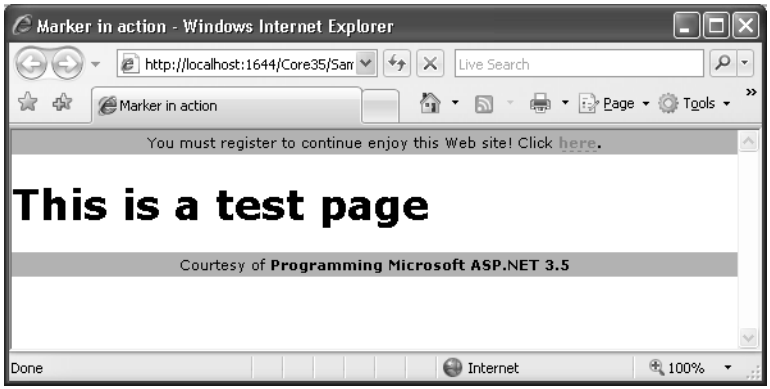

FIGURE 18-9 The *Marker* HTTP module adds a header and footer to each page within the application

Registering with the Configuration File

You register a new HTTP module by adding an entry to the <*httpModules*> section of the configuration file. The overall syntax of the <*httpModules*> section closely resembles that of HTTP handlers. To add a new module, you use the <*add*> node and specify the *name* and *type* attributes. The *name* attribute contains the public name of the module. This name is used to select the module within the *HttpApplication*'s *Modules* collection. If the module fires custom events, this name is also used as the prefix for building automatic event handlers in the *global.asax* file:

```
<system.web>
  <httpModules>
    <add name="Marker"
         type="Core35.Components.MarkerModule,Core35Lib" />
  </httpModules>
</system.web>
```

The *type* attribute is the usual comma-separated string that contains the name of the class and the related assembly. The configuration settings can be entered into the application's configuration file as well as into the global *web.config* file. In the former case, only pages within the application are affected; in the latter case, all pages within all applications are processed by the specified module.

The order in which modules are applied depends on the physical order of the modules in the configuration list. You can remove a system module and replace it with your own that provides a similar functionality. In this case, in the application's *web.config* file you use the <*remove*> node to drop the default module and then use <*add*> to insert your own. If you want to completely redefine the order of HTTP modules for your application, you can clear all the default modules by using the <*clear*> node and then re-register them all in the order you prefer.

Note HTTP modules are loaded and initialized only once, at the startup of the application. Unlike HTTP handlers, they apply to just any requests. So when you plan to create a new HTTP module, you should first wonder whether its functionality should span all possible requests in the application. Is it possible to choose which requests an HTTP module should process? The *Init* method is called only once in the application's lifetime; but the handlers you register are called once for each request. So to operate only on certain pages, you can do as follows:

```
public void OnBeginRequest(object sender, EventArgs e)
{
    HttpApplication app = (HttpApplication) sender;
    HttpContext ctx = app.Context;
    if (!ShouldHook(ctx))
         return;
    ...
}
```

OnBeginRequest is your handler for the *BeginRequest* event. The *ShouldHook* helper function returns a Boolean value. It is passed the context of the request—that is, any information that is available on the request. You can code it to check the URL as well as any HTTP content type and headers.

Accessing Other HTTP Modules

The sample just discussed demonstrates how to wire up pipeline events—that is, events fired by the *HttpApplication* object. But what about events fired by other modules? The *HttpApplication* object provides a property named *Modules* that gets the collection of modules for the current application.

The *Modules* property is of type *HttpModuleCollection* and contains the names of the modules for the application. The collection class inherits from the abstract class *NameObjectCollectionBase*, which is a collection of pairs made of a string and an object. The string indicates the public name of the module; the object is the actual instance of the module. To access the module that handles the session state, you need code like this:

```
SessionStateModule sess = app.Modules["Session"];
sess.Start += new EventHandler(OnSessionStart);
```

As mentioned, you can also handle events raised by HTTP modules within the *global.asax* file and use the *ModuleName_EventName* convention to name the event handlers. The name of the module is just one of the settings you need to define when registering an HTTP module.

The Page Refresh Feature

Let's examine a practical situation in which the ability to filter the request before it gets processed by an HTTP handler helps to implement a feature that would otherwise be impossible. The postback mechanism has a nasty drawback—if the user refreshes the currently displayed

page, the last action taken on the server is blindly repeated. If a new record was added as a result of a previous posting, for example, the application would attempt to insert an identical record upon another postback. Of course, this results in the insertion of identical records and should result in an exception. This snag has existed since the dawn of Web programming and was certainly not introduced by ASP.NET. To implement nonrepeatable actions, some countermeasures are required to essentially transform any critical server-side operation into an *idempotency*. In algebra, an operation is said to be *idempotent* if the result doesn't change regardless of how many times you execute it. For example, take a look at the following SQL command:

```
DELETE FROM employees WHERE employeeid=9
```

You can execute the command 1000 consecutive times, but only one record at most will ever be deleted—the one that satisfies the criteria set in the WHERE clause. Consider this command, instead:

```
INSERT INTO employees VALUES (...)
```

Each time you execute the command, a new record might be added to the table. This is especially true if you have auto-number key columns or nonunique columns. If the table design requires that the key be unique and specified explicitly, the second time you run the command a SQL exception would be thrown.

Although the particular scenario we considered is typically resolved in the data access layer (DAL), the underlying pattern represents a common issue for most Web applications. So the open question is, how can we detect whether the page is being posted as the result of an explicit user action or because the user simply hit F5 or the page refresh () toolbar button?

The Rationale Behind Page Refresh Operations

The page refresh action is a sort of internal browser operation for which the browser doesn't provide any external notification in terms of events or callbacks. Technically speaking, the page refresh consists of the "simple" reiteration of the latest request. The browser caches the latest request it served and reissues it when the user hits the page refresh key or button. No browsers that I'm aware of provide any kind of notification for the page refresh event—and if there are any that do, it's certainly not a recognized standard.

In light of this, there's no way the server-side code (for example, ASP.NET, classic ASP, or ISAPI DLLs) can distinguish a refresh request from an ordinary submit or postback request. To help ASP.NET detect and handle page refreshes, you need to build surrounding machinery that makes two otherwise identical requests look different. All known browsers implement the refresh by resending the last HTTP payload sent; to make the copy look different from the original, any extra service we write must add more parameters and the ASP.NET page must be capable of catching them.

I considered some additional requirements. The solution should not rely on session state and should not tax the server memory too much. It should be relatively easy to deploy and as unobtrusive as possible.

Outline of the Solution

The solution is based on the idea that each request will be assigned a ticket number and the HTTP module will track the last-served ticket for each distinct page it processes. If the number carried by the page is lower than the last-served ticket for the page, it can only mean that the *same* request has been served already—namely, a page refresh. The solution consists of a couple of building blocks: an HTTP module to make preliminary checks on the ticket numbers, and a custom page class that automatically adds a progressive ticket number to each served page. Making the feature work is a two-step procedure: first, register the HTTP module; second, change the base code-behind class of each page in the relevant application to detect browser refreshes.

The HTTP module sits in the middle of the HTTP runtime environment and checks in every request for a resource in the application. The first time the page is requested (when not posting back), there will be no ticket assigned. The HTTP module will generate a new ticket number and store it in the *Items* collection of the *HttpContext* object. In addition, the module initializes the internal counter of the last-served ticket to 0. Each successive time the page is requested, the module compares the last-served ticket with the page ticket. If the page ticket is newer, the request is considered a regular postback; otherwise, it will be flagged as a page refresh. Table 18-6 summarizes the scenarios and related actions.

TABLE 18-6 Scenarios and Actions

Scenario	Action
Page has no ticket associated: ■ No refresh	Counter of the last ticket served is set to 0. The ticket to use for the next request of the current page is generated and stored in *Items*.
Page has a ticket associated: ■ Page refresh occurs if the ticket associated with the page is lower than the last served ticket	Counter of the last ticket served is set with the ticket associated with the page. The ticket to use for the next request of the current page is generated and stored in *Items*.

Some help from the page class is required to ensure that each request—except the first—comes with a proper ticket number. That's why you need to set the code-behind class of each page that intends to support this feature to a particular class—a process that we'll discuss in a moment. The page class will receive two distinct pieces of information from the HTTP module—the next ticket to store in a hidden field that travels with the page, and whether or not the request is a page refresh. As an added service to developers, the code-behind class

will expose an extra Boolean property—*IsRefreshed*—to let developers know whether or not the request is a page refresh or a regular postback.

Important The *Items* collection on the *HttpContext* class is a cargo collection purposely created to let HTTP modules pass information down to pages and HTTP handlers in charge of physically serving the request. The HTTP module we employ here sets two entries in the *Items* collection. One is to let the page know whether the request is a page refresh; another is to let the page know what the next ticket number is. Having the module pass the page the next ticket number serves the purpose of keeping the page class behavior as simple and linear as possible, moving most of the implementation and execution burden on to the HTTP module.

Implementation of the Solution

There are a few open points with the solution I just outlined. First, some state is required. Where do you keep it? Second, an HTTP module will be called for each incoming request. How do you distinguish requests for the same page? How do you pass information to the page? How intelligent do you expect the page to be?

It's clear that each of these points might be designed and implemented in a different way than shown here. All design choices made to reach a working solution here should be considered arbitrary, and they can possibly be replaced with equivalent strategies if you want to rework the code to better suit your own purposes. Let me also add this disclaimer: I'm not aware of commercial products and libraries that fix this reposting problem. In the past couple of years, I've been writing articles on the subject of reposting and speaking at various user groups. The version of the code presented in this next example incorporates the most valuable suggestions I've collected along the way. One of these suggestions is to move as much code as possible into the HTTP module, as mentioned in the previous note.

The following code shows the implementation of the HTTP module:

```
public class RefreshModule : IHttpModule
{
    public void Init(HttpApplication app) {
        app.BeginRequest += new EventHandler(OnAcquireRequestState);
    }
    public void Dispose() {
    }
    void OnAcquireRequestState(object sender, EventArgs e) {
        HttpApplication app = (HttpApplication) sender;
        HttpContext ctx = app.Context;
        RefreshAction.Check(ctx);
        return;
    }
}
```

The module listens to the *BeginRequest* event and ends up calling the *Check* method on the helper *RefreshAction* class:

```
public class RefreshAction
{
    static Hashtable requestHistory = null;

    // Other string constants defined here
    ...

    public static void Check(HttpContext ctx) {
        // Initialize the ticket slot
        EnsureRefreshTicket(ctx);

        // Read the last ticket served in the session (from Session)
        int lastTicket = GetLastRefreshTicket(ctx);

        // Read the ticket of the current request (from a hidden field)
        int thisTicket = GetCurrentRefreshTicket(ctx, lastTicket);

        // Compare tickets
        if (thisTicket > lastTicket ||
        (thisTicket==lastTicket && thisTicket==0)) {
            UpdateLastRefreshTicket(ctx, thisTicket);
            ctx.Items[PageRefreshEntry] = false;
        }
        else
            ctx.Items[PageRefreshEntry] = true;
    }

    // Initialize the internal data store
    static void EnsureRefreshTicket(HttpContext ctx)
   {
        if (requestHistory == null)
            requestHistory = new Hashtable();
    }

    // Return the last-served ticket for the URL
    static int GetLastRefreshTicket(HttpContext ctx)
    {
        // Extract and return the last ticket
        if (!requestHistory.ContainsKey(ctx.Request.Path))
            return 0;
        else
            return (int) requestHistory[ctx.Request.Path];
    }

    // Return the ticket associated with the page
    static int GetCurrentRefreshTicket(HttpContext ctx, int lastTicket)
    {
        int ticket;
        object o = ctx.Request[CurrentRefreshTicketEntry];
        if (o == null)
            ticket = lastTicket;
        else
            ticket = Convert.ToInt32(o);
```

```
        ctx.Items[RefreshAction.NextPageTicketEntry] = ticket + 1;
        return ticket;
    }

    // Store the last-served ticket for the URL
    static void UpdateLastRefreshTicket(HttpContext ctx, int ticket)
    {
         requestHistory[ctx.Request.Path] = ticket;
    }
}
```

The *Check* method performs the following actions. It compares the last-served ticket with the ticket (if any) provided by the page. The page stores the ticket number in a hidden field that is read through the *Request* object interface. The HTTP module maintains a hashtable with an entry for each distinct URL served. The value in the hashtable stores the last-served ticket for that URL.

Note The *Item* indexer property is used to set the last-served ticket instead of the *Add* method because *Item* overwrites existing items. The *Add* method just returns if the item already exists.

In addition to creating the HTTP module, you also need to arrange a page class to use as the base for pages wanting to detect browser refreshes. Here's the code:

```
// Assume to be in a custom namespace
public class Page : System.Web.UI.Page
{
  public bool IsRefreshed {
    get {
      HttpContext ctx = HttpContext.Current;
      object o = ctx.Items[RefreshAction.PageRefreshEntry];
      if (o == null)
        return false;
      return (bool) o;
    }
  }

  // Handle the PreRenderComplete event
  protected override void OnPreRenderComplete(EventArgs e) {
    base.OnPreRenderComplete(e);
    SaveRefreshState();
  }

  // Create the hidden field to store the current request ticket
  private void SaveRefreshState() {
    HttpContext ctx = HttpContext.Current;
    int ticket = (int) ctx.Items[RefreshAction.NextPageTicketEntry];
    ClientScript.RegisterHiddenField(
      RefreshAction.CurrentRefreshTicketEntry,
      ticket.ToString());
  }
}
```

The sample page defines a new public Boolean property *IsRefreshed* that you can use in code in the same way you would use *IsPostBack* or *IsCallback*. It overrides *OnPreRenderComplete* to add the hidden field with the page ticket. As mentioned, the page ticket is received from the HTTP module through an ad hoc (and arbitrarily named) entry in the *Items* collection.

Figure 18-10 shows a sample page in action. Let's take a look at the source code of the page.

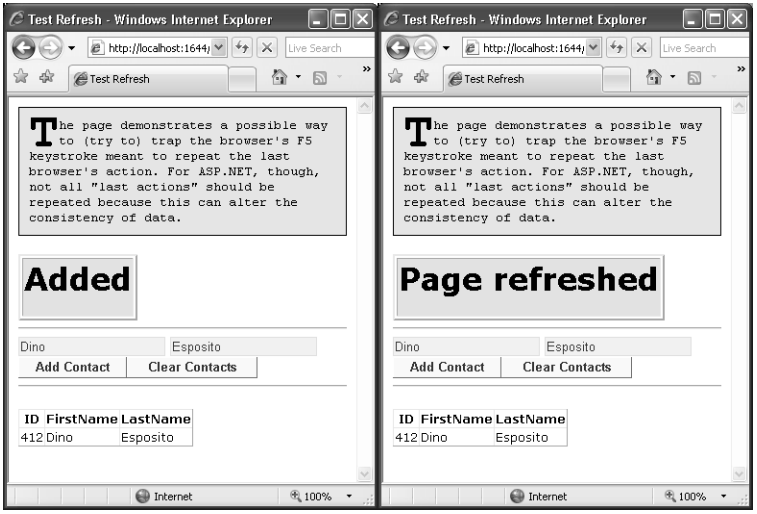

FIGURE 18-10 The page doesn't repeat a sensitive action if the user refreshes the browser's view.

```
public partial class TestRefresh : Core35.Components.Page
{
    protected void AddContactButton_Click(object sender, EventArgs e)
    {
        Msg.InnerText = "Added";
        if (!this.IsRefreshed)
            AddRecord(FName.Text, LName.Text);
        else
            Msg.InnerText = "Page refreshed";

        BindData();
    }
    ...
}
```

The *IsRefreshed* property lets you decide what to do when a postback action is requested. In the preceding code, the *AddRecord* method is not invoked if the page is refreshing. Needless to say, *IsRefreshed* is available only with the custom page class presented here. The custom page class doesn't just add the property, it also adds the hidden field, which is essential for the machinery to work.

Conclusion

HTTP handlers and HTTP modules are the building blocks of the ASP.NET platform. ASP.NET includes several predefined handlers and HTTP modules, but developers can write handlers and modules of their own to perform a variety of tasks. HTTP handlers, in particular, are faster than ordinary Web pages and can be used in all circumstances in which you don't need state maintenance and postback events. To generate images dynamically on the server, for example, an HTTP handler is more efficient than a page.

Everything that occurs under the hood of the ASP.NET runtime environment occurs because of HTTP handlers. When you invoke a Web page or an ASP.NET Web service method, an appropriate HTTP handler gets into the game and serves your request. At the highest level of abstraction, the behavior of an HTTP handler closely resembles that of an ISAPI extension. While the similarity makes sense, a key difference exists. HTTP handlers are managed and CLR-resident components. The CLR, in turn, is hosted by the worker process. An ISAPI extension, on the other hand, is a Win32 library that can live within the IIS process. In the ASP.NET process model, the *aspnet_isapi* component is a true ISAPI extension that collects requests and dispatches them to the worker process. ASP.NET internally implements an ISAPI-like extensibility model in which HTTP handlers play the role of ISAPI extensions in the IIS world. This model changes in IIS 7.0, at which point managed HTTP modules and extensions will also be recognized within the IIS environment.

HTTP modules are to ISAPI filters what HTTP handlers are to ISAPI extensions. HTTP modules are good at performing a number of low-level tasks for which tight interaction and integration with the request/response mechanism is a critical factor. Modules are sort of interceptors that you can place along an HTTP packet's path, from the Web server to the ASP.NET runtime and back. Modules have read and write capabilities, and they can filter and modify the contents of both inbound and outbound requests.

Just the Facts

- HTTP handlers and modules are like classic ISAPI extensions and filters except that they are managed components and provide a much simpler, less error-prone programming model.
- An HTTP handler is the ASP.NET component in charge of handling a request. In the end, an ASP.NET page is just an instance of an HTTP handler.
- HTTP handlers are classes that implement the *IHttpHandler* interface and take care of processing the payload of the request.
- HTTP modules are classes that implement the *IHttpModule* interface and listen to application-level events.
- Custom HTTP handlers and modules must be registered with the application, or all applications in the server machine, through special sections in the *web.config* file.