URL Rewriting in ASP.NET

Scott Mitchell
4GuysFromRolla.com

March 2004

Applies to:
Microsoft® ASP.NET

Page Options
Average rating:
8 out of 9
Rate this page
Print this page
E-mail this page

Summary: Examines how to perform dynamic URL rewriting with Microsoft ASP.NET. URL rewriting is the process of intercepting an incoming Web request and automatically redirecting it to a different URL. Discusses the various techniques for implementing URL rewriting, and examines real-world scenarios of URL rewriting. (31 printed pages)

Download the source code for this article.

Contents

Introduction

Take a moment to look at some of the URLs on your website. Do you find URLs like http://yoursite.com/info/dispEmployeeInfo.aspx?EmpID=459-099&type=summary? Or maybe you have a bunch of Web pages that were moved from one directory or website to another, resulting in broken links for visitors who have bookmarked the old URLs. In this article we'll look at using *URL rewriting* to shorten those ugly URLs to meaningful, memorable ones, by replacing http://yoursite.com/info/dispEmployeeInfo.aspx?EmpID=459-099&type=summary with something like http://yoursite.com/people/sales/chuck.smith. We'll also see how URL rewriting can be used to create an intelligent 404 error.

URL rewriting is the process of intercepting an incoming Web request and redirecting the request to a different resource. When performing URL rewriting, typically the URL being requested is checked and, based on its value, the request is redirected to a different URL. For example, in the case where a website restructuring caused all of the Web pages in the /people/ directory to be moved to a /info/employees/ directory, you would want to use URL rewriting to check if a Web request was intended for a file in the /people/ directory. If the request was for a file in the /people/ directory, you'd want to automatically redirect the request to the same file, but in the /info/employees/ directory instead.

With classic ASP, the only way to utilize URL rewriting was to write an ISAPI filter or to buy a third-party product that offered URL rewriting capabilities. With Microsoft® ASP.NET, however, you can easily create your own URL rewriting software in a number of ways. In this article we'll examine the techniques available to ASP.NET developers for implementing URL rewriting, and then turn to some real-world uses of URL rewriting. Before we delve into the technological specifics of URL rewriting, let's first take a look at some everyday scenarios where URL rewriting can be employed.

Common Uses of URL Rewriting

Creating data-driven ASP.NET websites often results in a single Web page that displays a subset of the database's data based on querystring parameters. For example, in designing an e-commerce site, one of your tasks would be to allow users to browse through the products for sale. To facilitate this, you might create a page called displayCategory.aspx that would display the products for a given category. The category's products to view would be specified by a querystring parameter.

That is, if the user wanted to browse the Widgets for sale, and all Widgets had a had a CategoryID of 5, the user would visit: http://yousite.com/displayCategory.aspx?CategoryID=5.

There are two downsides to creating a website with such URLs. First, from the end user's perspective, the URL http://yousite.com/displayCategory.aspx?CategoryID=5 is a mess. Usability expert Jakob Neilsen recommends that URLs be chosen so that they:

- Are short.
- Are easy to type.
- Visualize the site structure.
- "Hackable," allowing the user to navigate through the site by hacking off parts of the URL.

I would add to that list that URLs should also be easy to remember. The URL http://yousite.com/displayCategory.aspx?CategoryID=5 meets none of Neilsen's criteria, nor is it easy to remember. Asking users to type in querystring values makes a URL hard to type and makes the URL "hackable" only by experienced Web developers who have an understanding of the purpose of querystring parameters and their name/value pair structure.

A better approach is to allow for a sensible, memorable URL, such as http://yoursite.com/products/Widgets. By just looking at the URL you can infer what will be displayed—information about Widgets. The URL is easy to remember and share, too. I can tell my colleague, "Check out yoursite.com/products/Widgets," and she'll likely be able to bring up the page without needing to ask me again what the URL was. (Try doing that with, say, an Amazon.com page!) The URL also appears, and should behave, "hackable." That is, if the user hacks of the end of the URL, and types in http://yoursite.com/products, they should see a listing of *all* products, or at least a listing of all categories of products they can view.

Note For a prime example of a "hackable" URL, consider the URLs generated by many blog engines. To view the posts for January 28, 2004, one visits a URL like http://someblog.com/2004/01/28. If the URL is hacked down to http://someblog.com/2004/01, the user will see all posts for January 2004. Cutting it down further to http://someblog.com/2004 will display all posts for the year 2004.

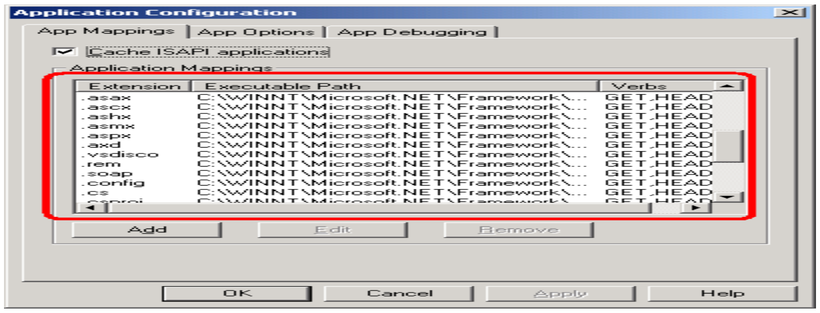
In addition to simplifying URLs, URL rewriting is also often used to handle website restructuring that would otherwise result in numerous broken links and outdated bookmarks.

What Happens When a Request Reaches IIS

Before we examine exactly how to implement URL rewriting, it's important that we have an understanding of how incoming requests are handled by Microsoft® Internet Information Services (IIS). When a request arrives at an IIS Web server, IIS examines the requested file's extension to determine how handle the request. Requests can be handled natively by IIS—as are HTML pages, images, and other static content—or IIS can route the request to an ISAPI extension. (An ISAPI extension is an unmanaged, compiled class that handles an incoming Web request. Its task is to generate the content for the requested resource.)

For example, if a request comes in for a Web page named Info.asp, IIS will route the message to the asp.dll ISAPI extension. This ISAPI extension will then load the requested ASP page, execute it, and return its rendered HTML to IIS, which will then send it back to the requesting client. For ASP.NET pages, IIS routes the message to the aspnet_isapi.dll ISAPI extension. The aspnet_isapi.dll ISAPI extension then hands off processing to the managed ASP.NET worker process, which processes the request, returning the ASP.NET Web page's rendered HTML.

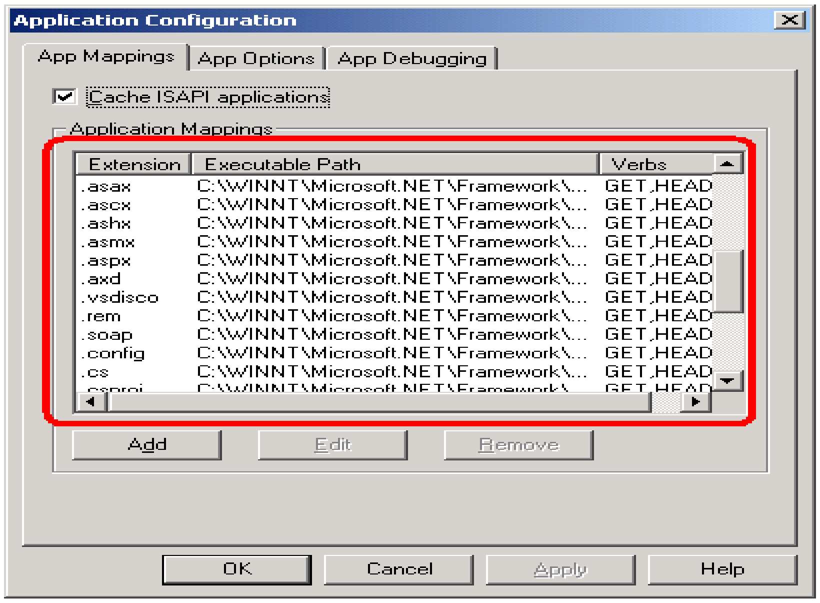
You can customize IIS to specify what extensions are mapped to what ISAPI extensions. Figure 1 shows the Application Configuration dialog box from the Internet Information Services Administrative Tool. Note that the ASP.NET-related extensions—.aspx, .ascx, .config, .asmx, .rem, .cs, .vb, and others—are all mapped to the aspnet_isapi.dll ISAPI extension.



[✓] Cache ISAPI applications

Application Mappings

Extension	Executable Path	Verbs
.asax	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.ascx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.ashx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.asmx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.aspx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.axd	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.vsdisco	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.rem	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.soap	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.config	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.cs	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.csproj	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD



[✓] Cache ISAPI applications

Application Mappings

Extension	Executable Path	Verbs
.asax	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.ascx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.ashx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.asmx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.aspx	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.axd	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.vsdisco	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.rem	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.soap	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.config	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.cs	C:\WINNT\Microsoft.NET\Framework\...	GET,HEAD
.csproj	C:\WINNT\Microsoft.NET\Framework\	GET,HEAD

Figure 1. Configured mappings for file extensions

A thorough discussion of how IIS manages incoming requests is a bit beyond the scope of this article. A great, in-depth discussion, though, can be found in Michele Leroux Bustamante's article Inside IIS and ASP.NET. It's important to understand that the ASP.NET engine gets its hands only on incoming Web requests whose extensions are explicitly mapped to the aspnet_isapi.dll in IIS.

Examining Requests with ISAPI Filters

In addition to mapping the incoming Web request's file extension to the appropriate ISAPI extension, IIS also performs a number of other tasks. For example, IIS attempts to authenticate the user making the request and determine if the authenticated user has authorization to access the requested file. During the lifetime of handling a request, IIS passes through several states. At each state, IIS raises an event that can be programmatically handled using ISAPI filters.

Like ISAPI extensions, ISAPI filters are blocks of unmanaged code installed on the Web server. ISAPI extensions are designed to generate the response for a request to a particular file type. ISAPI filters, on the other hand, contain code to respond to events raised by IIS. ISAPI filters can intercept and even modify the incoming and outgoing data. ISAPI filters have numerous applications, including:

- Authentication and authorization.
- Logging and monitoring.
- HTTP compression.
- URL rewriting.

While ISAPI filters can be used to perform URL rewriting, this article examines implementing URL rewriting using ASP.NET. However, we will discuss the tradeoffs between implementing URL rewriting as an ISAPI filter versus using techniques available in ASP.NET.

What Happens When a Request Enters the ASP.NET Engine

Prior to ASP.NET, URL rewriting on IIS Web servers needed to be implemented using an ISAPI filter. URL rewriting is possible with ASP.NET because the ASP.NET engine is strikingly similar to IIS. The similarities arise because the ASP.NET engine:

1. Raises events as it processes a request.
2. Allows an arbitrary number of *HTTP modules* handle the events that are raised, akin to IIS's ISAPI filters.
3. Delegates rendering the requested resource to an *HTTP handler*, which is akin to IIS's ISAPI extensions.

Like IIS, during the lifetime of a request the ASP.NET engine fires events signaling its change from one state of processing to another. The **BeginRequest** event, for example, is fired when the ASP.NET engine first responds to a request. The **AuthenticateRequest** event fires next, which occurs when the identity of the user has been established. (There are numerous other events—**AuthorizeRequest**, **ResolveRequestCache**, and **EndRequest**, among others. These events are events of the **System.Web.HttpApplication** class; for more information consult the HttpApplication Class Overview technical documentation.)

As we discussed in the previous section, ISAPI filters can be created to respond to the events raised by IIS. In a similar vein, ASP.NET provides *HTTP modules* that can respond to the events raised by the ASP.NET engine. An ASP.NET Web application can be configured to have multiple HTTP modules. For each request processed by the ASP.NET engine, each configured HTTP module is initialized and allowed to wire up event handlers to the events raised during the processing of the request. Realize that there are a number of built-in HTTP modules utilized on each an every request. One of the built-in HTTP modules is the **FormsAuthenticationModule**, which first checks to see if forms authentication is being used and, if so, whether the user is authenticated or not. If not, the user is automatically redirected to the specified logon page.

Recall that with IIS, an incoming request is eventually directed to an ISAPI extension, whose job it is to return the data for the particular request. For example, when a request for a classic ASP Web page arrives, IIS hands off the request to the asp.dll ISAPI extension, whose task it is to return the HTML markup for the requested ASP page. The ASP.NET engine utilizes a similar approach. After initializing the HTTP modules, the ASP.NET engine's next task is to determine what *HTTP handler* should process the request.

All requests that pass through the ASP.NET engine eventually arrive at an HTTP handler or an HTTP handler factory (an HTTP handler factory simply returns an instance of an HTTP handler that is then used to process the request). The final HTTP handler renders the requested resource, returning the response. This response is sent back to IIS, which then returns it to the user that made the request.

ASP.NET includes a number of built-in HTTP handlers. The **PageHandlerFactory**, for example, is used to render ASP.NET Web pages. The **WebServiceHandlerFactory** is used to render the response SOAP envelopes for ASP.NET Web services. The **TraceHandler** renders the HTML markup for requests to **trace.axd**.

Figure 2 illustrates how a request for an ASP.NET resource is handled. First, IIS receives the request and dispatches it to aspnet_isapi.dll. Next, the ASP.NET engine initializes the configured HTTP modules. Finally, the proper HTTP handler is invoked and the requested resource is rendered, returning the generated markup back to IIS and back to the requesting client.

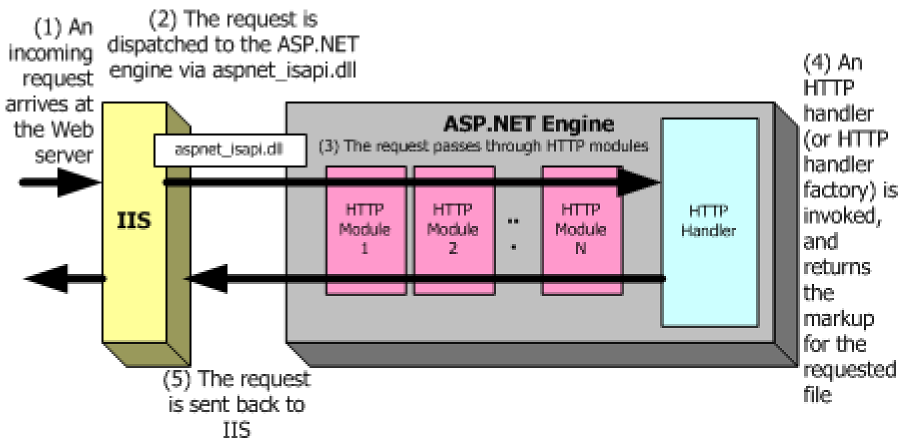

(1) An incoming request arrives at the Web server
(2) The request is dispatched to the ASP.NET engine via aspnet_isapi.dll
(4) An HTTP handler (or HTTP handler factory) is invoked, and returns the markup for the requested file
ASP.NET Engine
aspnet_isapi.dll
(3) The request passes through HTTP modules
IIS
HTTP Module 1
HTTP Module 2
HTTP Module N
HTTP Handler
(5) The request is sent back to IIS

Figure 2. Request processing by IIS and ASP.NET

Creating and Registering Custom HTTP Modules and HTTP Handlers

Creating custom HTTP modules and HTTP handlers are relatively simple tasks, which involve created a managed class that implements the correct interface. HTTP modules must implement the **System.Web.IHttpModule** interface, while HTTP handlers and HTTP handler factories must implement the **System.Web.IHttpHandler** interface and **System.Web.IHttpHandlerFactory** interface, respectively. The specifics of creating HTTP handlers and HTTP modules is beyond the scope of this article. For a good background, read Mansoor Ahmed Siddiqui's article, HTTP Handlers and HTTP Modules in ASP.NET.

Once a custom HTTP module or HTTP handler has been created, it must be registered with the Web application. Registering HTTP modules and HTTP handlers for an entire Web server requires only a simple addition to the machine.config file; registering an HTTP module or HTTP handler for a specific Web application involves adding a few lines of XML to the application's Web.config file.

Specifically, to add an HTTP module to a Web application, add the following lines in the Web.config's configuration/system.web section:

```
<httpModules>
   <add type="type" name="name" />
</httpModules>
```

The *type* value provides the assembly and class name of the HTTP module, whereas the *name* value provides a friendly name by which the HTTP module can be referred to in the Global.asax file.

HTTP handlers and HTTP handler factories are configured by the <httpHandlers> tag in the Web.config's configuration/system.web section, like so:

```
<httpHandlers>
   <add verb="verb" path="path" type="type" />
</httpHandlers>
```

Recall that for each incoming request, the ASP.NET engine determines what HTTP handler should be used to render the request. This decision is made based on the incoming requests verb and path. The verb specifies what type of HTTP request was made—GET or POST—whereas the path specifies the location and filename of the file requested. So, if we wanted to have an HTTP handler handle all requests—either GET or POST—for files with the .scott extension, we'd add the following to the Web.config file:

```
<httpHandlers>
   <add verb="*" path="*.scott" type="type" />
</httpHandlers>
```

where *type* was the type of our HTTP handler.

> **Note** When registering HTTP handlers, it is important to ensure that the extensions used by the HTTP handler are mapped in IIS to the ASP.NET engine. That is, in our .scott example, if the .scott extension is not mapped in IIS to the aspnet_isapi.dll ISAPI extension, a request for the file foo.scott will result in IIS attempting to return the contents of the file foo.scott. In order for the HTTP handler to process this request, the .scott extension must be mapped to the ASP.NET engine. The ASP.NET engine, then, will route the request correctly to the appropriate HTTP handler.

For more information on registering HTTP modules and HTTP handlers, be sure to consult the <httpModules> element documentation along with the <httpHandlers> element documentation.

Implementing URL Rewriting

URL rewriting can be implemented either with ISAPI filters at the IIS Web server level, or with either HTTP modules or HTTP handlers at the ASP.NET level. This article focuses on implementing URL rewriting with ASP.NET, so we won't be delving into the specifics of implementing URL rewriting with ISAPI filters. There are, however, numerous third-party ISAPI filters available for URL rewriting, such as:

- ISAPI Rewrite
- IIS Rewrite
- PageXChanger
- And many others!

Implementing URL rewriting at the ASP.NET level is possible through the **System.Web.HttpContext** class's **RewritePath()** method. The **HttpContext** class contains HTTP-specific information about a specific HTTP request. With each request received by the ASP.NET engine, an **HttpContext** instance is created for that request. This class has properties like: **Request** and **Response**, which provide access to the incoming request and outgoing response; **Application** and **Session**, which provide access to application and session variables; **User**, which provides information about the authenticated user; and other related properties.

With the Microsoft® .NET Framework Version 1.0, the **RewritePath()** method accepts a single string, the new path to use. Internally, the **HttpContext** class's **RewritePath(string)** method updates the **Request** object's **Path** and **QueryString** properties. In addition to **RewritePath(string)**, the .NET Framework Version 1.1 includes another form of the **RewritePath()** method, one that accepts three string input parameters. This alternate overloaded form not only sets the **Request** object's **Path** and **QueryString** properties, but also sets internal member variables that are used to compute the

Request object's values for its **PhysicalPath**, **PathInfo**, and **FilePath** properties.

To implement URL rewriting in ASP.NET, then, we need to create an HTTP module or HTTP handler that:

1. Checks the requested path to determine if the URL needs to be rewritten.
2. Rewrites the path, if needed, by calling the **RewritePath()** method.

For example, imagine that our website had information each employee, accessible through /info/employee.aspx?empID=employeeID. To make the URLs more "hackable," we might decide to have employee pages accessible by: /people/EmployeeName.aspx. Here is a case where we'd want to use URL rewriting. That is, when the page /people/ScottMitchell.aspx was requested, we'd want to rewrite the URL so that the page /info/employee.aspx?empID=1001 was used instead.

URL Rewriting with HTTP Modules

When performing URL rewriting at the ASP.NET level you can use either an HTTP module or an HTTP handler to perform the rewriting. When using an HTTP module, you must decide at what point during the request's lifecycle to check to see if the URL needs to be rewritten. At first glance, this may seem to be an arbitrary choice, but the decision can impact your application in both significant and subtle ways. The choice of where to perform the rewrite matters because the built-in ASP.NET HTTP modules use the **Request** object's properties to perform their duties. (Recall that rewriting the path alters the **Request** object's property values.) These germane built-in HTTP modules and the events they tie into are listed below:

HTTP Module	Event	Description
FormsAuthenticationModule	**AuthenticateRequest**	Determines if the user is authenticated using forms authentication. If not, the user is automatically redirected to the specified logon page.
FileAuthorizationMoudle	**AuthorizeRequest**	When using Windows authentication, this HTTP module checks to ensure that the Microsoft® Windows® account has adequate rights for the resource requested.
UrlAuthorizationModule	**AuthorizeRequest**	Checks to make sure the requestor can access the specified URL. URL authorization is specified through the <authorization> and <location> elements in the Web.config file.

Recall that the **BeginRequest** event fires before **AuthenticateRequest**, which fires before **AuthorizeRequest**.

One safe place that URL rewriting can be performed is in the **BeginRequest** event. That means that if the URL needs to be rewritten, it will have done so by the time any of the built-in HTTP modules run. The downside to this approach arises when using forms authentication. If you've used forms authentication before, you know that when the user visits a restricted resource, they are automatically redirected to a specified login page. After successfully logging in, the user is sent back to the page they attempted to access in the first place.

If URL rewriting is performed in the **BeginRequest** or **AuthenticateRequest** events, the login page will, when submitted, redirect the user to the rewritten page. That is, imagine that a user types into their browser window, /people/ScottMitchell.aspx, which is rewritten to /info/employee.aspx?empID=1001. If the Web application is configured to use forms authentication, when the user first visits /people/ScottMitchell.aspx, first the URL will be rewritten to /info/employee.aspx?empID=1001; next, the **FormsAuthenticationModule** will run, redirecting the user to the login page, if needed. The URL the user will be sent to upon successfully logging in, however, will be /info/employee.aspx?empID=1001, since that was the URL of the request when the **FormsAuthenticationModule** ran.

Similarly, when performing rewriting in the **BeginRequest** or **AuthenticateRequest** events, the **UrlAuthorizationModule** sees the rewritten URL. That means that if you use <location> elements in your Web.config file to specify authorization for specific URLs, you will have to refer to the rewritten URL.

To fix these subtleties, you might decide to perform the URL rewriting in the **AuthorizeRequest** event. While this approach fixes the URL authorization and forms authentication anomalies, it introduces a new wrinkle: file authorization no longer works. When using Windows authentication, the **FileAuthorizationModule** checks to make sure that the authenticated user has the appropriate access rights to access the specific ASP.NET page.

Imagine if a set of users does not have Windows-level file access to C:\Inetput\wwwroot\info\employee.aspx; if such users attempt to visit /info/employee.aspx?empID=1001, then they will get an authorization error. However, if we move the URL rewriting to the **AuthenticateRequest** event, when the **FileAuthorizationModule** checks the security settings, it still thinks the file being requested is /people/ScottMitchell.aspx, since the URL has yet to be rewritten. Therefore, the file authorization check will pass, allowing this user to view the content of the rewritten URL, /info/employee.aspx?empID=1001.

So, when should URL rewriting be performed in an HTTP module? It depends on what type of authentication you're employing. If you're not using any authentication, then it doesn't matter if URL rewriting happens in **BeginRequest**, **AuthenticateRequest**, or **AuthorizeRequest**. If you are using forms authentication and are not using Windows authentication, place the URL rewriting in the **AuthorizeRequest** event handler. Finally, if you are using Windows authentication, schedule the URL rewriting during the **BeginRequest** or **AuthenticateRequest** events.

URL Rewriting in HTTP Handlers

URL rewriting can also be performed by an HTTP handler or HTTP handler factory. Recall that an HTTP handler is a class responsible for generating the content for a specific type of request; an HTTP handler factory is a class responsible for returning an instance of an HTTP handler that can generate the content for a specific type of request.

In this article we'll look at creating a URL rewriting HTTP handler factory for ASP.NET Web pages. HTTP handler factories must implement the **IHttpHandlerFactory** interface, which includes a **GetHandler()** method. After initializing the appropriate HTTP modules, the ASP.NET engine determines what HTTP handler or HTTP handler factory to invoke for the given request. If an HTTP handler factory is to be invoked, the ASP.NET engine calls that HTTP handler factory's **GetHandler()** method passing in the **HttpContext** for the Web request, along with some other information. The HTTP handler factory, then, must return an object that implements **IHttpHandler** that can handle the request.

To perform URL rewriting through an HTTP handler, we can create an HTTP handler factory whose **GetHandler()** method checks the requested path to determine if it needs to be rewritten. If it does, it can call the passed-in **HttpContext** object's **RewritePath()** method, as discussed earlier. Finally, the HTTP handler factory can return the HTTP handler returned by the **System.Web.UI.PageParser** class's **GetCompiledPageInstance()** method. (This is the same technique by which the built-in ASP.NET Web page HTTP handler factory, **PageHandlerFactory**, works.)

Since all of the HTTP modules will have been initialized prior to the custom HTTP handler factory being instantiated, using an HTTP handler factory presents the same challenges when placing the URL rewriting in the latter stages of the events—namely, file authorization will not work. So, if you rely on Windows authentication and file authorization, you will want to use the HTTP module approach for URL rewriting.

Over the next section we'll look at building a reusable URL rewriting engine. Following our examination of the URL rewriting engine—which is available in this article's code download—we'll spend the remaining two sections examining real-world uses of URL rewriting. First we'll look at how to use the URL rewriting engine and look at a simple URL rewriting example. Following that, we'll utilize the power of the rewriting engine's regular expression capabilities to provide truly "hackable" URLs.

Building a URL Rewriting Engine

To help illustrate how to implement URL rewriting in an ASP.NET Web application, I created a URL rewriting engine. This rewriting engine provides the following functionality:

- The ASP.NET page developer utilizing the URL rewriting engine can specify the rewriting rules in the Web.config file.
- The rewriting rules can use regular expressions to allow for powerful rewriting rules.
- URL rewriting can be easily configured to use an HTTP module or an HTTP handler.

In this article we will examine URL rewriting with just the HTTP module. To see how HTTP handlers can be used to perform URL rewriting, consult the code available for download with this article.

Specifying Configuration Information for the URL Rewriting Engine

Let's examine the structure of the rewrite rules in the Web.config file. First, you'll need to indicate in the Web.config file if you want perform URL rewriting with the HTTP module or the HTTP handler. In the download, the Web.config file contains two entries that have been commented out:

```
<!--
```

```
<httpModules>
   <add type="URLRewriter.ModuleRewriter, URLRewriter" 
        name="ModuleRewriter" />
</httpModules>
-->

<!--
<httpHandlers>
   <add verb="*" path="*.aspx" 
        type="URLRewriter.RewriterFactoryHandler, URLRewriter" />
</httpHandlers>
-->
```

Comment out the <httpModules> entry to use the HTTP module for rewriting; comment out the <httpHandlers> entry instead to use the HTTP handler for rewriting.

In addition to specifying whether the HTTP module or HTTP handler is used for rewriting, the Web.config file contains the rewriting rules. A rewriting rule is composed of two strings: the pattern to look for in the requested URL, and the string to replace the pattern with, if found. This information is expressed in the Web.config file using the following syntax:

```
<RewriterConfig>
   <Rules>
   <RewriterRule>
      <LookFor>pattern to look for</LookFor>
      <SendTo>string to replace pattern with</SendTo>
   </RewriterRule>
   <RewriterRule>
      <LookFor>pattern to look for</LookFor>
      <SendTo>string to replace pattern with</SendTo>
   </RewriterRule>
   ...
   </Rules>
</RewriterConfig>
```

Each rewrite rule is expressed by a **<RewriterRule>** element. The pattern to search for is specified by the **<LookFor>** element, while the string to replace the found pattern with is entered in the **<SentTo>** element. These rewrite rules are evaluated from top to bottom. If a match is found, the URL is rewritten and the search through the rewriting rules terminates.

When specifying patterns in the **<LookFor>** element, realize that regular expressions are used to perform the matching and string replacement. (In a bit we'll look at a real-world example that illustrates how to search for a pattern using regular expressions.) Since the pattern is a regular expression, be sure to escape any characters that are reserved characters in regular expressions. (Some of the regular expression reserved characters include: ., ?, ^, $, and others. These can be escaped by being preceded with a backslash, like \. to match a literal period.)

URL Rewriting with an HTTP Module

Creating an HTTP module is as simple as creating a class that implements the **IHttpModule** interface. The **IHttpModule** interface defines two methods:

- **Init(HttpApplication)**. This method fires when the HTTP module is initialized. In this method you'll wire up event handlers to the appropriate **HttpApplication** events.
- **Dispose()**. This method is invoked when the request has completed and been sent back to IIS. Any final cleanup should be performed here.

To facilitate creating an HTTP module for URL rewriting, I started by creating an abstract base class, **BaseModuleRewriter**. This class implements **IHttpModule**. In the **Init()** event, it wires up the **HttpApplication**'s **AuthorizeRequest** event to the **BaseModuleRewriter_AuthorizeRequest** method. The **BaseModuleRewriter_AuthorizeRequest** method calls the class's **Rewrite()** method passing in the requested **Path** along with the **HttpApplication** object that was passed into the **Init()** method. The **Rewrite()** method is abstract, meaning that in the **BaseModuleRewriter** class, the **Rewrite()** method has no method body; rather, the class being derived from **BaseModuleRewriter** *must* override this method and provide a method body.

With this base class in place, all we have to do now is to create a class derived from **BaseModuleRewriter** that overrides **Rewrite()** and performs the URL rewriting logic there. The code for **BaseModuleRewriter** is shown below.

```
public abstract class BaseModuleRewriter : IHttpModule
{
   public virtual void Init(HttpApplication app)
   {
      // WARNING!  This does not work with Windows authentication!
      // If you are using Windows authentication,
      // change to app.BeginRequest
      app.AuthorizeRequest += new
         EventHandler(this.BaseModuleRewriter_AuthorizeRequest);
   }

   public virtual void Dispose() {}

   protected virtual void BaseModuleRewriter_AuthorizeRequest(
     object sender, EventArgs e)
   {
      HttpApplication app = (HttpApplication) sender;
      Rewrite(app.Request.Path, app);
   }

   protected abstract void Rewrite(string requestedPath,
     HttpApplication app);
}
```

Notice that the **BaseModuleRewriter** class performs URL rewriting in the **AuthorizeRequest** event. Recall that if you use Windows authentication with file authorization, you will need to change this so that URL rewriting is performed in either the **BeginRequest** or **AuthenticateRequest** events.

The **ModuleRewriter** class extends the **BaseModuleRewriter** class and is responsible for performing the actual URL rewriting. **ModuleRewriter** contains a single overridden method—**Rewrite()**—which is shown below:

```
 protected override void Rewrite(string requestedPath,
    System.Web.HttpApplication app)
 {
    // get the configuration rules
    RewriterRuleCollection rules =
      RewriterConfiguration.GetConfig().Rules;

    // iterate through each rule...
    for(int i = 0; i < rules.Count; i++)
    {
       // get the pattern to look for, and
       // Resolve the Url (convert ~ into the appropriate directory)
       string lookFor = "^" +
         RewriterUtils.ResolveUrl(app.Context.Request.ApplicationPath,
         rules[i].LookFor) + "$";

       // Create a regex (note that IgnoreCase is set...)
       Regex re = new Regex(lookFor, RegexOptions.IgnoreCase);

       // See if a match is found
       if (re.IsMatch(requestedPath))
       {
          // match found - do any replacement needed
          string sendToUrl =
 RewriterUtils.ResolveUrl(app.Context.Request.ApplicationPath,
             re.Replace(requestedPath, rules[i].SendTo));

          // Rewrite the URL
          RewriterUtils.RewriteUrl(app.Context, sendToUrl);
          break;      // exit the for loop
       }
    }
 }
```

The **Rewrite()** method starts with getting the set of rewriting rules from the Web.config file. It then iterates through the rewrite rules one at a time, and for each rule, it grabs its **LookFor** property and uses a regular expression to determine if a match is found in the requested URL.

If a match is found, a regular expression replace is performed on the requested path with the value of the **SendTo** property. This replaced URL is then passed into the **RewriterUtils.RewriteUrl()** method. **RewriterUtils** is a helper class that provides a couple of static methods used by both the URL rewriting HTTP module and HTTP handler. The **RewriterUrl()** method simply calls the **HttpContext** object's **RewriteUrl()** method.

> **Note** You may have noticed that when performing the regular expression match and replacement, a call to **RewriterUtils.ResolveUrl()** is made. This helper method simply replaces any instances of **~** in the string with the value of the application's path.

The entire code for the URL rewriting engine is available for download with this article. We've examined the most germane pieces, but there are other components as well, such as classes for deserializing the XML-formatted rewriting rules in the Web.config file into an object, as well as the HTTP handler factory for URL rewriting. The remaining three sections of this article examine real-world uses of URL rewriting.

Performing Simple URL Rewriting with the URL Rewriting Engine

To demonstrate the URL rewriting engine in action, let's build an ASP.NET Web application that utilizes simple URL rewriting.

Imagine that we work for a company that sells assorted products online. These products are broken down into the following categories:

Category ID	**Category Name**
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
...	...

Assume we already have created an ASP.NET Web page called ListProductsByCategory.aspx that accepts a Category ID value in the querystring and displays all of the products belonging to that category. So, users who wanted to view our Beverages for sale would visit ListProductsByCategory.aspx?CategoryID=1, while users who wanted to view our Dairy Products would visit ListProductsByCategory.aspx?CategoryID=4. Also assume we have a page called ListCategories.aspx, which lists the categories of products for sale.

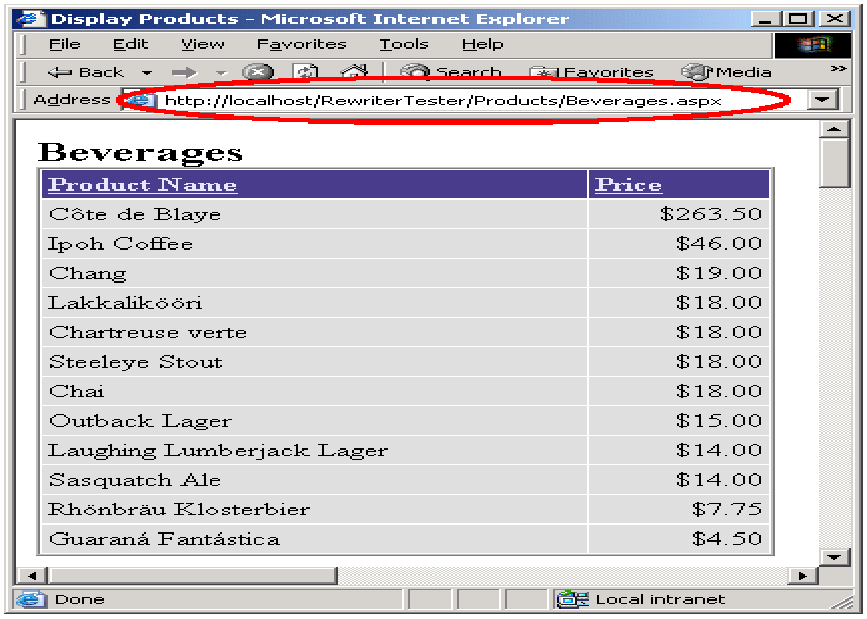
Clearly this is a case for URL rewriting, as the URLs a user is presented with do not carry any significance for the user, nor do they provide any "hackability." Rather, let's employ URL rewriting so that when a user visits /Products/Beverages.aspx, their URL will be rewritten to ListProductsByCategory.aspx?CategoryID=1. We can accomplish this with the following URL rewriting rule in the Web.config file:

```
<RewriterConfig>
   <Rules>
      <!-- Rules for Product Lister -->
      <RewriterRule>
         <LookFor>~/Products/Beverages\.aspx</LookFor>
         <SendTo>~/ListProductsByCategory.aspx?CategoryID=1</SendTo>
      </RewriterRule>
      <RewriterRule>
   </Rules>
</RewriterConfig>
```

As you can see, this rule searches to see if the path requested by the user was /Products/Beverages.aspx. If it was, it rewrites the URL as /ListProductsByCategory.aspx?CategoryID=1.

> **Note** Notice that the **<LookFor>** element escapes the period in Beverages.aspx. This is because the **<LookFor>** value is used in a regular expression pattern, and period is a special character in regular expressions meaning "match any character," meaning a URL of /Products/BeveragesQaspx, for example, would match. By escaping the period (using \.) we are indicating that we want to match a literal period, and not any old character.

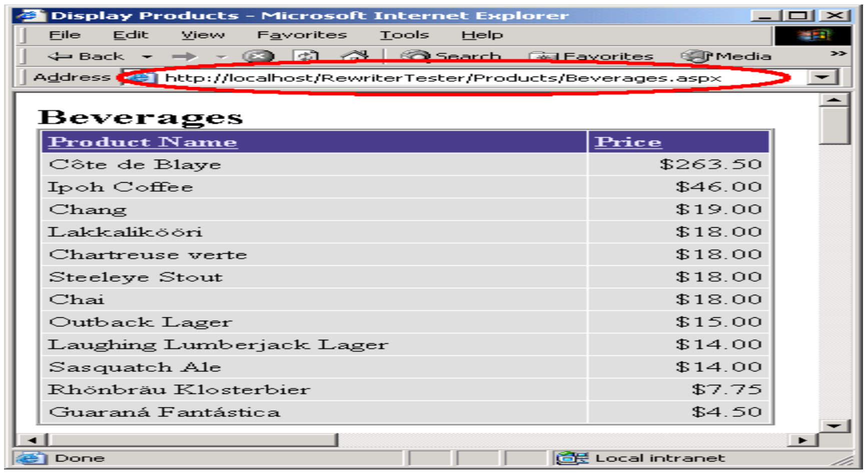
With this rule in place, when a user visits /Products/Beverages.aspx, they will be shown the beverages for sale. Figure 3 shows a screenshot of a browser visiting /Products/Beverages.aspx. Notice that in the browser's Address bar the URL reads /Products/Beverages.aspx, but the user is actually seeing the contents of ListProductsByCategory.aspx?CategoryID=1. (In fact, there doesn't even exist a /Products/Beverages.aspx file on the Web server at all!)


Display Products - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Done
Local intranet

http://localhost/RewriterTester/Products/Beverages.aspx

Beverages

Product Name	Price
Côte de Blaye	$263.50
Ipoh Coffee	$46.00
Chang	$19.00
Lakkalikööri	$18.00
Chartreuse verte	$18.00
Steeleye Stout	$18.00
Chai	$18.00
Outback Lager	$15.00
Laughing Lumberjack Lager	$14.00
Sasquatch Ale	$14.00
Rhönbräu Klosterbier	$7.75
Guaraná Fantástica	$4.50


Display Products - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/RewriterTester/Products/Beverages.aspx
Done
Local intranet

Beverages

Product Name	Price
Côte de Blaye	$263.50
Ipoh Coffee	$46.00
Chang	$19.00
Lakkalikööri	$18.00
Chartreuse verte	$18.00
Steeleye Stout	$18.00
Chai	$18.00
Outback Lager	$15.00
Laughing Lumberjack Lager	$14.00
Sasquatch Ale	$14.00
Rhönbräu Klosterbier	$7.75
Guaraná Fantástica	$4.50

Figure 3. Requesting category after rewriting URL

Similar to /Products/Beverages.aspx, we'd next add rewriting rules for the other product categories. This simply involves adding additional **<RewriterRule>** elements within the **<Rules>** element in the Web.config file. Consult the Web.config file in the download for the complete set of rewriting rules for the demo.

To make the URL more "hackable," it would be nice if a user could simply hack off the Beverages.aspx from /Products/Beverages.aspx and be shown a listing of the product categories. At first glance, this may appear a trivial task—just add a rewriting rule that maps /Products/ to /ListCategories.aspx. However, there is a fine subtlety—you must first create a /Products/ directory and add an empty Default.aspx file in the /Products/ directory.

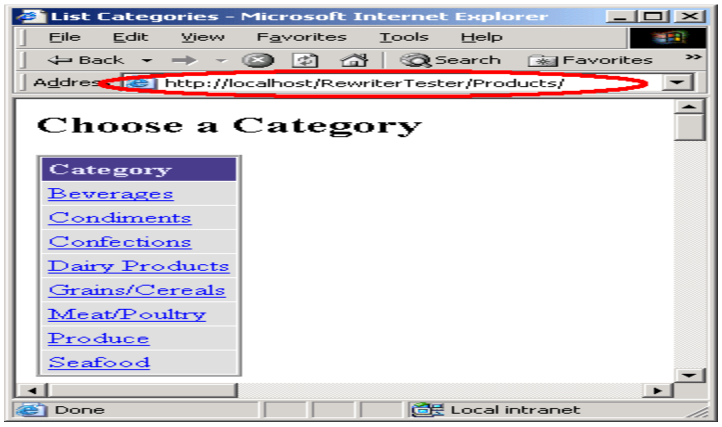
To understand why these extra steps need to be performed, recall that the URL rewriting engine is at the ASP.NET level. That is, if the ASP.NET engine is never given the opportunity to process the request, there's no way the URL rewriting engine can inspect the incoming URL. Furthermore, remember that IIS hands off incoming requests to the ASP.NET engine only if the requested file has an appropriate extension. So if a user visits /Products/, IIS doesn't see any file extension, so it checks the directory to see if there exists a file with one of the default filenames. (Default.aspx, Default.htm, Default.asp, and so on. These default filenames are defined in the Documents tab of the Web Server Properties dialog box in the IIS Administration dialog box.) Of course, if the /Products/ directory doesn't exist, IIS will return an HTTP 404 error.

So, we need to create the /Products/ directory. Additionally, we need to create a single file in this directory, Default.aspx. This way, when a user visits /Products/, IIS will inspect the directory, see that there exists a file named Default.aspx, and then hand off processing to the ASP.NET engine. Our URL rewriter, then, will get a crack at rewriting the URL.

After creating the directory and Default.aspx file, go ahead and add the following rewriting rule to the **<Rules>** element:

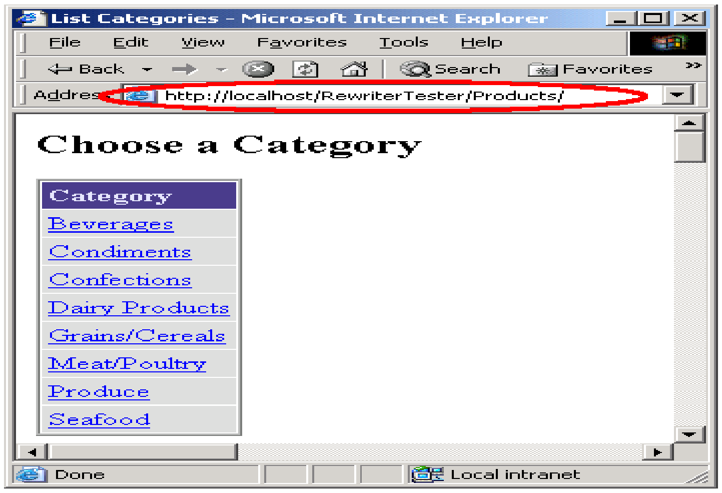
```
<RewriterRule>
   <LookFor>~/Products/Default\.aspx</LookFor>
   <SendTo>~/ListCategories.aspx</SendTo>
</RewriterRule>
```

With this rule in place, when a user visits /Products/ or /Products/Default.aspx, they will see the listing of product categories, shown in Figure 4.


List Categories - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/RewriterTester/Products/
Done
Local intranet

Choose a Category

Category
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood


List Categories - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/RewriterTester/Products/
Done Local intranet

Choose a Category

Category
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Figure 4. Adding "hackability" to the URL

Handling Postbacks

If the URLs you are rewriting contain a server-side Web Form and perform postbacks, when the form posts back, the underlying URL will be used. That is, if our user enters into their browser, /Products/Beverages.aspx, they will still see in their browser's Address bar, /Products/Beverages.aspx, but they will be shown the content for ListProductsByCategory.aspx?CategoryID=1. If ListProductsByCategory.aspx performs a postback, the user will be posted back to ListProductsByCategory.aspx?CategoryID=1, not /Products/Beverages.aspx. This won't break anything, but it can be disconcerting from the user's perspective to see the URL change suddenly upon clicking a button.

The reason this behavior happens is because when the Web Form is rendered, it explicitly sets its action attribute to the value of the file path in the **Request** object. Of course, by the time the Web Form is rendered, the URL has been rewritten from /Products/Beverages.aspx to ListProductsByCategory.aspx?CategoryID=1, meaning the **Request** object is reporting that the user is visiting ListProductsByCategory.aspx?CategoryID=1. This problem can be fixed by having the server-side form simply not render an action attribute. (Browsers, by default, will postback if the form doesn't contain an action attribute.)

Unfortunately, the Web Form does not allow you to explicitly specify an action attribute, nor does it allow you to set some property to disable the rendering of the action attribute. Rather, we'll have to extend the **System.Web.HtmlControls.HtmlForm** class ourselves, overriding the **RenderAttribute()** method and explicitly indicating that it not render the action attribute.

Thanks to the power of inheritance, we can gain all of the functionality of the **HtmlForm** class and only have to add a scant few lines of code to achieve the desired behavior. The complete code for the custom class is shown below:

```
namespace ActionlessForm {
  public class Form : System.Web.UI.HtmlControls.HtmlForm
  {
     protected override void RenderAttributes(HtmlTextWriter writer)
     {
        writer.WriteAttribute("name", this.Name);
        base.Attributes.Remove("name");

        writer.WriteAttribute("method", this.Method);
        base.Attributes.Remove("method");

        this.Attributes.Render(writer);

        base.Attributes.Remove("action");

        if (base.ID != null)
           writer.WriteAttribute("id", base.ClientID);
     }
  }
}
```

The code for the overridden **RenderAttributes()** method simply contains the exact code from the **HtmlForm** class's **RenderAttributes()** method, but without setting the action attribute. (I used Lutz Roeder's Reflector to view the source code of the **HtmlForm** class.)

Once you have created this class and compiled it, to use it in an ASP.NET Web application, start by adding it to the Web application's References folder. Then, to use it in place of the **HtmlForm** class, simply add the following to the top of your ASP.NET Web page:

```
<%@ Register TagPrefix="skm" Namespace="ActionlessForm"
   Assembly="ActionlessForm" %>
```

Then, where you have `<form runat="server">`, replace that with:

```
<skm:Form id="Form1" method="post" runat="server">
```

and replace the closing `</form>` tag with:

```
</skm:Form>
```

You can see this custom Web Form class in action in ListProductsByCategory.aspx, which is included in this article's download. Also included in the download is a Visual Studio .NET project for the action-less Web Form.

> **Note** If the URL you are rewriting to does not perform a postback, there's no need to use this custom Web Form class.

Creating Truly "Hackable" URLs

The simple URL rewriting demonstrated in the previous section showed how easily the URL rewriting engine can be configured with new rewriting rules. The true power of the rewriting rules, though, shines when using regular expressions, as we'll see in this section.

Blogs are becoming more and more popular these days, and it seems *everyone* has their own blog. If you are not familiar with blogs, they are often-updated personal pages that typically serve as an online journal. Most bloggers simply write about their day-to-day happenings, others focus on blogging about a specific theme, such as movie reviews, a sports team, or a computer technology.

Depending on the author, blogs are updated anywhere from several times a day to once every week or two. Typically the blog homepage shows the most recent 10 entries, but virtually all blogging software provides an archive through which visitors can read older postings. Blogs are a great application for "hackable" URLs. Imagine while searching through the archives of a blog you found yourself at the URL /2004/02/14.aspx. Would you be terribly surprised if you found yourself reading the posts made on February 14th, 2004? Furthermore, you might want to view all posts for February 2004, in which case you might try hacking the URL to /2004/02/. To view all 2004 posts, you might try visiting /2004/.

When maintaining a blog, it would be nice to provide this level of URL "hackability" to your visitors. While many blog engines provide this functionality, let's look at how it can be accomplished using URL rewriting.

First, we need a single ASP.NET Web page that will show blog entries by day, month, or year. Assume we have such a page, ShowBlogContent.aspx, that takes in querystring parameters year, month, and day. To view the posts for February 14th, 2004, we could visit ShowBlogContent.aspx?year=2004&month=2&day=14. To view all posts for February 2004, we'd visit ShowBlogContent.aspx?year=2004&month=2. Finally, to see all posts for the year 2004, we'd navigate to ShowBlogContent.aspx?year=2004. (The code for ShowBlogContent.aspx can be found in this article's download.)

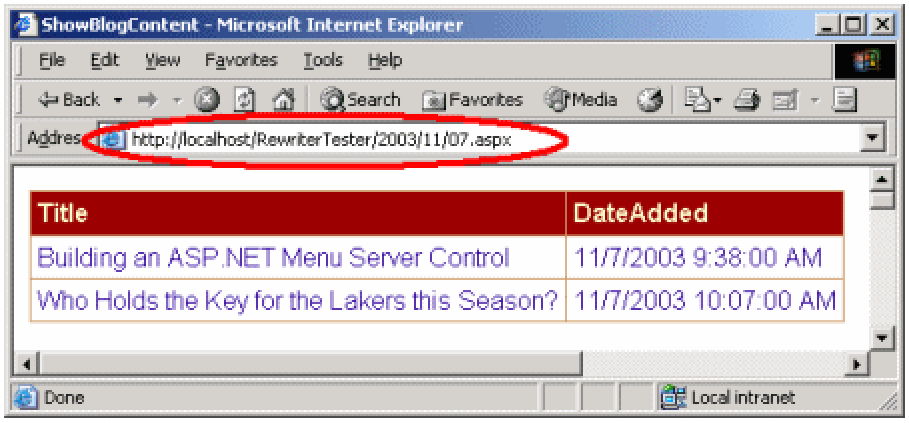
So, if a user visits /2004/02/14.aspx, we need to rewrite the URL to ShowBlogContent.aspx?year=2004&month=2&day=14. All three cases—when the URL specifies a year, month, and day; when the URL specifies just the year and month; and when the URL specifies only the yea—can be handled with three rewrite rules:

```
<RewriterConfig>
   <Rules>
      <!-- Rules for Blog Content Displayer -->
      <RewriterRule>
         <LookFor>~/(\d{4})/(\d{2})/(\d{2})\.aspx</LookFor>
         <SendTo>~/ShowBlogContent.aspx?year=$1&amp;month=$2&amp;day=$3</SendTo>
      </RewriterRule>
      <RewriterRule>
         <LookFor>~/(\d{4})/(\d{2})/Default\.aspx</LookFor>
         <SendTo><![CDATA[~/ShowBlogContent.aspx?year=$1&month=$2]]></SendTo>
      </RewriterRule>
      <RewriterRule>
         <LookFor>~/(\d{4})/Default\.aspx</LookFor>
         <SendTo>~/ShowBlogContent.aspx?year=$1</SendTo>
      </RewriterRule>
   </Rules>
</RewriterConfig>
```

These rewriting rules demonstrate the power of regular expressions. In the first rule, we look for a URL with the pattern `(\d{4})/(\d{2})/(\d{2})\.aspx`. In plain English, this matches a string that has four digits followed by a forward slash followed by two digits followed by a forward slash, followed by two digits followed by .aspx. The parenthesis around each digit grouping is vital—it allows us to refer to the matched characters inside those parentheses in the corresponding **<SendTo>** property. Specifically, we can refer back to the matched parenthetical groupings using $1, $2, and $3 for the first, second, and third parenthesis grouping, respectively.

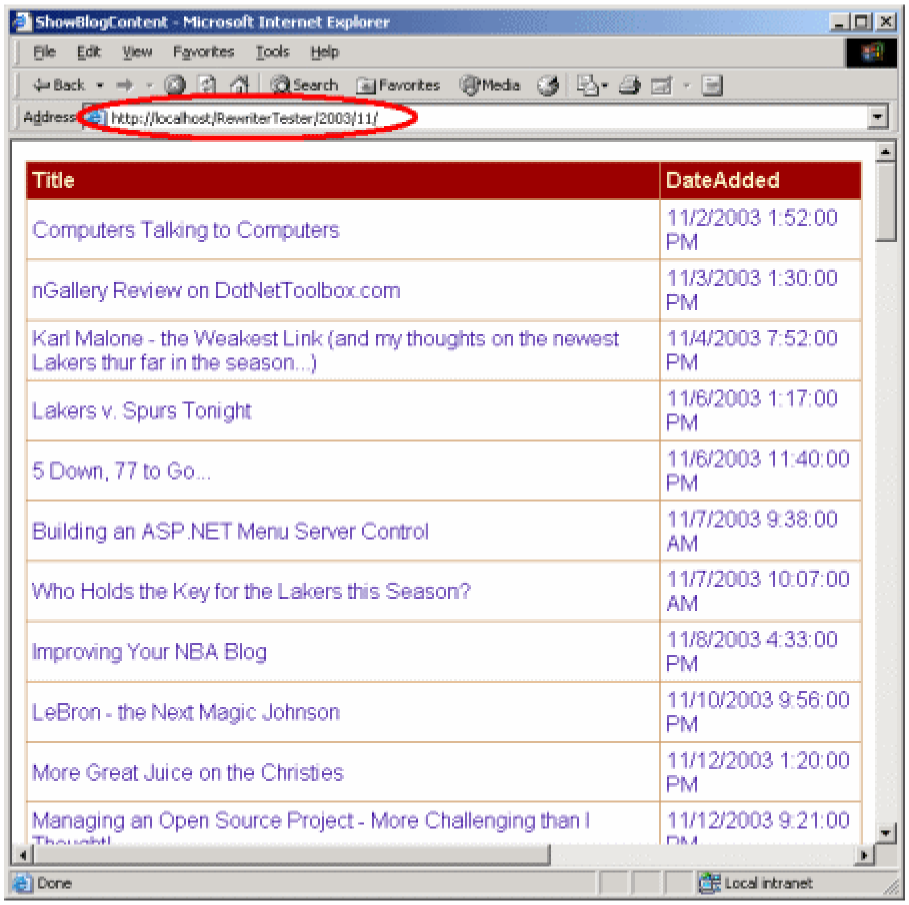
> **Note** Since the Web.config file is XML-formatted, characters like &, <, and > in the text portion of an element must be escaped. In the first rule's **<SendTo>** element, & is escaped to &amp;. In the second rule's **<SendTo>**, an alternative technique is used—by using a **<![CDATA[...]]>** element, the contents inside do not need to be escaped. Either approach is acceptable and accomplishes the same end.

Figures 5, 6, and 7 show the URL rewriting in action. The data is actually being pulled from my blog, http://ScottOnWriting.NET. In Figure 5, the posts for November 7, 2003 are shown; in Figure 6 all posts for November 2003 are shown; Figure 7 shows all posts for 2003.


ShowBlogContent - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/RewriterTester/2003/11/07.aspx
Done
Local intranet

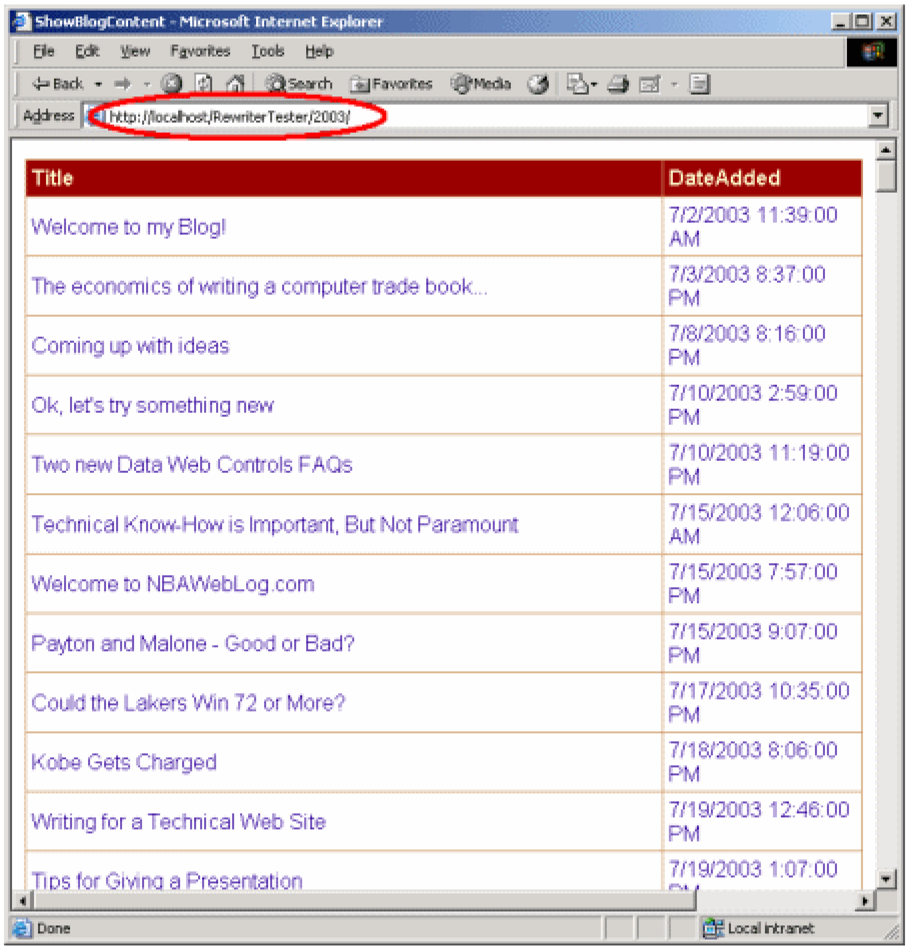
Title	DateAdded
Building an ASP.NET Menu Server Control	11/7/2003 9:38:00 AM
Who Holds the Key for the Lakers this Season?	11/7/2003 10:07:00 AM

Figure 5. Posts for November 7, 2003


ShowBlogContent - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/RewriterTester/2003/11/
Done
Local intranet

Title	DateAdded
Computers Talking to Computers	11/2/2003 1:52:00 PM
nGallery Review on DotNetToolbox.com	11/3/2003 1:30:00 PM
Karl Malone - the Weakest Link (and my thoughts on the newest Lakers thur far in the season...)	11/4/2003 7:52:00 PM
Lakers v. Spurs Tonight	11/6/2003 1:17:00 PM
5 Down, 77 to Go...	11/6/2003 11:40:00 PM
Building an ASP.NET Menu Server Control	11/7/2003 9:38:00 AM
Who Holds the Key for the Lakers this Season?	11/7/2003 10:07:00 AM
Improving Your NBA Blog	11/8/2003 4:33:00 PM
LeBron - the Next Magic Johnson	11/10/2003 9:56:00 PM
More Great Juice on the Christies	11/12/2003 1:20:00 PM
Managing an Open Source Project - More Challenging than I	11/12/2003 9:21:00

Figure 6. All posts for November 2003


ShowBlogContent - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address
Done
Local intranet

http://localhost/RewriterTester/2003/

Title	DateAdded
Welcome to my Blog!	7/2/2003 11:39:00 AM
The economics of writing a computer trade book...	7/3/2003 8:37:00 PM
Coming up with ideas	7/8/2003 8:16:00 PM
Ok, let's try something new	7/10/2003 2:59:00 PM
Two new Data Web Controls FAQs	7/10/2003 11:19:00 PM
Technical Know-How is Important, But Not Paramount	7/15/2003 12:06:00 AM
Welcome to NBAWebLog.com	7/15/2003 7:57:00 PM
Payton and Malone - Good or Bad?	7/15/2003 9:07:00 PM
Could the Lakers Win 72 or More?	7/17/2003 10:35:00 PM
Kobe Gets Charged	7/18/2003 8:06:00 PM
Writing for a Technical Web Site	7/19/2003 12:46:00 PM
Tips for Giving a Presentation	7/19/2003 1:07:00

Figure 7. All posts for 2003

> **Note** The URL rewriting engine expects a regular expression pattern in the **<LookFor>** elements. If you are unfamiliar with regular expressions, consider reading an earlier article of mine, An Introduction to Regular Expressions. Also, a great place to get your hands on commonly used regular expressions, as well as a repository for sharing your own crafted regular expressions, is RegExLib.com.

Building the Requisite Directory Structure

When a request comes in for /2004/03/19.aspx, IIS notes the .aspx extension and routes the request to the ASP.NET engine. As the request moves through the ASP.NET engine's pipeline, the URL will get rewritten to ShowBlogContent.aspx?year=2004&month=03&day=19 and the visitor will see those blog entries for March 19, 2004. But what happens when the user navigates to /2004/03/? Unless there is a directory /2004/03/, IIS will return a 404 error. Furthermore, there needs to be a Default.aspx page in this directory so that the request is handed off to the ASP.NET engine.

So with this approach, you have to manually create a directory for each year in which there are blog entries, with a Default.aspx page in the directory. Additionally, in each year directory you need to manually create twelve more directories—01, 02, Ã‚Â…, 12—each with a Default.aspx file. (Recall that we had to do the same thing—add a /Products/ directory with a Default.aspx file—in the previous demo so that visiting /Products/ correctly displayed ListCategories.aspx.)

Clearly, adding such a directory structure can be a pain. A workaround to this problem is to have all incoming IIS requests map to the ASP.NET engine. This way, even if when visiting the URL /2004/03/, IIS will faithfully hand off the request to the ASP.NET engine even if there does not exist a /2004/03/ directory. Using this approach, however, makes the ASP.NET engine responsible for handling all types of incoming requests to the Web server, including images, CSS files, external JavaScript files, Macromedia Flash files, and so on.

A thorough discussion of handling all file types is far beyond the scope of this article. For an example of an ASP.NET Web application that uses this technique, though, look into .Text, an open-source blog engine. .Text can be configured to have all requests mapped to the ASP.NET engine. It can handle serving all file types by using a custom HTTP handler that knows how to serve up typical static file types (images, CSS files, and so on).

Conclusion

In this article we examined how to perform URL rewriting at the ASP.NET-level through the **HttpContext** class's **RewriteUrl()** method. As we saw, **RewriteUrl()** updates the particular **HttpContext's Request** property, updating what file and path is being requested. The net effect is that, from the user's perspective, they are visiting a particular URL, but actually a different URL is being requested on the Web server side.

URLs can be rewritten either in an HTTP module or an HTTP handler. In this article we examined using an HTTP module to perform the rewriting, and looked at the consequences of performing the rewriting at different stages in the pipeline.

Of course, with ASP.NET-level rewriting, the URL rewriting can only happen if the request is successfully handed off from IIS to the ASP.NET engine. This naturally occurs when the user requests a page with a .aspx extension. However, if you want the person to be able to enter a URL that might not actually exist, but would rather rewrite to an existing ASP.NET page, you have to either create mock directories and Default.aspx pages, or configure IIS so that all incoming requests are blindly routed to the ASP.NET engine.

Related Books

ASP.NET: Tips, Tutorials, and Code

Microsoft ASP.NET Coding Strategies with the Microsoft ASP.NET Team

Essential ASP.NET with Examples in C#

Works consulted

URL rewriting is a topic that has received a lot of attention both for ASP.NET and competing server-side Web technologies. The Apache Web server, for instance, provides a module for URL rewriting called mod_rewrite. mod_rewrite is a robust rewriting engine, providing rewriting rules based on conditions such as HTTP headers and server variables, as well as rewriting rules that utilize regular expressions. For more information on mod_rewrite, check out A User's Guide to URL Rewriting with the Apache Web Server.

There are a number of articles on URL rewriting with ASP.NET. Rewrite.NET - A URL Rewriting Engine for .NET examines creating a URL rewriting engine that mimics mod_rewrite's regular expression rules. URL Rewriting With ASP.NET also gives a good overview of ASP.NET's URL rewriting capabilities. Ian Griffiths has a blog entry on some of the caveats associated with URL rewriting with ASP.NET, such as the postback issue discussed in this article. Both Fabrice Marguerie (read more) and Jason Salas (read more) have blog entires on using URL rewriting to boost search engine placement.

About the author

Scott Mitchell, author of five books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies for the past five years. Scott works as an independent consultant, trainer, and writer. He can be reached at mitchell@4guysfromrolla.com or through his blog, which can be found at http://ScottOnWriting.NET.

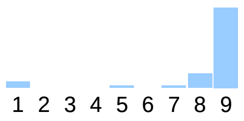
Print E-Mail

How would you rate the quality of this content?

	1	2	3	4	5	6	7	8	9	
Poor	○	○	○	○	○	○	○	○	○	Outstanding

Tell us why you rated the content this way. (optional)

Average rating:
8 out of 9



211 people have rated th

Manage Your Profile | Legal | Contact Us | MSDN Flash Newsletter

Microsoft