Chapter 3—Modules and Handlers

Modules and Handlers

In this chapter, we will look more closely at the extensibility of ASP.NET through the addition of custom modules and handlers. Handlers and modules are integrated into IIS so that your web applications can perform and scale well. In the previous chapter, you learned how to use asynchronous handlers to handle custom threads. Here, we’ll learn more about how ASP.NET can be extended using handlers. The objectives of this chapter are:

\* Learn what internal modules are for

\* Find out how to create, activate, and debug custom modules

\* Discover the handlers included in ASP.NET

\* Extend, customize, and replace handlers

\* Write your own handlers with both synchronous and asynchronous behavior

## Module, handlers and IIS

IIS7 Web server features fit into one of two categories:

\* Modules

\* Handlers

Similar to the *ISAPI filter* in previous IIS versions, a module participates in the processing of each request. Its role is to change or add content to the request. Examples of some out-of-the-box modules in IIS7 include authentication modules, compression modules, and logging modules. The names indicate the function of each module.

A module is a .NET class that implements the System.Web.IHttpModule interface and uses APIs in the System.Web namespace to participate in one or more of ASP.NET’s request processing stages. I explained the stages of this ‘pipeline’ in chapter 1.

By contrast, a handler, similar to the *ISAPI extension* in previous IIS versions, is responsible for handling requests and creating responses for specific content types. The main difference between modules and handlers is that handlers are typically mapped to a particular request path or extension. They also support the processing of a specific resource to which that path or extension corresponds. Handlers provided with IIS7 include ASP.NET’s PageHandlerFactory, which processes .aspx pages, among others. This kind of a handler is a .NET class that implements the ASP.NET System.Web.IHttpHandler or System.Web.IHttpAsyncHandler interface. It uses APIs in the System.Web namespace to produce an HTTP response for the specific content it creates.

When developing an IIS7 feature or ASP.NET extension, you’ll need to decide whether a module or a handler is appropriate. No common task requires both. If your feature is responsible for serving requests to a specific URL or file extension, like \*.png, then a handler is the right choice as handlers are primarily for specific tasks. Alternatively, if you want to respond to some or all requests, a module is appropriate.

Creating images on the fly corresponds to a specific file type—use a handler to achieve this. Adding footers to all your pages from one location is a good idea—implement a module to do that.

## Modules

This section explains how to create internal modules and gives some examples that you can use in your own applications.

### IIS7 Architecture

ASP.NET is tightly integrated with IIS7. Even though it’s possible to run ASP.NET with any host, thanks to its modular architecture, you should keep in mind that IIS is the best platform “by design”. Extending and customizing ASP.NET is only possible with a good understanding of IIS and its parts.

Microsoft changed large parts of the architecture of IIS7 compared to previous versions. One of the major changes was the greatly enhanced extensibility. Instead of a powerful but monolithic Web server, with IIS7 there is now a Web server engine to which you can add or remove components. These components are called modules.

Modules build the features offered by the Web server. All modules have one primary task—processing a request. This can become complicated, however, as a request isn’t just a call to static resources. Consider requests involving the authentication of client credentials, compression and decompression, or cache management.

Assuming that IIS7 is the primary platform for running ASP.NET, any discussion about extensibility does not make sense without understanding what accompanies IIS7. IIS7 comes with two module types:

\* Native Modules

\* Managed Modules

#### Native Modules

Native modules perform all the basic tasks of a Web server. However, not all modules manage common requests. It depends on your installation and configuration as to whether a module is available and running. Inside IIS7 are:

\* HTTP Modules

\* Security Modules

\* Content Modules

\* Compression Modules

\* Caching Modules

\* Logging and Diagnosing Modules

\* Integration of Managed Modules

You can find all modules—apart from those whose full path is given below—in the following directory:

%WinDir%\System32\InetSrv

The HTTP modules perform tasks specific to Hypertext Transfer Protocol.

Table 3-1. HTTP Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| CustomErrorModule | Custerr.dll | Sends default and configured HTTP error messages when an error status code is set on a response. |
| HttpRedirectionModule | Redirect.dll | Supports configurable redirection for HTTP requests. |
| ProtocolSupportModule | Protsup.dll | Performs protocol-related actions, such as setting response headers and redirecting headers based on configuration. |

Security is essential for a Web server, as shown by the number of modules in this section.

Table 3-2. Security Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| AnonymousAuthenticationModule | Authanon.dll | Performs anonymous authentication when no other authentication method succeeds. |
| BasicAuthenticationModule | Authbas.dll | Performs Basic authentication. |
| CertificateMappingAuthenticationModule | Authcert.dll | Performs Certificate Mapping authentication using Active Directory. |
| DigestAuthenticationModule | Authmd5.dll | Performs Digest authentication. |
| IISCertificateMappingAuthenticationModule | Authmap.dll | Performs Certificate Mapping authentication using IIS certificate configuration. |
| RequestFilteringModule | Modrqflt.dll | Performs URLScan tasks such as configuring allowed verbs and file extensions, setting limits, and scanning for bad character sequences. |
| UrlAuthorizationModule | Urlauthz.dll | Performs URL authorization. |
| WindowsAuthenticationModule | Authsspi.dll | Performs NTLM integrated authentication. |
| IpRestrictionModule | iprestr.dll | Restricts IPv4 addresses listed in the IpSecurity list in configuration. |

Once the request is accepted and authorized, the requested resources must be handled. Several modules perform these specific tasks.

Table 3-3. Content Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| CgiModule | Cgi.dll | Executes Common Gateway Interface (CGI) processes to build response output. |
| DefaultDocumentModule | Defdoc.dll | Attempts to return a default document for requests made to the parent directory. |
| DirectoryListingModule | dirlist.dll | Lists the contents of a directory. |
| IsapiModule | Isapi.dll | Hosts ISAPI extension DLLs. |
| IsapiFilterModule | Filter.dll | Supports ISAPI filter DLLs. |
| ServerSideIncludeModule | Iis\_ssi.dll | Processes server-side includes code. |
| StaticFileModule | Static.dll | Serves static files. |
| FastCgiModule | iisfcgi.dll | Supports FastCGI, which provides a high-performance alternative to CGI. |

Compression is a common way to save bandwidth and transfer large files more efficiently.

Table 3-4. Compression Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| DynamicCompressionModule | Compdyn.dll | Compresses responses using *gzip* compression transfer coding on-the-fly. |
| StaticCompressionModule | Compstat.dll | Performs pre-compression of static content. |

Caching is another way to improve performance. Several modules store files so that the delivery process is either accelerated or eliminated altogether.

Table 3-5. Caching Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| FileCacheModule | Cachfile.dll | Provides user mode caching for files and file handles. |
| HTTPCacheModule | Cachhttp.dll | Provides kernel mode and user mode caching in HTTP.sys. |
| TokenCacheModule | Cachtokn.dll | Provides user mode caching of user name and token pairs for modules that produce Windows user principals. |
| UriCacheModule | Cachuri.dll | Provides user mode caching of URL information. |

Knowing what is happening is essential for Web server administrators. There are several steps from the code on your server—where you probably have debug capabilities—to the browser, where you might miss seeing the desired output. Even production systems do not always behave as expected. Logging and diagnostic modules help you understand the internal processing of requests and responses.

Table 3-6. Logging and Diagnostic Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL | Description |
| CustomLoggingModule | Logcust.dll | Loads custom logging modules. |
| FailedRequestsTracingModule | Iisfreb.dll | Supports the Failed Request Tracing feature. |
| HttpLoggingModule | Loghttp.dll | Passes information and processing status to HTTP.sys for logging. |
| RequestMonitorModule | Iisreqs.dll | Tracks requests currently executing in worker processes and reports information with Runtime Status and Control Application Programming Interface (RSCA). |
| TracingModule | Iisetw.dll | Reports events to Microsoft Event Tracing for Windows (ETW). |

Finally, you need an interface to the managed world. In chapter 1, you saw how modules interacted. The next table shows which modules are involved in this interaction.

Table 3-7. Integration of Managed Modules

|  |  |  |
| --- | --- | --- |
| Module Name | DLL/Assembly | Description |
| ManagedEngine | Microsoft.NET\Framework\v2.0.50727\webengine.dll | Provides integration of managed code modules in the IIS request-processing pipeline. |
| ConfigurationValidationModule | validcfg.dll | Validates configuration issues, such as when an application is running in Integrated mode but has handlers or modules declared in the system.web section. |

#### Managed Modules

While extending IIS7 via native modules is one method of writing high performance applications, for most projects, this is not necessary or desirable. Extending ASP.NET using managed modules has significant advantages in development time and reliability. IIS7 includes several built-in, managed modules, which show that it is possible to write low-level infrastructure components in managed code.

You can find the definition of security modules in the System.Web.Security namespace.

Table 3-8. Managed code Security Modules shipped with IIS7

|  |  |  |
| --- | --- | --- |
| Module Name | Class | Description |
| AnonymousIdentification | AnonymousIdentificationModule | Manages anonymous identifiers, which are used by features that support anonymous identification such as the ASP.NET profile. |
| DefaultAuthentication | DefaultAuthenticationModule | Ensures that an authentication object is present in the context. |
| FileAuthorization | FileAuthorizationModule | Verifies that a user has permission to access the requested file. |
| FormsAuthentication | FormsAuthenticationModule | Supports authentication using Forms authentication. |
| RoleManager | RoleManagerModule | Manages a RolePrincipal instance for the current user. |
| UrlAuthorization | UrlAuthorizationModule | Determines whether the current user is permitted access to the requested URL, based on the user name or membership of a suitable role. |
| WindowsAuthentication | WindowsAuthenticationModule | Sets the identity of the user for an ASP.NET application when Windows authentication is enabled. |

The definition of the cache module is stored in System.Web.Caching. This module is declared as internal and is not accessible by user code.

Table 3-9. Managed code Caching Modules shipped with IIS7

|  |  |  |
| --- | --- | --- |
| Module Name | Class | Description |
| OutputCache | OutputCacheModule | Supports output caching. |

The module for profile handling is in the System.Web.Profile namespace.

Table 3-10. Managed code Profiles Modules shipped with IIS7

|  |  |  |
| --- | --- | --- |
| Module Name | Class | Description |
| Profile | ProfileModule | Manages user profiles by using the ASP.NET profile, which stores and retrieves user settings in a data source such as a database. |

For session handling modules, look into the System.Web.SessionState namespace.

Table 3-11. Managed code Session Modules shipped with IIS7

|  |  |  |
| --- | --- | --- |
| Module Name | Class | Description |
| Session | SessionStateModule | Supports the maintenance of a session state, which enables the storage of data specific to a single client within an application on the server. |

The modules for URL handling are stored in the root namespace System.Web.

Table 3-12. Managed code URL Modules shipped with IIS7

|  |  |  |  |
| --- | --- | --- | --- |
| Module Name | Class | Description |  |
| UrlMappingsModule | UrlMappingsModule | Supports mapping a real URL to a more user-friendly URL. |  |

Given this list of modules, you might feel that there is no need to write custom modules to perform standard tasks. This is true; the developers of ASP.NET and IIS have delivered everything you need for common infrastructure work. However, if you’d like to program an application-specific task, writing your own module is an excellent way to extend ASP.NET and add sophisticated features of your choice.

### The IIS7 Managed Module Starter Kit

Microsoft provides a starter kit to make it easy to write your first module for the new IIS7 managed API. This Visual Studio Content Installer contains a project template for building IIS7 Modules using the .NET Framework.

#### Get the Starter Kit

The Starter Kit is available at no cost from Microsoft at the www.iis.net website:

http://www.iis.net/downloads/default.aspx?tabid=34&i=1302&g=6

Click on Download, save the file and unzip to a current folder. The kit is a Visual Studio template installer, provided as a .vsi file.

#### Benefits of Module Development Kit

Since IIS7 supports development using managed code, it means you can program HTTP request processing in managed code. The entire event structure of IIS7, written in native API (C/C++), is also available to managed code developers.

Figure 3-1.Installing the template

Note: Using the Starter Kit simplifies the first steps. However, you don’t need it to run the samples provided in this chapter.

Once the template is installed, you can add new modules by using the right item for your project.

Insert ASPEXTf0302.tif

Figure 3-2. Adding a new module code item to a current project

This item creates a class skeleton, which looks like:

Listing 3-1. Skeleton of a module class

using System;

using System.Web;

namespace Apress.HttpModules

{

public class DirListModule : IHttpModule

{

/// <summary>

/// You will need to configure this module in the web.config file

/// and register it with IIS before being able to use it.   
 /// For more information

/// see the following link: http://go.microsoft.com/?linkid=8101007

/// </summary>

#region IHttpModule Members

public void Dispose()

{

//clean-up code here.

}

public void Init(HttpApplication context)

{

// Below is an example of how you can handle LogRequest   
 // event and provide

// custom logging implementation for it

context.LogRequest += new EventHandler(OnLogRequest);

}

#endregion

public void OnLogRequest(Object source, EventArgs e)

{

//custom logging logic can go here

}

}

}

For the sake of clarity and space, I will not repeat this part of the module code in the following sections when examining the various examples.

### Building a Module

As requests move through the pipeline, a number of events fire on the HttpApplication object. As you’ve seen already, these events are published as event methods in *Global.asax*. This approach is application-specific, but not always ideal. If you want to build generic HttpApplication event hooks that can be plugged into any Web application, you can use HttpModules. These are reusable and require an entry in *web.config* instead of application-specific code.

#### Define the Modules

Modules allow you to hook events for any request that passes through the ASP.NET HttpApplication object. These modules are stored as classes in external assemblies configured in *web.config*, which causes them to load when the application starts. By implementing specific interfaces and methods, the module is hooked up to the HttpApplication event chain. Multiple HttpModules can hook the same event. Their order is determined by their order of appearance in *web.config*. A handler definition in *web.config* looks like:

Listing 3-2. Register a module in web.config

<configuration>

<system.web>

<httpModules>

<add name= "MyModule" type="Apress.HttpModules.Modules,MyModule" />

</httpModules>

</system.web>

</configuration>

Note that you need to specify both a full type name and an assembly name without the DLL extension.

Modules allow you observe each incoming request and perform an action based on the events that fire. Modules are excellent for modifying a request or responding to particular content in order to provide custom authentication or pre- or post-processing to each request that arrives.

Many of ASP.NET’s features, such as the Authentication and Session engines, are implemented as HTTP Modules. While HttpModules might feel similar to *ISAPI Filters* in that they examine every request that arrives through an ASP.NET Application, in reality they are limited to scrutinizing requests mapped to a single specific ASP.NET application or virtual directory.

You can thus inspect all .aspx pages or any other custom extensions that are mapped to this application. However, you cannot look at standard .html or image files unless you explicitly map the extension to the ASP.NET ISAPI.dll.

#### Example—Write a Simple Authentication Module

This first example shows how to intercept the authentication procedure of the request pipeline. It also demonstrates how to add your own authentication module to handle tasks independently of the existing code.

Implementing an HTTP Module is very easy. Implement the IHttpModule interface, which contains two methods: Init and Dispose.

Note: The Starter Kit item additionally creates a LogRequest handler. You can safely remove the handler and event assignment if you don’t need it.

The event parameters passed include a reference to the HTTPApplication object, which in turn gives you access to the HttpContext object. Using the Init method, you can hook up to HttpApplication events. For example, if you want to hook the AuthenticateRequest event to a module you would do so as shown in Listing 3-3.

Listing 3-3. Simple implementation of an HTTP Module

public class BasicAuthCustomModule : IHttpModule

{

public void Init(HttpApplication application)

{

application.AuthenticateRequest += ⮰

new EventHandler(this.OnAuthenticateRequest);

}

public void Dispose() { }

public void OnAuthenticateRequest(object source, Even-tArgs eventArgs)

{

HttpApplication app = (HttpApplication) source;

HttpContext Context = HttpContext.Current;  
 // and action

}

Remember that your Module has access to the HttpContext object and from there to all other intrinsic ASP.NET pipeline objects, such as Response and Request. From here, you can retrieve input, create content, and so forth. However, keep in mind that certain things may not be available until further down the chain.

You can hook as many events as you like in the Init method so that your module is able to manage multiple operations with different functions. It is tidier to separate differing logic into separate modules. This ensures that the modules are, indeed, modular, as their name implies. In many cases, any functionality that you implement could require hooking multiple events. For example, a logging filter might log the start time of a request in BeginRequest and then write the request completion into the log in EndRequest.

Caution: Modules work deep inside the processing pipeline. Calling certain methods can prevent the pipeline from proceeding to the next step. In particular, Response.End and Application.CompleteRequest complete the request and force the pipeline to end, thus skipping all subsequent steps. The pipeline will return control to the Web server and no further modules will be invoked. A better practice is to leave the pipeline running, but use a context variable to inform subsequent modules not to execute.

The purpose of the Dispose method is to clean up any resources when the module unloads and to release other resources before the garbage collector finalizes the module instance. If there is nothing to dispose, leave the method body blank.

The Init method is the main method of interest. Here, you can initialize your module and wire it up to one or more request processing events available on the HttpApplication class. Keep in mind that events fire when the appropriate step in the pipeline is reached, in a defined order and in conjunction with other modules. Without a clear understanding of the pipeline architecture explained in chapter 1, you’ll have difficulty writing sophisticated modules which perform well.

#### Example—Check for a Specific Header

The following example checks for a specific header, called a referrer, which provides information about the referring page. The referrer is, paradoxically, named “referer”. Don’t worry about the misspelled word (the sidebar explains more).

Wikipedia on the Word, Referer

*Referer* is a common misspelling of the word *referrer*. It is so common, in fact, that it made it into the official specification of HTTP – the communication protocol of the World Wide Web – and has therefore become a widely used industry spelling when discussing HTTP referrers. The misspelling usage is not universal; the correct spelling of "referrer" occurs in some web specifications such as the Document Object Model. [Source: http://en.wikipedia.org/wiki/Referer]

I advise you to pay attention to which word you’re using!

In this example, we’ll look for a specific referrer or referring page—the URL of the previous page. (It’s the page containing the link to the page we’re currently processing.) The usage of the header is voluntary, according to HTTP standards, but most sites use it to track users or manage logging. However, some pages are not intended to be called from outside our site. If linked to from another page within our application, such pages will execute correctly, but if linked to from anywhere else, we’ll treat that as an exception. Such external links typically occur when a user bookmarks a specific page deep within your application. When attempting to open one of these pages, it is not possible, because there are several prerequisite steps to be completed beforehand. Using a module like the one in Listing 3-4, you can capture these requests outside the common page code and redirect users to a suitable page, such as one which explains appropriate usage of bookmarks.

Listing 3-4. An HTTP Module that looks for the Referer Header

public class ReferrerModule : IHttpModule

{

#region IHttpModule Members

public void Dispose()

{

//clean-up code here.

}

public void Init(HttpApplication context)

{

context.PreRequestHandlerExecute += ⮰  
 new EventHandler(context\_PreRequestHandlerExecute);

}

void context\_PreRequestHandlerExecute(object sender, EventArgs e)

{

HttpApplication app = (HttpApplication)sender;

HttpRequest request = app.Context.Request;

if (!request.Url.LocalPath.EndsWith("Default.aspx"))

{

if (String.IsNullOrEmpty(request.Headers["Referer"]))

{

throw new HttpException(403, "Bookmarking is not allowed");

}

}

}

#endregion

}

This code assumes that you have a page called Default.aspx linking to another page in your application. The name of the other page doesn’t matter.

To test this module:

1. Configure *web.config* to activate the module.

2. Create two pages, Default.aspx and RefererTest.aspx. Default.aspx has a hyperlink to RefererTest.aspx.

3. Compile and start the application by launching the Default.aspx page.

4. Click the hyperlink on Default.aspx—the RefererTest.aspx page is displayed.

5. Bookmark the RefererTest.aspx page.

6. Close your browser, re-open it, and load the bookmark, then press F5 to force a refresh of the page from the server.

7. An exception occurs and the browser shows a 403 error.

You might insist that this all can be accomplished on the page level using conventional code. This is correct, but fundamental tasks are best handled on a fundamental level. In addition, intercepting low-level events to handle low-level action is faster, more secure, and more reliable. Adding more pages with the same behavior does not require any change to the code. It simply works because the module tests all pages in the application.

### Interaction between Modules

Writing private modules is a powerful technique for extending ASP.NET. However, extending can mean replacing existing functionality. Sometimes a smart solution results simply from using the internal modules and your own module together.

To begin with, you’ll need access to the internal modules at runtime. You should make connections in the Init event in order to have access at an early stage. The next example shows you how to retrieve information about internal modules and other kinds of modules attached to the pipeline so far.

Listing 3-5. Retrieving information about modules

public class SessionLogModule : IHttpModule

{

#region IHttpModule Members

public void Dispose()

{

}

public void Init(HttpApplication application)

{

HttpContext context = HttpContext.Current;

foreach (string key in application.Modules.AllKeys)

{

context.Response.Write(String.Format("{0}= {1} {2}<br>",

key,

application.Modules[key].GetType().IsPublic ? "public" : "internal",

application.Modules[key].GetType().AssemblyQualifiedName));

}

}

#endregion

}

The current context is used to output the text directly into the current page where the request is handled. You can use the Modules property to get a list of the modules and where they are defined.

Figure 3-3. Modules already available

You probably need to use this method of getting access to an embedded module, as not all modules offer direct access to their states and events. Once you know the name and type of a specific module, you can cast the type and get the object you need.

### Configuration and Deployment

Now that the module is implemented, we can compile it into an assembly that ASP.NET is able to load at runtime. This is straightforward as long as the module is in the web application. No special action is required. You will probably want to create several modules and have them in different assemblies for easy reuse. The assemblies will need to be referenced by your web project. To construct such a module, choose “Class Library” as the project template. Remove the default class created by the template, and add an object of type “ASP.NET Module”.

Figure 3-4. Add a module to current project

#### Configuring the Default Web Server and Development Environment

To test the module, you’ll need to configure the settings in *web.config*. Place the appropriate lines in the <system.web> section:

<httpModules>

<add name="ReferrerModule"

type="Apress.AspNetExtensibility.HttpModules.ReferrerModule "/>

</httpModules>

The settings for the development environment also apply for IIS5, IIS6, and IIS7 in classic mode. There are several advantages to running the IIS7 integrated pipeline, however, which requires different settings:

Table 3-13. Options of the httpModule settings

Attribute Typical Values Description

name any string The module name that appears in settings dialogs

type class, assembly Module type

#### Configuring IIS7 Settings

In the main (web) project, add a reference to the project containing the module. Assuming the namespace of the external project is Apress.HttpHandler.ImageHandler, add the following to *web.config*:

<system.webServer>

<modules>

<add name="ReferrerModule"

type="Apress.HttpHandler.ImageHandler" resourceType="File"

requireAccess="Read" preCondition="integratedMode" />

</modules>

</system.webServer>

Compile both the project containing the module and the web project. Add the mapping in Internet Information Services Manager, as shown before. The mapping will now function perfectly for both the development environment and direct usage from the local IIS7.

Table 3-14. Options for the handler settings for IIS7 integrated mode

Attribute Typical Values Description

name any string The module name that appears in settings dialogs

precondition string Name of another handler or module required before this one

type class, assembly Type information of the handlers definition

#### Configure using IIS Management Console

Rather than adding the IIS7 integrated mode settings to *web.config* you can simply use the IIS Management Console. The settings correspond directly. Altering *web.config* will result in an immediate change to the Management Console settings, and vice versa. To configure using the IIS Management Console:

1. Open Internet Information Service Manager.

2. Open the web you want to change.

3. In the IIS section, double click on the Modules icon.

4. Click on “Add managed module” in the task list to the right.

5. Enter these values in the dialog:

a. Give the module an appropriate name

b. Open the type drop down and select the module’s type

g. Close the dialog by clicking “OK”

6. Close the main dialog by pressing “OK”.

No restart is required to activate the new settings.

## Handlers

This section focuses on developing HTTP handlers for IIS7 using the .NET Framework. We’ll look at when it is appropriate to develop an IIS7 handler rather than a module.

### Built-in Handlers

ASP.NET offers several default HTTP handlers:

\* Page Handler (.aspx) – Handles Web pages

\* User Control Handler (.ascx) – Handles Web user control pages

\* Web Service Handler (.asmx) – Handles Web service pages

\* Trace Handler (trace.axd) – Handles trace functionality

\* Assembly Resource Loader (WebResource.axd) – Handles embedded resources in assemblies

\* Script resource handler (ScriptResource.axd) – Handles the scripting support for AJAX enabled projects

\* Forbidden Handler (.config) – Denies access to files that contain confidential information

The IIS configuration defines the assignments. You will also find other assignments there. Extensions such as .xoml, .rem, .soap, and .svc relate to the capabilities provided by Windows Communication Foundation (WCF) and its predecessor, .NET remoting.

### Extending ASP.NET using Http Handlers

While modules are low level, and run against every inbound request to the ASP.NET application, HTTP Handlers focus more on a specific request mapping. This is usually a mapping of a file extension.

HTTP Handler implementations are very simple in their concept, but having access to the HttpContext object enables enormous versatility. Handlers are implemented through the IHttpHandler interface, or its asynchronous counterpart, IHttpAsyncHandler. The interface consists of a single method, ProcessRequest and a single property IsReusable. The asynchronous version has a pair of methods (BeginProcessRequest and EndProcessRequest) and the same IsReusable property. The vital ingredient is ProcessRequest, which receives an instance of the HttpContext object. This single method is responsible for handling a Web request from start to finish.

However, simple does not imply simplistic. As you may know, the regular page processing code and the web service processing code are implemented as handlers. Both are anything but simple. Their power originates from the HttpContext object, which has access to both the request information and the response data. This means that, like a web server, a handler can control the whole process on its own. Whatever you want to implement on the level of specific mapping is achievable using handlers.

#### Scenarios when to use HTTP Handlers

To better understand the power of handlers, let’s take a look at what others have implemented on top of IHttpHandler:

\* Creating dynamic images

\* Watermarking existing images

\* “Pretty printing” of the page’s source code

\* Generating dynamic content pulled from a database or external resource

\* Transforming content from other resources, such as XML into HTML

\* Extracting resources from assemblies on the fly

\* Redirecting to/from SSL

\* Implementing Pingback and Trackback capabilities, even if the site is not a blog

Additionally, you can implement handlers asynchronously. This vastly extends the potential usage scenarios. Because asynchronous calls are closely related to threading and performance, we looked at threading and how it could benefit from asynchronous programming in chapter 2. In this section, I will focus more on common usages of basic HTTP handler implementations.

#### Getting Started

For an HTTP Handler, all the action occurs through a single call to ProcessRequest. This can be as simple as:

public void ProcessRequest(HttpContext context)

{  
 context.Response.Write("Hello World");

}

Using the HttpContext object, you have access to the Request, Response, Session and Cache objects. You have all the key features of an ASP.NET request at your disposal, and you can use this to determine what users submitted and to return content back to the client. (Refer to chapter 1 to see why HttpContext plays such an important role in the request processing process.)

The key operation of the handler is to write output into the Response object—or, more specifically, the Response object’s OutputStream. This output is what is sent back to the client. Behind the scenes, the ISAPIWorkerRequest sends the OutputStream back to the ISAPI ecb.WriteClient method, which actually performs the IIS output generation. Again, refer to chapter 1 and chapter 2 to learn more about these steps.

### Building a Handler

Now, let’s build a simple handler. To do this, we define a class, which implements the System.Web.IHttpHandler interface.

Despite the prominence of the ProcessRequest method, you’ll also need to implement a property—IsReusable. This property, which returns a Boolean value, indicates whether the instance can be re-used for subsequent requests. In some cases, after processing a request, your handler may not be in a valid state for processing further requests—especially if data about the previous request was stored in member variables. This is because the ASP.NET runtime can handle many requests at the same time. As long as there are threads available in the thread pool, a new request will be processed even if another one is still running. Each thread requires a new instance of the handler, even if the handler is marked “is reusable”. When a request is completed, the current handler instance is retained in memory, and re-used for the next request. This can lead to odd behavior, depending on the workload and on the existence of other instances of the handler. Such problems can be unpredictable and difficult to simulate or recognize in a development environment.

For stable and reliable behavior, you might assume that setting the IsReusable property to false is the solution. After all, this would create a new instance of the object any time a request is about to be processed. However, depending on how “intensive” your code is, this can lead to higher memory consumption, more CPU workload, and less throughput. There is no strict rule about it, but re-using the instances is the preferred solution. Keep in mind that access to members is not exactly what you might expect. Therefore, it is advisable to avoid private members that hold data, if possible. If you still wish to use member variables, remember that they need to be thread safe. When replacing regular members with static methods, you’ll have to implement thread safe code. If any of these requirements cannot be fulfilled, you should set IsReusable to false. Otherwise, the implementation will look like:

public bool IsReusable  
{  
 get   
 {   
 return true;   
 }   
}

IsReusable should be a constant.

#### The Entry Point

The ProcessRequest method is the main entry point for the handler. Its role is to work off the request specified by the HttpRequest instance, from the provided HttpContext instance, and generate an appropriate response using the HttpResponse instance. The ProcessRequest method is invoked by the .NET runtime during the ExecuteRequestHandler request processing stage, assuming that the mapping is able to route the request to the specific handler. This is in contrast to modules, which receive all requests passing through the pipeline.

Finally, let’s implement the ProcessRequest method, so that our handler has something to do. To keep things simple, our handler will return the current time of the server. We can specify the time zone in the query string. Our goal is to request a URL, such as http://myserver/page.time, and obtain the current time of the server. In addition, we can get the universal coordinated time (UTC) by requesting http://myserver/page.time?utc=true. Here’s the implementation:

Listing 3-6. Simple Handler mapped to a new \*.time file extension

public class TimeHandler : IHttpHandler

{

#region IHttpHandler Members

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

DateTime dt;

string useUtc = context.Request.QueryString["utc"];

if (!String.IsNullOrEmpty(useUtc) && useUtc.Equals("true"))

{

dt = DateTime.UtcNow;

}

else

{

dt = DateTime.Now;

}

context.Response.Write( ⮰

String.Format("<html><body><h1>{0}</h1></body></html>", ⮰  
 dt.ToLongTimeString() ⮰

));

}

#endregion

}

As we assign this handler to a specific extension—time—we’ll only receive it when the client uses this specific URL. The response is simple and creates a small HTML page. You could even tailor the response to suit clients that are not browsers.

We use the HttpRequest.QueryString collection to retrieve a query string variable, and write the current time in response using the HttpResponse.Write method. I recommend using the OutputStream if other handlers are processing the request, or if you want to add to the existing response. In the example above, we write a complete response in the one handler and thus the Write method is appropriate.

Figure 3-5. Setting the mapping of a managed handler in IIS7

IIS7 does not require a restart or any other action in order to activate the handler. A request that uses the mapped extension should work immediately.

#### Example—Image Handler

A very common scenario for handlers is the manipulation of images. As with any other resource, a browser obtains an image by sending a GET request. Handling large numbers of images at multiple resolutions can be a challenge. Imagine a web shop with thousands of product images stored at one resolution. However, different image sizes are required throughout the site, from catalog thumbnails to large preview panes and icons in the shopping basket. Converting all these images into several different sizes could be expensive, even with a batch script. Images change frequently, and maintaining all current pictures in many different source sizes is an image management headache.

Writing code and creating images dynamically is a typical task for a handler. Attaching requests to an image could achieved by using a path filter like \*.png. The following example shows how easy it is to manipulate content and send it to a client.

Listing 3-7. Adding a watermark to an image using a handler.

namespace Apress.HttpHandler

{

public class ImageHandler : IHttpHandler

{

#region IHttpHandler Members

private const float FONTSIZE = 72F;

private const string FONT = "Verdana";

private const string TEXT = "Watermark";

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

// determine an image request

if ((Path.GetDirectoryName( ⮰

context.Request.Url.AbsolutePath)).EndsWith("Images"))

{

// load image and add watermark

Bitmap img = (Bitmap) Bitmap.FromFile( ⮰

context.Server.MapPath(context.Request.Url.AbsolutePath));

Graphics g = Graphics.FromImage(img);

Brush b = new SolidBrush(Color.Silver);

Font f = new Font(FONT, FONTSIZE);

SizeF stringMeasure = g.MeasureString(TEXT, f);

// calculate the string position to center output

float x, y;

x = img.Width / 2 - stringMeasure.Width / 2;

y = img.Height / 2 - stringMeasure.Height / 2;

g.DrawString(TEXT, new Font(FONT, FONTSIZE), b, x, y);

// output to the response stream

img.Save(context.Response.OutputStream, ImageFormat.Jpeg);

img.Dispose();

}

}

}

#endregion

}

}

The handler first checks that it is dealing with a file from a particular directory. While the file mapping forces the handler to run for every request for an image with the specified extension, this test restricts the special processing to images in the “Images” folder only. The handler loads the image from disk, resolving the local path via Server.MapPath. Then it applies the watermark to the image. The MeasureString method measures the string size in order to align it with the image so that the text appears centered horizontally and vertically. Streams simplify the output of the image. The Save method sends the output directly into an output stream in JPEG format. Disposing of the image is required, as frequent use of a handler on a system with high workload could prevent the garbage collector from freeing the memory often enough.

Figure 3-6. Adding a watermark to an image on the fly

Dynamic image manipulation is powerful and flexible, and there are numerous possibilities for using an image handler in your applications. The final step in creating such a handler is to configure it in *web.config*. Please refer to the section “Configuration and Deployment” to read more about the various settings.

#### Example—Read Dynamic CSS from Resource

In the next example, I describe a handler that loads stylesheets stored as embedded resources in an assembly. It replaces any call to a stylesheet by delivering a specific .css file. The definition in the *web.config* looks like this:

<add verb="GET" path="\*.css" type="Apress.HttpHandler.CssHandler" />

See the section “Configuration and Deployment” for more information about configuring a handler for the IIS7 integrated mode.

To obtain the style files, they must be marked as an Embedded Resource. There are several ways to handle data from sources other than the file system.

Figure 3-7. To handle files as embedded resources, use the file’s Property box

The code itself does not have any quirks. It processes whatever it encounters, and decides how to proceed from the information in HttpContext.

Listing 3-8. Dynamic handling of CSS from embedded resources

public class CssHandler : IHttpHandler

{

#region IHttpHandler Members

public bool IsReusable

{

get { return false; }

}

public void ProcessRequest(HttpContext context)

{

UnmanagedMemoryStream s;

if (context.Request.UserAgent.Contains("MSIE"))

{

s = (UnmanagedMemoryStream)

this.GetType().Assembly.GetManifestResourceStream(

"Apress.HttHandler.HandlerAssembly.Css.ie.css");

}

else

{

s = (UnmanagedMemoryStream)

this.GetType().Assembly.GetManifestResourceStream(

"Apress.HttHandler.HandlerAssembly.Css.ff.css");

}

s.Seek(0, SeekOrigin.Begin);

using (MemoryStream ms = new MemoryStream((int)s.Length))

{

byte[] buffer = new byte[s.Length];

s.Read(buffer, 0, buffer.Length);

ms.Write(buffer, 0, buffer.Length);

ms.WriteTo(context.Response.OutputStream);

}

}

#endregion

}

Firstly, let’s locate the current user agent in this code by using the property UserAgent of the HttpRequest object. If the UserAgent is “MSIE”, indicating an Internet Explorer browser, then the resource named “ie.css” is retrieved from the assembly. For any other UserAgents, “ff.css” is loaded instead. The stream is copied to a MemoryStream object. The MemoryStream is helpful as it is able to copy its own content to another stream by using the WriteTo method. In the case of a handler, this is the output stream provided by the HttpResponse object through the Response property.

This technique can be re-used. By looking for specific file extensions and for the “user-agent” header, you can block certain clients from reading the content of these resources or supply them with a replacement resource. Alternatively, reading resources from a database instead of an assembly gives more flexibility without adding code to the pages themselves.

#### Example—Handler That Does Not Create Content

Handlers usually create content—typically HTML or data that builds an image. However, it’s not imperative. Handlers are primarily invoked by the associated file extension. Imagine you define a custom extension named “.counter” and associate it with a handler. Your handler might execute some code, but ignore the Writer and not send anything back to the client. Another idea is to only write data to the client when you run the handler in debug mode and save bandwidth when running on a production server. Whether or not the output is required, you’ll still have to invoke the handler by issuing a GET or POST request to the server. Therefore, you’ll need an element that can force the browser to create such a request. There are only few ways to do this:

\* Image element

\* Form element

\* IFrame element

\* JavaScript code

I’m using an IFRAME element in the example below, because I want to show the content when in debug mode, and leave the content empty, otherwise. The definition of the IFRAME demonstrates how we invoke the handler:

Listing 3-9. Invoke a handler using a registered extension

<iframe src="my.counter" width="100" height="50" scrolling="no"></iframe>

Furthermore, the handler itself shows the processing within the ProcessRequest method.

Listing 3-10. Code of the handler

public class CounterHandler : IHttpHandler

{

#region IHttpHandler Members

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

string path = context.Request.UrlReferrer.LocalPath;

string file = context.Server.MapPath("Counter.xml");

XDocument cntDoc = XDocument.Load(file);

var cnt = (from e in cntDoc.Root.Elements("page") where ⮰

e.Attribute("path").Value.Equals(path) ⮰

select e).FirstOrDefault<XElement>();

if (Debugger.IsAttached)

{

// In debug mode read only and create output

// reading XML is thread safe

context.Response.Write(String.Format("Counter = {0}", ⮰

cnt.Attribute("count").Value));

}

else

{

// in production just store values

ReaderWriterLock rwl = new ReaderWriterLock();

rwl.AcquireWriterLock(TimeSpan.FromSeconds(2));

if (cnt == null)

{

// page does not exist yet

XElement newPage = new XElement("page",

new XAttribute("count", 1),

new XAttribute("path", path));

cntDoc.Element("Pages").Add(newPage);

}

else

{

// increase counter, set time stamp

int i = Int32.Parse(cnt.Attribute("count").Value);

cnt.Attribute("count").Value = (++i).ToString();

}

cntDoc.Save(file);

rwl.ReleaseWriterLock();

}

}

#endregion

}

The handler stores the number of page requests in a single XML document. Other developers can add the IFRAME code to the pages they want included in the counter total. Internally, the file access is shared read mode. The FileStream class used behind the scenes in XDocument.Load supports reading from different threads. However, write access requires an exclusive lock, blocking other threads. To achieve this, a ReaderWriterLock class monitors the threads and blocks other threads from accessing the file during write mode. This isn’t a very efficient method. In a real life scenario, consider replacing the XML access with a database operation. After obtaining access to the XML, LINQ to XML is used to either create elements for the first time or add to the existing counter when the page is requested again.

In the example, we want to distinguish between debug mode and production mode. You can ascertain debug mode by using the Debugger.IsAttached property. In debug mode, the content of the IFRAME is filled with the counter for the current page. The LINQ statement prepared at the beginning of the ProcessRequest method returns either null or the element containing the counter information.

The figure shows the output in debug mode. The counter does not increase for requests in debug mode. In production mode, the counter increases but no output is displayed.

Figure 3-8. Output of the handler in debug mode

In this example, I showed a handler that doesn’t create content with every use. The concepts behind this are, at a glance:

\* Defining how and where to invoke the handler

\* Remembering that concurrent threads access the code

\* Recognizing that creating output is not mandatory

#### Example—Using IHttpHandlerFactory to Perform URL Rewriting

In this example, we’ll introduce another interface. For more flexibility with creating handlers on the fly, the IHttpHandlerFactory interface is available. A closer look at the default PageHandlerFactory used to process .aspx pages shows that it is not derived from IhttpHandler, but from IHttpHandlerFactory.

You don’t have to create a factory, but it gives more control over the creation process. It can be useful in more complex scenarios. First, let’s look into the interface definition:

public interface IHttpHandlerFactory

{

IHttpHandler GetHandler(HttpContext context, ⮰

string requestType, ⮰

string url, ⮰

string pathTranslated);

void ReleaseHandler(IHttpHandler handler);

}

Note: If you decode this using Reflector, you may see another interface named IHttpHandlerFactory2. This provides another overload for the GetHandler method. We don’t need this at this stage as we’ll only be using the IHttpHandlerFactory.

Imagine you have a page with content such as:

<% = DateTime.Now.ToString("M") %>

Set the current culture by adding the culture ID to the URL. Typical URLs look like this:

\* http://localhost/Chapter03/UrlRewriteFactoryHandler/Default.aspx

\* http://localhost/Chapter03/UrlRewriteFactoryHandler/en-us/Default.aspx

\* http://localhost/Chapter03/UrlRewriteFactoryHandler/de-de/Default.aspx

\* http://localhost/Chapter03/UrlRewriteFactoryHandler/fr-fr/Default.aspx

Our goal is to find a way to extract the culture code (such as “de-de” or “en-us”) from the URL, set it as the current culture of the thread, and process the page without the culture code, as shown in the first URL.

Listing 3-11. Using a factory to rewrite a URL

public abstract class RewriteFactoryHandler : IHttpHandlerFactory

{

protected RewriteFactoryHandler()

: base()

{

}

IHttpHandler IHttpHandlerFactory.GetHandler(HttpContext context, ⮰

string requestType, ⮰

string url, ⮰

string pathTranslated)

{

Pair target = GetRemapInfo(context, requestType, url, pathTranslated);

string filename = context.Server.MapPath(target.First.ToString());

context.RewritePath(url, url, target.Second.ToString());

IHttpHandler appHandler = ⮰

PageParser.GetCompiledPageInstance(target.First.ToString(), ⮰

filename, context);

return appHandler;

}

void IHttpHandlerFactory.ReleaseHandler(IHttpHandler handler)

{

}

protected abstract Pair GetRemapInfo(HttpContext context, string requestType, ⮰

string url, string pathTranslated);

}

public class CultureRewriteHandler : RewriteFactoryHandler

{

protected override Pair GetRemapInfo(HttpContext context, ⮰

string requestType, ⮰

string url, ⮰

string pathTranslated)

{

string originalPath = HttpContext.Current.Request.Path;

string stemPath = "/";

string newPath = originalPath.Substring(originalPath.IndexOf(stemPath) + ⮰

stemPath.Length);

string[] segments = newPath.Split('/');

string queryString = HttpContext.Current.Request.Url.Query;

try

{

string languagePart = segments[0];

CultureInfo ci = new CultureInfo(languagePart);

System.Threading.Thread.CurrentThread.CurrentCulture = ci;

return new Pair("/" + stemPath + string.Join("/", segments, 1, ⮰

segments.Length - 1), queryString);

}

catch (NullReferenceException)

{

}

return new Pair(originalPath, queryString);

}

}

The definition in *web.config* or IIS settings is the same as for any other handler. This means that the ASP.NET engine will accept both IHttpHandler and IHttpHandlerFactory in order to obtain access to the handler object.

The code is simplified for clarity. You can extend the error handling by adding code to handle the stem path, even if it’s set to a value other than the root path. The core implementation is around the GetHandler method, which returns the used handler. If resources are blocked and need to be freed or disposed of, the required code will appear in the ReleaseHandler method. As shown in the example, this is not always necessary. Keep in mind that the memory consumption of the handler might be an issue if the server has a high workload, in which case releasing resources could help.

Figure 3-9 shows the behavior for several languages. URL rewriting is a flexible, search engine friendly method of modifying behavior.

Insert ASPEXTf0316.tif

Figure 3-9. Use URL rewriting to set the current culture of a Web application

This example demonstrated a very basic handler factory. In the abstract base class, the GetHandler method rewrites using HttpContext.RewritePath. The default page handler is subsequently retrieved and returned. This is required because otherwise, the processing of all .aspx pages is remapped to the new factory. Redefining the internal handler with your own one replaces the mapping, as only one handler can process a specific request. Remapping the default handler requires either a complete implementation of a handler with similar behavior, or creating the original handler and returns it using a factory. From the perspective of extensibility, the latter is the better option.

### Advanced Usage of Handlers

The standard handlers cover most, but not all, tasks. Http Handlers can go further. In this section, I’ll discuss advanced extensibility topics:

\* Accessing the session state

\* Dynamically dealing with handlers in the pipeline

#### Handlers and Session State

Handlers are low level programming constructs. They are critical for overall performance and if badly written or configured, could degrade the server’s throughput. While there are ways to deal with long-running threads in the handler code (as you saw in chapter 2), it’s preferable to write handlers that run as fast as possible. To maximize handler speed, Microsoft removed session information from the default handlers. The previous examples show useful tasks accomplished without needing session information.

If you do need to access session state information, it is available by implementing one of the following two interfaces:

\* IRequiredSessionState

\* IReadOnlySessionState

It’s possible to obtain session information with minimal performance loss. If you only require read access to the session data, the IReadOnlySessionState is ideal. IRequiredSessionState gives full access to all session data. When adding either interface to your class, you’ll notice that Visual Studio does not attempt to implement any method bodies. Both interfaces are simply marker interfaces which modify the internal processing within the base class. Your class declaration should look like this:

public class TimeHandler : IHttpHandler, IRequiredSessionState

The session information is now provided to the HttpContext object, and available through the context parameter of the entry method.

#### Accessing the Pipeline Using the Context

You can use these three properties of the HttpContext object to further modify the behavior of the handler or to retrieve more information about what’s taking place:

\* context.Handler

\* context.PreviousHandler

\* context.RemapHandler

Using the Handler property, you have access to the current handler employed in the current context. As the context is available as a static property, it’s easy to access the handler in classes defined elsewhere. This also applies to the PreviousHandler property, a property which is set when the handler is remapped. Remapping a handler might occur in complex scenarios where a default handler processes all requests, but remaps to another handler under certain circumstances. Listing 3-12 demonstrates this technique:

Listing 3-12. Remapping to another handler

public class RemapHandler : IHttpHandler

{

#region IHttpHandler Members

public bool IsReusable

{

get { return true; }

}

public void ProcessRequest(HttpContext context)

{

IHttpHandler remapHandler = null;

// determine an image request and handle with private handler

if ((Path.GetExtension(context.Request.Url.AbsolutePath)).Equals(".png"))

{

remapHandler = new ImageHandler();

}

else

{

// process any other request with default handler

string virtualPath = context.Request.Url.AbsolutePath;

string filename = HttpContext.Current.Request.Path;

remapHandler = PageParser.GetCompiledPageInstance(virtualPath, ⮰

filename, context);

}

context.RemapHandler(remapHandler);

}

#endregion

}

In this code, the handler searches for .png extensions and assigns private handlers to process them. Otherwise, we use the standard page processing; no other custom code is involved. Simply call context.RemapHandler(remapHandler) to assign to a different handler. All internal processing is redirected to the new handler from the beginning of the pipeline. As there’s no additional overhead, there are no performance issues with the handler remapping.

### Configuration and Deployment

To create a handler, you chose “Class Library” as the project template, removed the default class created by the template and added an object of type “ASP.NET Handler”. Now that you have implemented the handler, you can compile it into an assembly loaded by ASP.NET at runtime. As long as the handler remains in the web application, this is simple. No special action is required. You’ll probably want to implement several handlers and keep them in different assemblies for easy reuse. Simply reference the assemblies in your web project.

Figure 3-10. Add a handler to project

#### Configuring Default Web Server and Development Environment

To test the handler, you’ll need to configure the settings in *web.config*. Place the appropriate settings in the <system.web> section:

<httpHandlers>

<add verb="GET" path="\*.png" type="Apress.HttpHandler.ImageHandler" />

</httpHandlers>

The settings for the development environment also apply to IIS5, IIS6, and IIS7 in classic mode. There are several advantages to running the IIS7 integrated pipeline, however, which involves different settings:

Table 3-15. Options of the httpHandler settings

Attribute Typical Values Description

verb GET, POST The handler responds to the HTTP verbs only

path full path or wildcards The path that defines the requests the handlers responds to

type class, assembly Type information of the handler’s definition

#### Configuring IIS7 Settings

In the main (web) project, add a reference to this project. If the namespace of the external project is Apress.HttHandler.ImageHandler, the following addition to *web.config* will be required:

<system.webServer>

<handlers>

<add name="ImageHandler" path="\*.png" verb="GET"

type="Apress.HttpHandler.ImageHandler" resourceType="File"

requireAccess="Read" preCondition="integratedMode" />

</handlers>

</system.webServer>

Compile both the project containing the handler and the web project. Add the mapping in Internet Information Services Manager, as shown before. The mapping will now function perfectly for both the development environment and direct usage from the local IIS7.

Table 3-16. Handler settings for IIS7 integrated mode

Attribute Typical Values Description

verb GET, POST The handler responds to the HTTP verbs only. Other verbs are DEBUG and HEAD

path full path or wildcards The path that defines the requests the handlers respond to

resourceType File Expect that the file exists

scriptProcessor a path Path to the engine (DLL) that handles the request

requireAccess Script, Execute, None, Read Required settings for resource access

preCondition see below Conditions that must be fulfilled to activate the handler. If the request fails, an HTTP error 412 “precondition failed” is send to the client

type class, assembly Type information of the handler’s definition

Typical values for preCondition are these:

\* bitness32, bitness64: Activate 32-bit or 64-bit mode respectively

\* runtimeVersion1.1, runtimeVersion2.0: The required runtime on the server

\* classicMode, integratedMode: Mode that IIS7 is running in

\* managedHandler: Requires the handler to be written in managed code

#### Configure via IIS Management Console

The settings in *web.config* required by IIS7 integrated mode can be altered via the IIS Management Console. The settings correspond directly. Altering *web.config* will result in an immediate change to the Management Console settings, and vice versa. To configure using the IIS Management Console:

1. Open Internet Information Service Manager.

2. Open the web you want to change.

3. In the IIS section, double click on the Handler Assignments.

4. Click on “Add Managed Handler” in the task list to the right.

5. Enter these values in the dialog:

a. The mapping path: \*.time

b. Choose the handler from the drop down list. The handler will appear in the list, as long as you’re in the right web and the project compiles.

c. Give the handler an appropriate name

d. Click on Restrictions…

e. Open the Verb tab

f. Click on one of the following and enter the value “GET”

g. Leave the dialog by clicking “OK”

6. Leave the main dialog by clicking “OK”.

#### Configure Using Generic Handlers

The final option is not a configuration option, but a way of invoking handlers without configuring them in *web.config*. By default, ASP.NET defines handlers using the extension .ashx. Therefore, placing the code for a handler in a file using the declarative form below is sufficient to get it working. There is no further need for *web.config*. An advantage of this is that you won’t need to distinguish between the settings for IIS7 integrated mode and other Web servers, nor will you need to maintain settings in the *web.config* regarding handlers.

Creating such a handler involves two steps:

\* For each handler required, create a file with the extension .ashx

\* Add the following declaration at the top of the file:

<%@ WebHandler Language="C#" Class="Apress.HttpHandler.MyHandler" %>

The WebHandler directive is similar to the Page directive. Only a few options are available, however:

Table 3-17. Attributes supported for WebHandler directive

Attribute Available Values Usage

Language C#, VB Language of the code section

Class Name Name of the class defined in code, including the namespace

CodeBehind Name Name of the file containing the code

CompilerOptions Options for on-the-fly compilation

Debug True, False Compile in debug or release mode. In debug mode the symbol file (.pdb) is created

WarningLevel 0 to 4 Warning level

Description A description for documentation purposes only. The page parser does not recognize this attribute.

Whether you use the configuration file or the .ashx extension is a matter of preference. However, there are some basic guidelines for selecting the best option. Using the .ashx file within an application is better for small projects or handlers that have simple, but specific, tasks. If you plan to reuse the handler several times, or in several projects on the server, you should separate it into its own assembly and register it in the Global Assembly Cache (GAC). Storing handlers in assemblies with signing and deployment capabilities is for larger projects involving handler reuse.

## Testing and Debugging Modules and Handlers

Having deployed and configured the module or handler, debugging may be required. In Visual Studio, the normal Debug mode works well for debugging handlers. You’re probably already familiar with the debugging capabilities in Visual Studio. Setting breakpoints and viewing variable’s values is just as simple for handlers as for any other type of .NET solution.

### Debug using IIS

You may occasionally experience trouble with your application when running on IIS. There are no breakpoints, and adding logging and tracing capabilities can be tedious. In large projects there are often coding guidelines that require you to add tracing code and to log pertinent messages (such as exceptions). In smaller projects, it might not be appropriate to write more code for logging and tracing. Setting a simple breakpoint and examining a value or condition during a request would be nice.

It can be done. To do so, you can attach a debugger to a running application. In the case of an ASP.NET application running on IIS7, this is the worker process—w3wp.exe. Let’s consider the case where you have not published your project, but simply compiled on the fly, and your sources—the .cs files—are still available beside the .aspx files. (I’ll discuss techniques for attaching to a precompiled project later.) Here, the Visual Studio debugger attaches automatically to the current process when you hit F5, and start a debug session. This current process is the internal Web server included in Visual Studio. You can achieve the same thing simply by attaching the debugger to the worker process. If the worker process is not running, force it by requesting the first page and invoking the modules or handlers configured for your application. It doesn’t matter whether it is running properly or not.

Here is a brief summary of the pre-conditions so far:

\* IIS is configured to run the Web directly from project files

\* Visual Studio is running and has the project loaded (however, there is no debug session so far)

\* The Worker process is up and running

Tip: You can force the worker process to start by invoking a first request to the application. To check whether it is available, open Task Manager, switch to the Processes tab, check the box (Windows Server 2008) or click the button (Windows Vista) “Show processes from all users”, and search for w3wp.exe in the list.

Figure 3-11. Use the Task Manager to check for the worker process

Now open Visual Studio and attach the debugger to the worker process:

1. Open *Debug > Attach to Process…*

2. In the subsequent dialog, check these settings:

a) *Transport*: default

b) *Qualifier*: The name of the server or workstation

c) In the *Attached to* section, you should at least have the option Managed Code selected. Use the Select button to change settings.

3. In the list of Available Processes, look for the worker process. If it’s not there, tick the checkbox *Show processes from all users*. This is same option as in the Task Manager.

4. Use the Refresh button to reload the list of processes during the session without closing the dialog

5. Mark available worker processes and click *Attach*.

Figure 3-12. Attaching the debugger to the worker process

Depending on your server conditions, you may find that several worker processes appear in the list. If you’re not sure which process is the one handling the current request, you can attach to all of them. Alternatively, use Task Manager to kill all the worker processes, issue a new request to your application, and refresh the list. If no one else is using the server, one worker process will appear. Of course, the usage of worker processes—as explained in chapter 2—depends on the threads required. Under rare circumstances, the application needs more power and splits the requests into multiple worker processes. However, attaching several instances of the worker process to the same debugger session is quite easy.

With Visual Studio running in debug mode, as it is when you hit the F5 key, you can set breakpoints within the module or handler code and invoke a request to hit the breakpoint. You can even watch the debug and trace information in the output window.

#### Problems Debugging the Worker Process

Sometimes the behavior of the debugger does not match your expectations. Quite frequently the breakpoints appear inactive, or they can’t be ‘hit’, as Visual Studio calls it.

Figure 3-13. The breakpoint will not currently be hit

Usually this is because the page has not yet been loaded. Because we decided to let ASP.NET compile pages on the fly, the current page might not be available yet and therefore the symbols are not built. To check this, use the *Modules* dialog in Visual Studio.

Figure 3-14. Use the Modules window to retrieve information about loaded symbols

In the Modules window, look for the assembly you built for your module or handler. In the context menu of each entry, use the *Symbol load information* item to retrieve more information. Either you’ll obtain the full path, or the symbol file (.pdb) is loaded from the list of paths Visual Studio has tried so far.

Tip: To debug parts of the operating system or .NET framework, use the Modules dialog to attach foreign pdb files. Additionally, it can be a good idea to set up a symbol server in your company in order to have common symbol files ready, or, alternatively, attach to Microsoft’s public symbol server.

Figure 3-15. Use the symbols settings in Visual Studio to optimize access to public PDBs

Once everything is functioning normally and the attached symbols are available, the breakpoints should function as expected and be “hit” when the code execution reaches them.

Figure 3-16. Everything is ready to go if the breakpoint is active

### Set up Tracing for Handlers

Aspx pages have a powerful and popular tracing feature. When you set the following page directive, the page’s content will be replaced by a complete analysis of the request:

<%@ Page Trace="true" …

However, the @WebHandler directive does not support this. If you write your module or handler in a separate assembly, using plain code, there is no directive at all.

An example of a handler invoked by file extension is the *Trace.axd* file used for debugging. In order to invoke the *Trace.axd* handler, configure the website for tracing by adding a trace section to *web.config* as this:

<configuration>

<system.web>

<trace enabled="true"/>

</system.web>

</configuration>

Call the *trace.axd* file from the root of the website: for example, *http://localhost/trace.axd*.

## Summary

In this chapter, you learned how to extend the pipeline by creating your own modules and handlers. A module provides low level processing and is invoked twice: when the request bubbles up the pipeline and again after all internal processing by the designated handler is complete. Using a handler, you can add your own processing code either assigned to your own file extension or using the generic .ashx extension. Handlers process requests and create the output sent to the client.

Asynchronous handlers help to process long-running requests—such as database queries or Web services—without filling the thread pool with threads. This technique improves the overall performance of a Web application and leads to a consistently smooth user experience.