Excerpted from Professional Search Engine Optimization with ASP.NET: A Developer's Guide to SEO

URL Rewriting Using ISAPI_Rewrite

by Cristian Darie and Jaimie Sirovich

"**Click me!**" If the ideal URL could speak, its speech would resemble the communication of an experienced salesman. It would grab your attention with relevant keywords and a call to action; and it would persuasively argue that one should choose it instead of the other one. Other URLs on the page would pale in comparison.

URLs are more visible than many realize, and a contributing factor in CTR. They are often cited directly in copy, and they occupy approximately 20% of the real estate in a given search engine result page. Apart from "looking enticing" to humans, URLs must be friendly to search engines. URLs function as the "addresses" of all content in a web site. If confused by them, a search engine spider may not reach some of your content in the first place. This would clearly reduce search engine friendliness.

So let's enumerate all of the benefits of placing keywords in URLs:

1. Doing so has a small beneficial effect on search engine ranking in and of itself.
2. The URL is roughly 20% of the real estate you get in a SERP result. It functions as a call to action and increases perceived relevance.
3. The URL appears in the status bar of a browser when the mouse hovers over anchor text that references it. Again—it functions as a call to action and increases perceived relevance.
4. Keyword-based URLs tend to be easier to remember than `?ProductID=5&CategoryID=2`.
5. Query keywords, including those in the URL, are highlighted in search result pages.
6. Often, the URL is cited as the actual anchor text, that is:

```
<a href="http://www.example.com/foo.html">http://www.example.com/foo.html</a>
```

   Obviously, a user is more likely to click a link to a URL that contains relevant keywords, than a link that does not. Also, because keywords in anchor text *are* a decisive ranking factor, having keywords in the URL-anchor-text *will* help you rank better for "foos."

To sum up these benefits in one phrase:

Keyword-rich URLs are more aesthetically pleasing and more visible, and are likely to enhance your CTR and search engine rankings.

Implementing URL Rewriting

The hurdle we must overcome to support keyword-rich URLs like those shown earlier is that they don't actually exist anywhere in your web site. Your site still contains a script—named, say, `Product.aspx`—which expects to receive parameters through the query string and generate content depending on those parameters. This script would be ready to handle a request such as this:

```
http://www.example.com/Product.aspx?ProductID=123
```

but your web server would normally generate a 404 error if you tried any of the following:

```
http://www.example.com/Products/123.html
http://www.example.com/my-super-product.html
```

URL rewriting allows you to transform the URL of such an incoming request (which we'll call the *original URL*) to a different, existing URL (which we'll call the *rewritten URL*), according to a defined set of rules. You could use URL rewriting to transform the previous nonexistent URLs to `Product.aspx?ProductID=123`, which *does* exist.

If you happen to have some experience with the Apache web server, you probably know that it ships by default with the **mod_rewrite** module, which is the standard way to implement URL rewriting in the LAMP (Linux/Apache/MySQL/PHP) world. That is covered in the PHP edition of this book.

Unfortunately, IIS doesn't ship by default with such a module. IIS 7 contains a number of new features that make URL rewriting easier, but it will take a while until all existing IIS 5 and 6 web servers will be upgraded. Third-party URL-rewriting modules for IIS 5 and 6 do exist, and also several URL-rewriting libraries, hacks, and techniques, and each of them can (or cannot) be used depending on your version and configuration of IIS, and the version of ASP.NET. In this chapter we try to cover the most relevant scenarios by providing practical solutions.

To understand why an apparently easy problem—that of implementing URL rewriting—can become so problematic, you first need to understand how the process really works. To implement URL rewriting, there are three steps:

1. **Intercept the incoming request.** When implementing URL rewriting, it's obvious that you need to intercept the incoming request, which usually points to a resource that doesn't exist on your server physically. This task is not trivial when your web site is hosted on IIS 6 and older. There are different ways to implement URL rewriting depending on the version of IIS you use (IIS 7 brings some additional features over IIS 5/6), and depending on whether you implement rewriting using an IIS extension, or from within your ASP.NET application (using C# or VB.NET code). In this latter case, usually IIS still needs to be configured to pass the requests we need to rewrite to the ASP.NET engine, which doesn't usually happen by default.
2. **Associate the incoming URL with an existing URL on your server.** There are various techniques you can use to calculate what URL should be loaded, depending on the incoming URL. The "real" URL usually is a dynamic URL.
3. **Rewrite the original URL to the rewritten URL.** Depending on the technique used to capture the original URL and the form of the original URL, you have various options to specify the real URL your application should execute.

The result of this process is that the user requests a URL, but a different URL actually serves the request. The rest of the article covers how to implement these steps using ISAPI_Rewrite by Helicontech. For background information on how IIS processes incoming requests, we recommend Scott Mitchell's article "How ASP.NET Web Pages are Processed on the Web Server," located at `http://aspnet.4guysfromrolla.com/articles/011404-1.aspx`.

URL Rewriting with ISAPI_Rewrite v2

Using a URL rewriting engine such as Helicon's ISAPI_Rewrite has the following advantages over writing your own rewriting code:

* Simple implementation. Rewriting rules are written in configuration files; you don't need to write any supporting code.
* Task separation. The ASP.NET application works just as if it was working with dynamic URLs. Apart from the link building functionality, the ASP.NET application doesn't need to be aware of the URL rewriting layer of your application.
* You can easily rewrite requests for resources that are not processed by ASP.NET by default, such as those for image files, for example.

To process incoming requests, IIS works with ISAPI extensions, which are code libraries that process the incoming requests. IIS chooses the appropriate ISAPI extension to process a certain request depending on the extension of the requested file. For example, an ASP.NET-enabled IIS machine will redirect ASP.NET-specific requests (which are those for `.aspx` files, `.ashx` files, and so on), to the ASP.NET ISAPI extension, which is a file named `aspnet_isapi.dll`.

Figure 3-3 describes how an ISAPI_Rewrite fits into the picture. Its role is to rewrite the URL of the incoming requests, but doesn't affect the output of the ASP.NET script in any way.

At first sight, the rewriting rules can be added easily to an existing web site, but in practice there are other issues to take into consideration. For example, you'd also need to modify the existing links within the web site content. This is covered in Chapter 4 of Professional Search Engine Optimization with ASP.NET: A Developer's Guide to SEO.

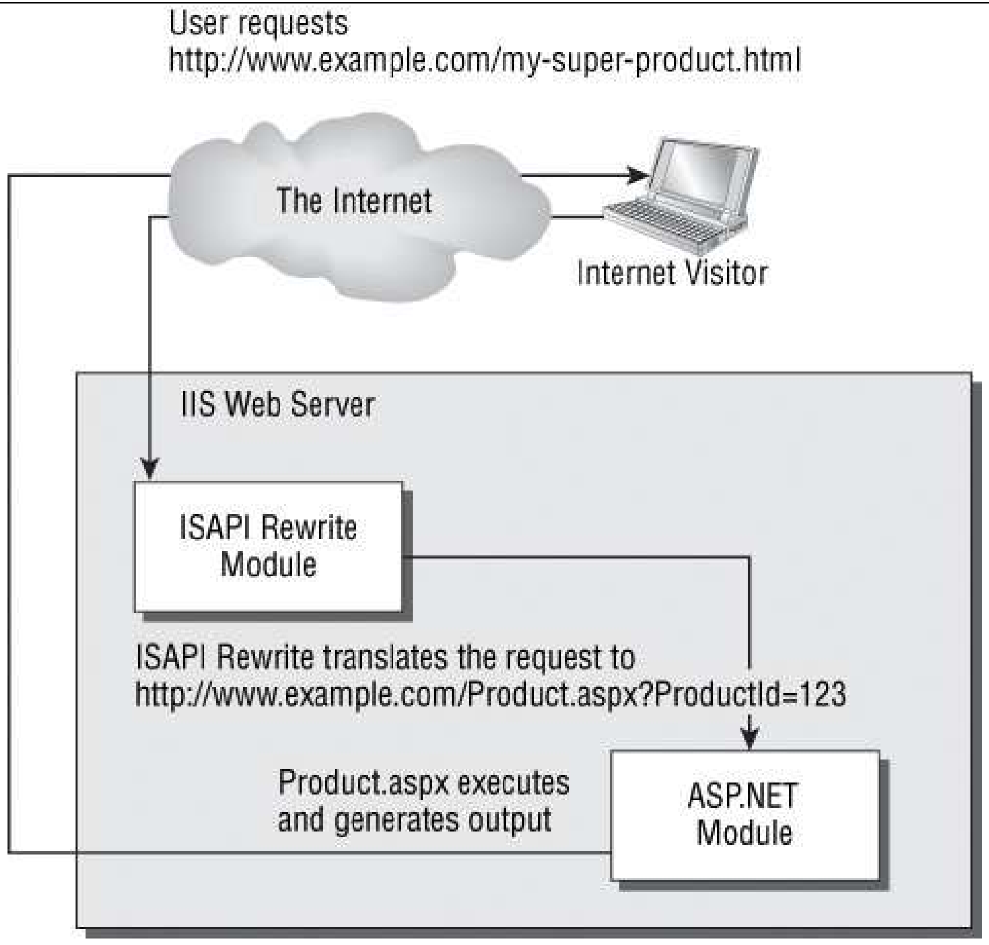

User requests
http://www.example.com/my-super-product.html
The Internet
Internet Visitor
IIS Web Server
ISAPI Rewrite Module
ISAPI Rewrite translates the request to
http://www.example.com/Product.aspx?ProductId=123
Product.aspx executes and generates output
ASP.NET Module

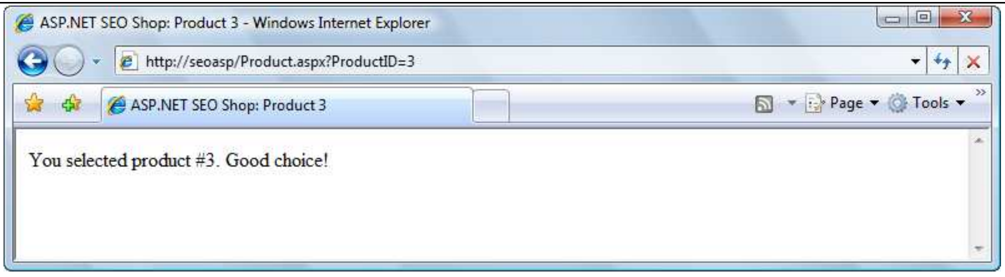
Figure 3-3

ISAPI_Rewrite allows the programmer to easily declare a set of rules that are applied by IIS on-the-fly to map incoming URLs requested by the visitor to dynamic query strings sent to various ASP.NET pages. As far as a search engine spider is concerned, the URLs are static.

The following few pages demonstrate URL rewriting functionality by using Helicon's ISAPI_Rewrite filter. You can find its official documentation at `http://www.isapirewrite.com/docs/`. Ionic's ISAPI rewriting module has similar functionality.

In the first exercise we'll create a simple rewrite rule that translates `my-super-product.html` to `Product.aspx?ProductID=123`. This is the exact scenario that was presented in Figure 3-3.

The `Product.aspx` Web Form is designed to simulate a real product page. The script receives a query string parameter named `ProductID`, and generates a very simple output message based on the value of this parameter. Figure 3-4 shows the sample output that you'll get by loading `http://seoasp/Product.aspx?ProductID=3`.


ASP.NET SEO Shop: Product 3 - Windows Internet Explorer
http://seoasp/Product.aspx?ProductID=3
ASP.NET SEO Shop: Product 3
Page
Tools

You selected product #3. Good choice!

Figure 3-4

In order to improve search engine friendliness, we want to be able to access the same page through a static URL: `http://seoasp/my-super-product.html`. To implement this feature, we'll use—you guessed it!—URL rewriting, using Helicon's ISAPI_Rewrite.

As you know, what ISAPI_Rewrite basically does is to translate an input string (the URL typed by your visitor) to another string (a URL that can be processed by your ASP.NET code). In this exercise we'll make it rewrite `my-super-product.html` to `Product.aspx?ProductID=123`.

This article covers ISAPI_Rewrite version 2. At the moment of writing, ISAPI_Rewrite 3.0 is in beta testing. The new version comes with an updated syntax for the configuration files and rewriting rules, which is compatible to that of the Apache mod_rewrite module, which is the standard rewriting engine in the Apache world. Please visit Cristian's web page dedicated to this book, `http://www.cristiandarie.ro/seo-asp/`, for updates and additional information regarding the following exercises.

Exercise: Using Helicon's ISAPI_Rewrite

1. The first step is to install ISAPI_Rewrite. Navigate to http://www.helicontech.com/download.htmand download ISAPI_Rewrite Lite (freeware). The file name should be something like `isapi_rwl_x86.msi`. At the time of writing, the full (not freeware) version of the product comes in a different package if you're using Windows Vista and IIS 7, but the freeware edition is the same for all platforms.

2. Execute the MSI file you just downloaded, and install the application using the default options all the way through.

If you run into trouble, you should visit the Installation section of the product's manual, at `http://www.isapirewrite.com/docs/#install`. If you run Windows Vista, you need certain IIS modules to be installed in order for ISAPI_Rewrite to function. If you configured IIS as shown in Chapter 1 of the book Professional Search Engine Optimization with ASP.NET: A Developer's Guide to SEO, you already have everything you need, and the installation of ISAPI_Rewrite should run smoothly.

3. Make sure your IIS web server is running and open the `http://seoasp/` web site using Visual Web Developer. (Code samples for this demo site are available from Wrox at http://www.wrox.com/WileyCDA/WroxTitle/productCd-0470131470,descCd-download_code.html.)

4. Create a new Web Form named `Product.aspx` in your project, with no code-behind file or Master Page. Then modify the generated code as shown in the following code snippet. (Remember that you can have Visual Web Developer generate the `Page_Load` signature for you by switching to Design view, and double-clicking an empty area of the page or using the Properties window.)

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  protected void Page_Load(object sender, EventArgs e)
  {
    // retrieve the product ID from the query string
    string productId = Request.QueryString["ProductID"];

    // use productId to customize page contents
    if (productId != null)
    {
      // set the page title
      this.Title += ": Product " + productId;

      // display product details
      message.Text =
        String.Format("You selected product #{0}. Good choice!", productId);
    }
    else
    {
      // display product details
      message.Text = "Please select a product from our catalog.";
    }

  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>ASP.NET SEO Shop</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:Literal runat="server" ID="message" />
  </form>
</body>
</html>
```

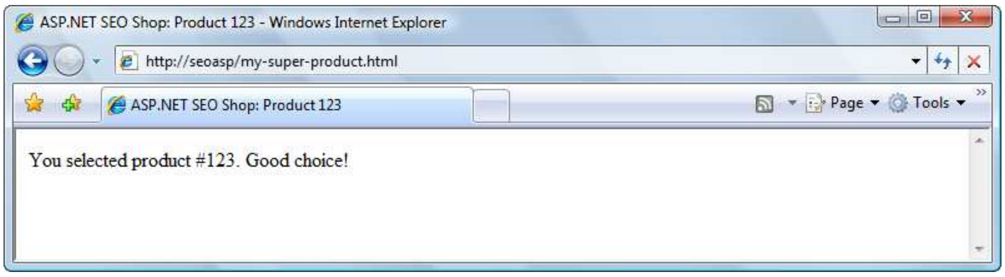
5. Test your Web Form by loading `http://seoasp/Product.aspx?ProductID=3`. The result should resemble Figure 3-4.

6. Let's now write the rewriting rule. Open the `Program Files/Helicon/ISAPI_Rewrite/httpd.ini` file (you can find a shortcut to this file in Programs), and add the following highlighted lines to the file. Note the file is read-only by default. If you use Notepad to edit it, you'll need to make it writable first.

```
[ISAPI_Rewrite]

# Translate /my-super.product.html to /Product.aspx?ProductID=123
```

```
RewriteRule ^/my-super-product\.html$ /Product.aspx?ProductID=123
```

7. Switch back to your browser again, and this time load `http://seoasp/my-super-product.html`. If everything works as it should, you should get the output that's shown in Figure 3-5.


ASP.NET SEO Shop: Product 123 - Windows Internet Explorer
http://seoasp/my-super-product.html
ASP.NET SEO Shop: Product 123
Page
Tools

You selected product #123. Good choice!

Figure 3-5

Congratulations! You've just written your first rewrite rule using Helicon's ISAPI_Rewrite. The free edition of this product only allows server-wide rewriting rules, whereas the commercial edition would allow you to use an application-specific `httpd.ini` configuration file, located in the root of your web site. However, this limitation shouldn't affect your learning process.

The exercise you've just finished features a very simplistic scenario, without much practical value—at least compared with what you'll learn next! Its purpose was to install ISAPI_Rewrite, and to ensure your working environment is correctly configured.

You started by creating a very simple ASP.NET Web Form that takes a numeric parameter from the query string. You could imagine this is a more involved page that displays lots of details about the product with the ID mentioned by the `ProductID` query string parameter, but in our case we're simply displaying a text message that confirms the ID has been correctly read from the query string.

`Product.aspx` is indeed very simple! It starts by reading the product ID value:

```
protected void Page_Load(object sender, EventArgs e)
{
  // retrieve the product ID from the query string
  string productId = Request.QueryString["ProductID"];
```

Next, we verify if the value we just read is `null`. If that is the case, then `ProductID` doesn't exist as a query string parameter. Otherwise, we display a simple text message, and update the page title, to confirm that `ProductID` was correctly read:

```
  // use productId to customize page contents
  if (productId != null)
  {
    // set the page title
    this.Title += ": Product " + productId;

    // display product details
    message.Text =
      String.Format("You selected product #{0}. Good choice!", productId);
    }
    else
    {
      // display product details
      message.Text = "Please select a product from our catalog.";
    }
```

URL Rewriting and ISAPI_Rewrite

As Figure 3-3 describes, the `Product.aspx` page is accessed *after* the original URL has been rewritten. This explains why `Request.QueryString["ProductID"]` reads the value of `ProductID` from the *rewritten* version of the URL. This is helpful, because the script works fine no matter if you accessed `Product.aspx` directly, or if the initial request was for another URL that was rewritten to `Product.aspx`.

The `Request.QueryString` collection, as well as the other values you can read through the `Request` object, work with the rewritten URL. For example, when requesting `my-super-product.html` in the context of our exercise, `Request.RawUrl` will return `/Product.aspx?ProductID=123`.

The rewriting engine allows you to retrieve the originally requested URL by saving its value to a server variable named `HTTP_X_REWRITE_URL`. You can read this value through `Request.ServerVariables["HTTP_X_REWRITE_URL"]`.This is helpful whenever you need to know what was the original request initiated by the client.

The `Request` class offers complete details about the current request. The following table describes the most commonly used `Request` members. You should visit the documentation for the complete list, or use IntelliSense in Visual Web Developer to quickly access the class members.

Server Variable	**Description**
`Request.RawURL`	Returns a string representing the URL of the request excluding the domain name, such as `/Product.aspx?ID=123`. When URL rewriting is involved, `RawURL` returns the rewritten URL.
`Request.Url`	Similar to `Request.RawURL`, except the return value is a `Uri` object, which also contains data about the request domain.
`Request.PhysicalPath`	Returns a string representing the physical path of the requested file, such as `C:\seoasp\Product.aspx`.
`Request.QueryString`	Returns a `NameValueCollection` object that contains the query string parameters of the request. You can use this object's indexer to access its values by name or by index, such as in `Request.QueryString[0]` or `Request.QueryString[ProductID]`.
`Request.Cookies`	Returns a `NameValueCollection` object containing the client's cookies.
`Request.Headers`	Returns a `NameValueCollection` object containing

	the request headers.
`Request.ServerVariables`	Returns a `NameValueCollection` object containing IIS variables.
`Request.ServerVariables[HTTP_X_REWRITE_URL]`	Returns a string representing the originally requested URL, when the URL is rewritten by Helicon's ISAPI_Rewrite or IIRF (Ionic ISAPI Rewrite).

After testing that `Product.aspx` works when accessed using its physical name (`http://seoasp/Product.aspx?ProductID=123`), we moved on to access this same script, but through a URL that doesn't physically exist on your server. We implemented this feature using Helicon's ISAPI_Rewrite.

As previously stated, the free version of Helicon's ISAPI_Rewrite only supports server-wide rewriting rules, which are stored in a file named `httpd.ini` in the product's installation folder (`\Program Files\Helicon\ISAPI_Rewrite`). This file has a section named `[ISAPI_Rewrite]`, usually at the beginning of the file, which can contain URL rewriting rules.

We added a single rule to the file, which translates requests to `/my-super-product.html` to `/Product.aspx?ProductID=123`. The line that precedes the `RewriteRule` line is a comment; comments are marked using the # character at the beginning of the line, and are ignored by the parser:

```
# Translate my-super.product.html to /Product.aspx?ProductID=123
RewriteRule ^/my-super-product\.html$ /Product.aspx?ProductID=123
```

In its basic form, `RewriteRule` takes two parameters. The first parameter *describes* the original URL that needs to be rewritten, and the second specifies what is should be rewritten to. The pattern that describes the form of the original URL is delimited by `^` and `$`, which mark the beginning and the end of the matched URL. The pattern is written using *regular expressions*, which you learn about in the next exercise.

In case you were wondering why the `.html` extension in the rewrite rule has been written as `\.html`, we will explain it now. In regular expressions—the programming language used to describe the original URL that needs to be rewritten—the dot is a character that has a special significance. If you want that dot to be read as a literal dot, you need to escape it using the backslash character. As you'll learn, this is a general rule with regular expressions: when special characters need to be read literally, they need to be escaped with the backslash character (which is a special character in turn—so if you wanted to use a backslash, it would be denoted as `\\`).

At the end of a rewrite rule you can also add one or more flag arguments, which affect the rewriting behavior. For example, the `[L]` flag, demonstrated in the following example, specifies that when a match is found the rewrite should be performed immediately, without processing any further `RewriteRule` entries:

```
RewriteRule ^/my-super-product\.html$ /Product.aspx?ProductID=123 [L]
```

These arguments are specific to the `RewriteRule` command, and not to regular expressions in general. Table 3-1 lists the possible `RewriteRule` arguments. The rewrite flags must always be placed in square brackets at the end of an individual rule.

Table 3-1

RewriteRule Option	Significance	Description
`I`	Ignore case	The regular expression of the `RewriteRule` and any corresponding `RewriteCond` directives is performed using case-insensitive matching.
`F`	Forbidden	In case the `RewriteRule` regular expression matches, the web server returns a 404 Not Found response, regardless of the format string (second parameter of `RewriteRule`) specified. Read Chapter 4 for more details about the HTTP status codes.
`L`	Last rule	If a match is found, stop processing further rules.
`N`	Next iteration	Restarts processing the set of rules from the beginning, but using the current rewritten URL. The number of restarts is limited by the value specified with the `RepeatLimit` directive.
`NS`	Next iteration of the same rule	Restarts processing the rule, using the rewritten URL. The number of restarts is limited by the value specified with the `RepeatLimit` directive, and is calculated independently of the number of restarts counted for the `N` directive.
`P`	Proxy	Immediately passes the rewritten URL to the ISAPI extension that handles proxy requests. The new URL must be a complete URL that includes the protocol, domain name, and so on.
`R`	Redirect	Sends a 302 redirect status code to the client pointing to the new URL, instead of rewriting the URL. This is always the last rule, even if the `L` flag is not specified.
`RP`	Permanent redirect	The same as R, except the 301 status code is used instead.
`U`	Unmangle log	Log the new URL as it was the originally requested URL.
`O`	Normalize	Normalize the URL before processing by removing illegal characters, and so on, and also deletes the query string.
`CL`	Lowercase	Changes the rewritten URL to lowercase.
`CU`	Uppercase	Changes the rewritten URL to uppercase.

Also, you should know that although `RewriteRule` is arguably the most important directive that you can use for URL rewriting with Helicon's ISAPI_Rewrite, it is not the only one. Table 3-2 quickly describes a few other directives. Please visit the product's documentation for a complete reference.

Table 3-2

Directive	Description
`RewriteRule`	This is the directive that allows for URL rewriting.
`RewriteHeader`	A generic version of `RewriteRule` that can rewrite any HTTP headers of the request. `RewriteHeader URL` is the same as `RewriteRule`.
`RewriteProxy`	Similar to `RewriteRule`, except it forces the result URL to be passed to the ISAPI extension that handles proxy requests.
`RewriteCond`	Allows defining one or more conditions (when more `RewriteCond` entries are used) that must be met before the following `RewriteRule`, `RewriteHeader`, or `RewriteProxy` directive is processed.

Introducing Regular Expressions

Before you can implement any really useful rewrite rules, it's important to learn about regular expressions. We'll teach them now, while discussing ISAPI_Rewrite, but regular expressions will also be needed when implementing other URL-related tasks, or when performing other kinds of string matching and parsing—so pay attention to this material.

Many love regular expressions, whereas others hate them. Many think they're very hard to work with, whereas many (or maybe not so many) think they're a piece of cake. Either way, they're one of those topics you can't avoid when URL rewriting is involved. We'll try to serve a gentle introduction to the subject, although entire books have been written on the subject. The Wikipedia page on regular expressions is great for background information (`http://en.wikipedia.org/wiki/Regular_expression`).

Appendix A of this book is a generic introduction to regular expressions. You should read it if you find that the theory in the following few pages—which is a fast-track introduction to regular expressions in the context of URL rewriting—is too sparse. For comprehensive coverage of regular expressions we recommend Andrew Watt's *Beginning Regular Expressions* (Wrox, 2005).

A regular expression (sometimes referred to as a *regex*) is a special string that describes a text *pattern*. With regular expressions you can define rules that match groups of strings, extract data from strings, and transform strings, which enable very flexible and complex text manipulation using concise rules. Regular expressions aren't specific to ISAPI_Rewrite, or even to URL rewriting in general. On the contrary, they've been around for a while, and they're implemented in many tools and programming languages, including the .NET Framework—and implicitly ASP.NET.

To demonstrate their usefulness with a simple example, we'll assume your web site needs to rewrite links as shown in Table 3-3.

Table 3-3

Original URL	**Rewritten URL**
`Products/P1.html`	`Product.aspx?ProductID=1`
`Products/P2.html`	`Product.aspx?ProductID=2`
`Products/P3.html`	`Product.aspx?ProductID=3`
`Products/P4.html`	`Product.aspx?ProductID=4`
…	…

If you have100,000 products, without regular expressions you'd be in a bit of a trouble, because you'd need to write just as many rules—no more, no less. *You don't want to manage a configuration file with 100,000 rewrite rules!* That would be unwieldy.

However, if you look at the Original URL column of the table, you'll see that all entries follow the same *pattern*. And as suggested earlier, regular expressions can come to rescue! Patterns are useful because with a single pattern you can match a theoretically infinite number of possible input URLs, so you just need to write a rewriting rule for every *type* of URL you have in your web site.

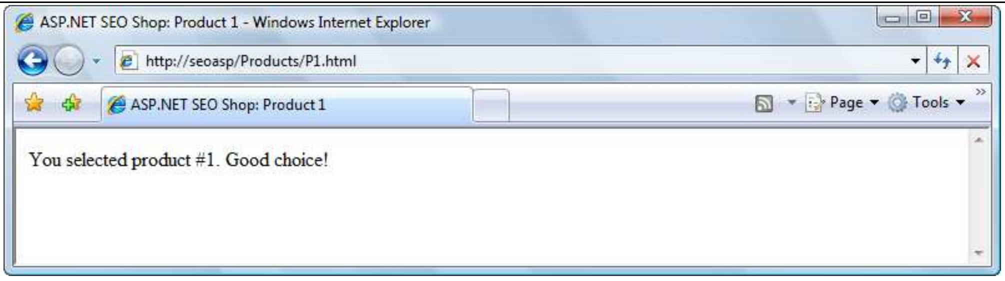
In the exercise that follows, we'll use a regular expression that matches `Products/P`***`n`***`.html`, and we'll use ISAPI_Rewrite to translate URLs that match that pattern to `Product.aspx?ProductID=`***`n`***. This will implement exactly the rules described in Table 3-3.

Exercise: Working with Regular Expressions

1. Open the `httpd.ini` configuration file and add the following rewriting rule to it.

```
[ISAPI_Rewrite]

# Defend your computer from some worm attacks
RewriteRule .*(?:global.asa|default\.ida|root\.exe|\.\.).* . [F,I,O]

# Translate my-super.product.html to /Product.aspx?ProductID=123
RewriteRule ^/my-super-product\.html$ /Product.aspx?ProductID=123

# Rewrite numeric URLs
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

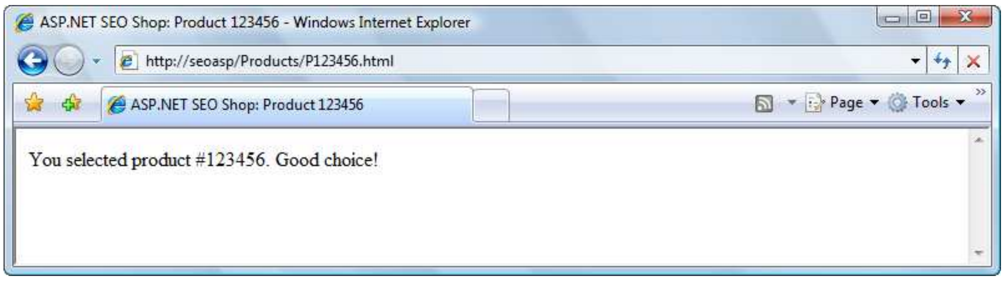
2. Switch back to your browser, and load `http://seoasp/Products/P1.html`. If everything works as planned, you will get the output that's shown in Figure 3-7.


ASP.NET SEO Shop: Product 1 - Windows Internet Explorer
http://seoasp/Products/P1.html
ASP.NET SEO Shop: Product 1
Page
Tools

You selected product #1. Good choice!

Figure 3-7

3. You can check that the rule really works, even for IDs formed of more digits. Loading `http://seoasp/Products/P123456.html` would give you the output shown in Figure 3-8.


ASP.NET SEO Shop: Product 123456 - Windows Internet Explorer
http://seoasp/Products/P123456.html
ASP.NET SEO Shop: Product 123456
Page
Tools

You selected product #123456. Good choice!

Figure 3-8

Note that by default, regular expression matching is case sensitive. So the regular expression in your `RewriteRule` directive will match `/Products/P123.html`, but will not match `/products/p123.html`, for example. Keep this in mind when performing your tests. To make the matching case sensitive, you need to use the `[I] RewriteRule` flag, as you'll soon learn.

Congratulations! The exercise was quite short, but you've written your first "real" regular expression! Let's take a closer look at your new rewrite rule:

```
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

If this is your first exposure to regular expressions, it must look scary! Just take a deep breath and read on: we promise, it's not as complicated as it looks.

As you learned in the previous exercise, a basic `RewriteRule` takes two arguments. In our example it also received a special flag—`[L]`—as a third argument. We'll discuss the meaning of these arguments next.

The first argument of `RewriteRule` is a regular expression that describes the *matching* URLs we want to rewrite. The second argument specifies the destination (*rewritten*) URL—this is *not* a regular expression. So, in geek-speak, the `RewriteRule` line from the exercise basically says: "rewrite any URL that *matches* the `^/Products/P([0-9]+)\.html$` pattern to

/Product.aspx?ProductID=$1." In English, the same line can be roughly read as: "delegate any request to a URL that looks like /Products/P***n***.html to /Product.aspx?ProductID=***n***."

In regular expressions, most characters, including alphanumeric characters, are read literally and simply match themselves. Remember the first RewriteRule you've written in this chapter to match my-super-product.html, which was mostly created of such "normal" characters. However, what makes regular expressions so powerful (and sometimes complicated), are the special characters (or *metacharacters*), such as ^, ., or *, which have special meanings. Table 3-4 describes the most frequently used metacharacters.

Table 3-4

Metacharacter	**Description**
^	Matches the beginning of the line. In our case, it will always match the beginning of the URL. The domain name isn't considered part of the URL, as far RewriteRule is concerned. It is useful to think of ^ as "anchoring" the characters that follow to the beginning of the string, that is, asserting that they be the first part.
.	Matches any single character.
*	Specifies that the preceding character or expression can be repeated zero or more times—not at all to an infinite number of times.
+	Specifies that the preceding character or expression can be repeated one or more times. In other words, the preceding character or expression must match at least once.
?	Specifies that the preceding character or expression can be repeated zero or one time. In other words, the preceding character or expression is optional.
{m,n}	Specifies that the preceding character or expression can be repeated between *m* and *n* times; *m* and *n* are integers, and *m* needs to be lower than *n*.
( )	The parentheses are used to define a *captured expression*. The string matching the expression between parentheses can be then read as a variable. The parentheses can also be used to group the contents therein, as in mathematics, and operators such as *, +, or ? can then be applied to the resulting expression.
[ ]	Used to define a character class. For example, [abc] will match any of the characters a, b, c. The - character can be used to define a range of characters. For example, [a-z] matches any lowercase letter. If - is meant to be interpreted literally, it should be the last character before]. Many metacharacters lose their special function when enclosed between [and], and are interpreted literally.
[^]	Similar to [ ], except it matches everything except the mentioned character class. For example, [^a-c] matches all characters except a, b, and c.
$	Matches the end of the line. In our case, it will always match the end of the URL. It is useful to think of it as "anchoring" the previous characters

	to the end of the string, that is, asserting that they be the last part.
`\`	The backslash is used to escape the character that follows. It is used to escape metacharacters when we need them to be taken for their literal value, rather than their special meaning. For example, `\.` will match a dot, rather than "any character" (the typical meaning of the dot in a regular expression). The backslash can also escape itself—so if you want to match `C:\Windows`, you'll need to refer to it as `C:\\Windows`.

Using Table 3-4 as reference, let's analyze the expression `^/Products/P([0-9]+)\.html$`. The expression starts with the `^` character, matching the beginning of the requested URL (remember, this doesn't include the domain name). The characters `/Products/P` assert that the next characters in the URL string match those characters.

Let's recap: the expression `^/Products/P` will match any URL that starts with `/Products/P`.

The next characters, `([0-9]+)`, are the crux of this process. The `[0-9]` bit matches any character between 0 and 9 (that is, any digit), and the `+` that follows indicates that the pattern can repeat one or more times, so we can have an entire number rather than just a digit. The enclosing round parentheses around `[0-9]+` indicate that the regular expression engine should store the matching string (which will be a digit or number) inside a variable called `$1`. (We'll need this variable to compose the rewritten URL.)

Finally, we have `\.html$`, which means that string should end in `.html`. The `\` is the escaping character that indicates that the `.` should be taken as a literal dot, not as "any character" (which is the significance of the `.` metacharacter). The `$` matches the end of the string.

The second argument of `RewriteRule`, `/Product.aspx?ProductID=$1`, plugs the digit or number extracted by the matching regular expression into the `$1` variable. If the regular expression matched more than one string, the subsequent matches could be referenced as `$2`, `$3`, and so on. You'll meet several such examples later in this book.

The second argument of `RewriteRule` isn't written using the regular expression language. Indeed, it doesn't need to, because it's not meant to match anything. Instead, it simply supplies the form of the rewritten URL. The only part with a special significance here are the variables (`$1`, `$2`, and so on) whose values are extracted from the expressions written between parentheses in the first argument of `RewriteRule`.

As you can see, this rule does indeed rewrite any request for a URL that looks like `/Products/P`***`n`***`.html` to `Product.aspx?ProductID=`***`n`***, which can be executed by our `Product.aspx` page. The `[L]` makes sure that if a match is found, the rewriting rules that follow won't be processed.

```
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

This is particularly useful if you have a long list of `RewriteRule` commands, because using `[L]` improves performance and prevents ISAPI_Rewrite from processing all the `RewriteRule` commands that follow once a match is found. This is usually what we want regardless.

Helicon's ISAPI_Rewrite ships with a regular expression tester application, which allows you to verify if a certain rewriting rule matches a test string. The application is named `RXTest.exe`, and is located in the product's installation folder (by default `Program Files\Helicon\ISAPI_Rewrite\`).

Rewriting Numeric URLs with Two Parameters

What you've accomplished in the previous exercise is rewriting numeric URLs with one parameter. We'll now expand that little example to also rewrite URLs with two parameters. The URLs with one parameter that we support looks like `http://seoasp/Products/Pn.html`. Now we'll assume that our links need to support links that include a category ID as well, in addition to the product ID. The new URLs will look like:

```
http://seoasp/Products/C2/P1.html
```

The existing `Product.aspx` script will be modified to handle links such as:

```
http://seoasp/Product.aspx?CategoryID=2&ProductID=1
```

As a quick reminder, here's the rewriting rule you used for numeric URLs with one parameter:

```
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

For rewriting two parameters, the rule would be a bit longer, but not much more complex:

```
RewriteRule ^/Products/C([0-9]+)/P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]
```

Let's put this to work in a quick exercise.

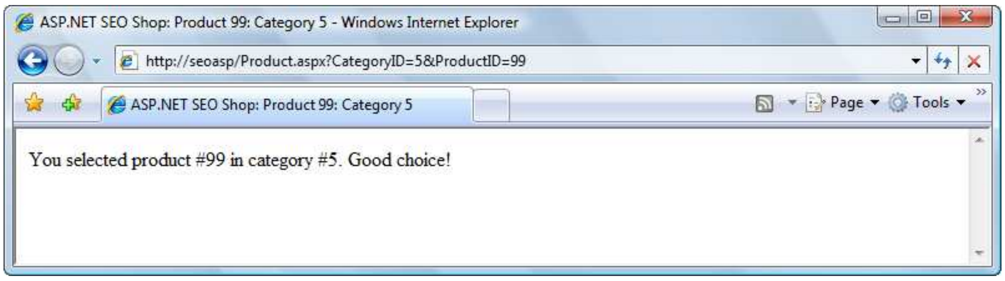
Exercise: Rewriting Numeric URLs

1. Modify your `Product.aspx` page that you created in the previous exercise, like this:

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  protected void Page_Load(object sender, EventArgs e)
  {
    // retrieve the product ID and category ID from the query string
    string productId = Request.QueryString["ProductID"];
    string categoryId = Request.QueryString["CategoryID"];

    // use productId to customize page contents
    if (productId != null && categoryId == null)
    {
      // set the page title
      this.Title += ": Product " + productId;

      // display product details
      message.Text =
        String.Format("You selected product #{0}. Good choice!", productId);
    }
    // use productId and categoryId to customize page contents
    else if (productId != null && categoryId != null)
    {
      // set the page title
      this.Title +=
        String.Format(": Product {0}: Category {1}", productId, categoryId);

      // display product details
      message.Text =
        String.Format("You selected product #{0} in category #{1}. Good choice!",
                      productId, categoryId);
    }
    else
    {
      // display product details
      message.Text = "Please select a product from our catalog.";
    }

  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>ASP.NET SEO Shop</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:Literal runat="server" ID="message" />
  </form>
</body>
</html>
```

2. Test your script with a URL that contains just a product ID, such as `http://seoasp/Products/P123456.html`, to ensure that the old functionality still works. The result should resemble Figure 3-8.

3. Now test your script by loading `http://seoasp/Product.aspx?CategoryID=5&ProductID=99`. You should get the output shown in Figure 3-9.


ASP.NET SEO Shop: Product 99: Category 5 - Windows Internet Explorer
http://seoasp/Product.aspx?CategoryID=5&ProductID=99
ASP.NET SEO Shop: Product 99: Category 5
Page
Tools

You selected product #99 in category #5. Good choice!

Figure 3-9

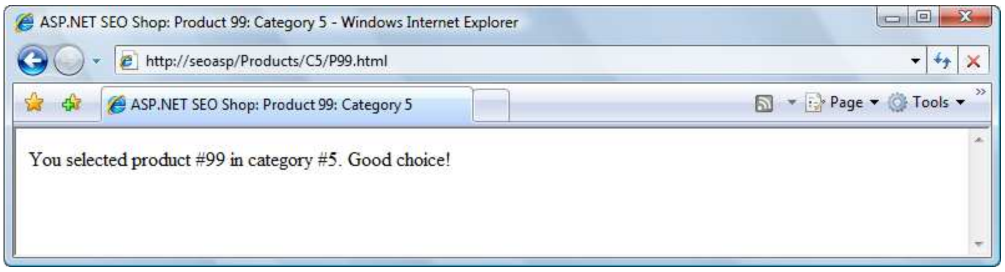
4. Add a new rewriting rule to the `httpd.ini` file as shown here:

```
[ISAPI_Rewrite]
```

```
# Defend your computer from some worm attacks
RewriteRule .*(?:global.asa|default\.ida|root\.exe|\.\.).* . [F,I,O]

# Translate my-super.product.html to /Product.aspx?ProductID=123
RewriteRule ^/my-super-product\.html$ /Product.aspx?ProductID=123

# Rewrite numeric URLs that contain a product ID
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]

# Rewrite numeric URLs that contain a product ID and a category ID
RewriteRule ^/Products/C([0-9]+)/P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]
```

Note that the entire `RewriteRule` command and its parameters must be written on a single line in your `httpd.ini` file. If you split it in two lines as printed in the book, it will not work.

5. Load `http://seoasp/Products/C5/P99.html`, and expect to get the same output as with the previous request, as shown in Figure 3-10.


ASP.NET SEO Shop: Product 99: Category 5 - Windows Internet Explorer
http://seoasp/Products/C5/P99.html
ASP.NET SEO Shop: Product 99: Category 5
Page
Tools

You selected product #99 in category #5. Good choice!

Figure 3-10

In this example you started by modifying `Product.aspx` to accept URLs that accept a product ID and a category ID. Then you added URL rewriting support for URLs with two numeric parameters. You created a rewriting rule to your `httpd.ini` file, which handles URLs with two parameters:

```
RewriteRule ^/Products/C([0-9]+)/P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]
```

The rule looks a bit complicated, but if you look carefully, you'll see that it's not so different from the rule handling URLs with a single parameter. The rewriting rule has now two parameters—`$1` is the number that comes after `/Products/C`, and is defined by `([0-9]+)`, and the second parameter, `$2`, is the number that comes after `/P`.

The result is that we now delegate any URL that looks like `/Products/C`*`m`*`/P`*`n`*`.html` to `/Product.aspx?CategoryID=`*`m`*`&ProductID=`*`n`*.

Rewriting Keyword-Rich URLs

Here's where the real fun begins! This kind of URL rewriting is a bit more complex, and there are more strategies you could take. When working with rewritten numeric URLs, it was relatively easy to extract the product and category IDs from a URL such as `/Products/C5/P9.html`, and rewrite the URL to `Product.aspx?CategoryID=5&ProductID=9`.

A keyword-rich URL doesn't necessarily have to include any IDs. Take a look at this one:

```
http://www.example.com/Products/Tools/Super-Drill.html
```

(You met a similar example in the first exercise of this chapter, where you handled the rewriting of `http://seoasp/my-super-product.html`.)

This URL refers to a product named "Super Drill" located in a category named "Tools." Obviously, if you want to support this kind of URL, you need some kind of mechanism to find the IDs of the category and product the URL refers to.

One solution that comes to mind is to add a column in the product information table that associates such beautified URLs to "real" URLs that your application can handle. In such a request you could look up the information in the Category and Product tables, get their IDs, and use them. We demonstrate this technique in an exercise later in this chapter.

We also have a solution for those who prefer an automated solution that doesn't involve a lookup database. This solution still brings the benefits of a keyword-rich URL, while being easier to implement. Look at the following URLs:

```
http://www.example.com/Products/Super-Drill-P9.html
http://www.example.com/Products/Tools-C5/Super-Drill-P9.html
```

These URLs include keywords. However, we've sneaked IDs in these URLs, in a way that isn't unpleasant to the human eye, and doesn't distract attention from the keywords that matter, either. In the case of the first URL, the rewriting rule can simply extract the number that is tied at the end of the product name (`-P9`), and ignore the rest of the URL. For the second URL, the rewriting rule can extract the category ID (`-C5`) and product ID (`-P9`), and then use these numbers to build a URL such as `Product.aspx?CategoryID=5&ProductID=9`.

This book generally uses such keyword-rich URLs, which also contain item IDs. Later in this chapter, however, you'll be taught how to implement ID-free keyword-rich URLs as well.

The rewrite rule for keyword-rich URLs with a single parameter looks like this:

```
RewriteRule ^/Products/.*-P([0-9]+)\.html?$ /Product.aspx?ProductID=$1 [L]
```

The rewrite rule for keyword-rich URLs with two parameters looks like this:

```
RewriteRule ^/Products/.*-C([0-9]+)/.*-P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]
```

Let's see these rules at work in an exercise.

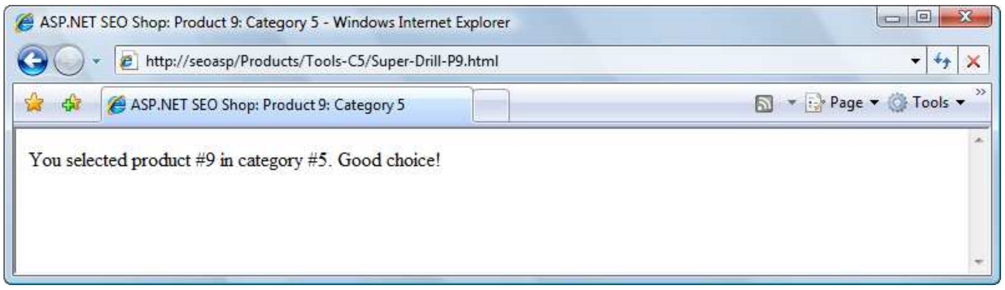
Exercise: Rewriting Keyword-Rich URLs

1. Modify the `httpd.ini` configuration file like this:

```
[ISAPI_Rewrite]

# Rewrite numeric URLs that contain a product ID
RewriteRule ^/Products/P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

```
# Rewrite numeric URLs that contain a product ID and a category ID
RewriteRule ^/Products/C([0-9]+)/P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]

# Rewrite keyword-rich URLs with a product ID and a category ID
RewriteRule ^/Products/.*-C([0-9]+)/.*-P([0-9]+)\.html$ @@ta
/Product.aspx?CategoryID=$1&ProductID=$2 [L]

# Rewrite keyword-rich URLs with a product ID
RewriteRule ^/Products/.*-P([0-9]+)\.html$ /Product.aspx?ProductID=$1 [L]
```

2. Load `http://seoasp/Products/Tools-C5/Super-Drill-P9.html`, and voila, you should get the result that's shown in Figure 3-12.


ASP.NET SEO Shop: Product 9: Category 5 - Windows Internet Explorer
http://seoasp/Products/Tools-C5/Super-Drill-P9.html
ASP.NET SEO Shop: Product 9: Category 5
Page
Tools

You selected product #9 in category #5. Good choice!

Figure 3-12

3. To test the rewrite rule that matches product keyword-rich URLs that don't include a category, try loading `http://seoasp/Products/Super-Drill-P9.html`. The result should be the expected one.

There's one interesting gotcha for you to keep in mind when developing web applications, especially when they use URL rewriting. Your web browser sometimes caches the results returned by your URLs—which can lead to painful debugging experiences—so we recommend that you disable your browser's cache during developing.

You now have two new rules in your `httpd.ini` file, and they are working beautifully! The first rule handles keyword-rich URLs that include a product ID and a category ID, and the second rule handles keyword-rich URLs that include only a product ID. Note that the order of these rules is important, because the second rule matches the URLs that are meant to be captured by the first rule. Also remember that because we didn't use the `[I]` flag, the matching is case sensitive.

The first new rule matches URLs that start with the string `/Products/`, then contain a number of zero or more characters (`.*`), followed by `–C`. This is expressed by `^/Products/.*-C`. The next characters must be one or more digits, which as a whole are saved to the `$1` variable, because the expression is written between parentheses –`([0-9]+)`. This first variable extracted from the URL, `$1`, is the category ID.

After the category ID, the URL must contain a slash, then zero or more characters (`.*`), then `-P`, as expressed by `/.*-P`. Afterwards, another captured group follows, to extract the ID of the product, ([0-9]+), which becomes the $2 variable. The final bit of the regular expression, \.html$, specifies the URL needs to end in .html.

The two extracted values, $1 and $2, are used to create the new URL, /Product.aspx?CategoryID=$1&ProductID=$2.

The second rewrite rule you implemented is a simpler version of this one.

Technical Considerations

Apart from basic URL rewriting, no matter how you implement it, you need to be aware of additional technical issues you may encounter when using such techniques in your web sites:

* If your web site contains ASP.NET controls or pages that generate postback events that you handled at server-side, you need to perform additional changes to your site so that it handles the postbacks correctly.
* You need to make sure the relative links in your pages point to the correct absolute locations after URL rewriting.

Let's deal with these issues one at a time.

Handling Postbacks Correctly

Although they appear to be working correctly, the URL-rewritten pages you've loaded in all the exercises so far have a major flaw: they can't handle postbacks correctly. Postback is the mechanism that fires server-side handlers as response of client events by submitting the ASP.NET form. In other words, a postback occurs every time a control in your page that has the runat="server" attribute fires an event that is handled at server-side with C# or VB.NET code.

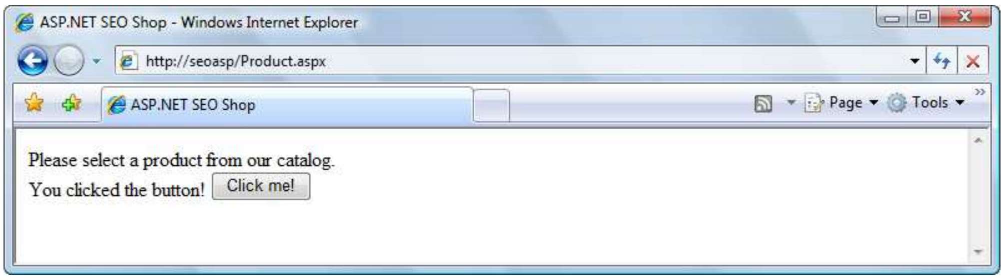
To understand the flaw in our solution, add the following button into the form in Product.aspx:

```
<body>
  <form id="form1" runat="server">
    <asp:Literal runat="server" ID="message" />
    <asp:Button ID="myButton" runat="server" Text="Click me!" />
  </form>
</body>
```

Switch the form to Design view, and double-click the button in the designer to have Visual Web Developer generate its Click event handler for you. Then complete its code by adding the following line:

```
protected void myButton_Click(object sender, EventArgs e)
{
  message.Text += "<br />You clicked the button!";
}
```

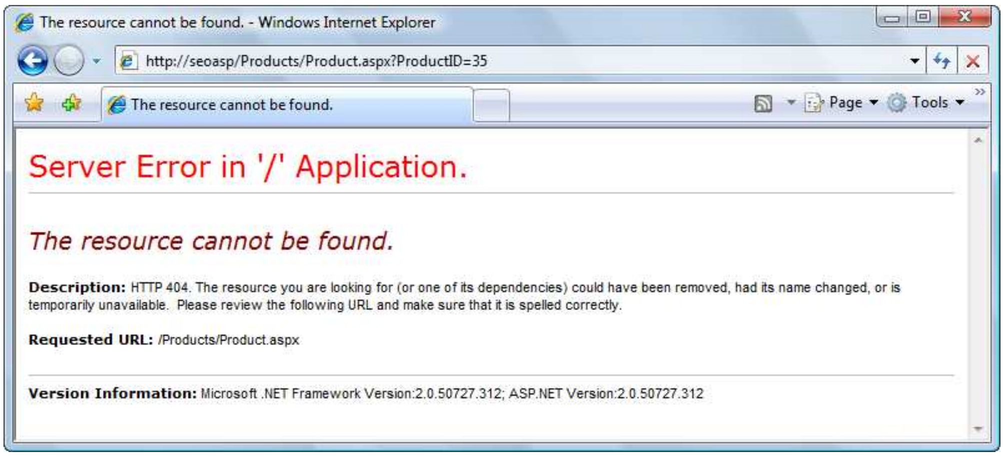
Alright, you have one button that displays a little message when clicked. To test this button, load http://seoasp/Product.aspx, and click the button to ensure it works as expected. The result should resemble that in Figure 3-16. (Note that clicking it multiple times doesn't display additional text, because the contents of the `Literal` control used for displaying the message is refreshed on every page load.)


ASP.NET SEO Shop - Windows Internet Explorer
http://seoasp/Product.aspx
ASP.NET SEO Shop
Page
Tools
Click me!

Please select a product from our catalog.
You clicked the button!

Figure 3-16

Now, load the same `Product.aspx` form, but this time using a rewritten URL. I'll choose `http://seoasp/Products/Super-AJAX-PHP-Book-P35.html`, which should be properly handled by your existing code and rewritten to `http://seoasp/Product.aspx?ProductID=35`. Then click the button. Oops! You'll get an error, as shown in Figure 3-17.


The resource cannot be found. - Windows Internet Explorer
http://seoasp/Products/Product.aspx?ProductID=35
The resource cannot be found.
Page
Tools

Server Error in '/' Application.

The resource cannot be found.

Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly.

Requested URL: /Products/Product.aspx

Version Information: Microsoft .NET Framework Version:2.0.50727.312; ASP.NET Version:2.0.50727.312

Figure 3-17

If you look at the new URL in the address bar of your web browser, you can intuit what happens: the page is unaware that it was loaded using a rewritten URL, and it submits the form to the wrong URL—in this example, `http://seoasp/Products/Product.aspx?ProductID=35`. The presence of the `Products` folder in the initial URL broke the path to which the form is submitted.

The new URL doesn't exist physically in our web site, and it's also not handled by any rewrite rules. This happens because the `action` attribute of the form points back to the name of the physical page it's located on, which in this case is `Products.aspx` (this behavior isn't configurable via properties). This can be verified simply by looking at the HTML source of the form, before clicking the button:

```
<form name="form1" method="post" action="Product.aspx?ProductID=35" id="form1">
```

When this form is located on a page that contains folders, the action path will be appended to the path including the folders. When URL rewriting is involved, it's easy to intuit that this behavior isn't what we want. Additionally, even if the original path doesn't contain folders, the form still submits to a dynamic URL, rendering our URL rewriting efforts useless.

To overcome this problem, there are three potential solutions. The first works with any version of ASP.NET, and involves creating a new `HtmlForm` class that removes the `action` attribute, like this:

```
namespace ActionlessForm
{
  public class Form : System.Web.UI.HtmlControls.HtmlForm
  {
    protected override void RenderAttributes(System.Web.UI.HtmlTextWriter writer)
    {
      Attributes.Add("enctype", Enctype);
      Attributes.Add("id", ClientID);
      Attributes.Add("method", Method);
      Attributes.Add("name", Name);
      Attributes.Add("target", Target);
      Attributes.Render(writer);
    }
  }
}
```

If you save this file as `ActionlessForm.cs`, you can compile it into a library file using the C# compiler, like this:

```
csc.exe /target:library ActionlessForm.cs
```

The default location of the .NET 2.0 C# compiler is `\windows\microsoft.net\framework\v2.0.50727\csc.exe`. Note that you may need to download and install the Microsoft .NET Software Development Kit to have access to the C# compiler. To create libraries you can also use Visual C# 2005 Express Edition, in which case you don't need to compile the C# file yourself. Copying the resulted file, `SuperHandler.dll`, to the `Bin` folder of your application would make it accessible to the rest of the application. Then you'd need to replace all the `<form>` elements in your Web Forms and Master Pages with the new form, like this:

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<%@ Register TagPrefix="af" Namespace="ActionlessForm" Assembly="ActionlessForm" %>

...

<html xmlns="http://www.w3.org/1999/xhtml" >
<head id="Head1" runat="server">
  <title>ASP.NET SEO Shop</title>
</head>
<body>
  <af:form id="form1" runat="server">
    <asp:Literal runat="server" ID="message" />
    <asp:Button ID="myButton" runat="server" Text="Click me!"
OnClick="myButton_Click" />
```

```
  </af:form>
</body>
</html>
```

Needless to say, updating all your Web Forms and Master Pages like this isn't the most elegant solution in the world, but it's the best option you have with ASP.NET 1.x. Fortunately, ASP.NET 2.0 offers a cleaner solution, which doesn't require you to alter your existing pages, and it consists of using the ASP.NET 2.0 Control Adapter extensibility architecture. This method is covered by Scott Guthrie in his article at `http://weblogs.asp.net/scottgu/archive/2007/02/26/tip-trick-url-rewriting-with-asp-net.aspx`.

The last solution implies using `Context.RewritePath` to rewrite the current path to `/?`, effectively stripping the `action` tag of the form. This technique is demonstrated in the case study in Chapter 14 in Professional Search Engine Optimization with ASP.NET: A Developer's Guide to SEO, but as you'll see, it's not recommended that you use it in more complex applications because of the restrictions it implies on your code, and its potential side effects.

Absolute Paths and ~/

Another potential problem when using URL rewriting is that relative links will stop working when folders are used. For example, a link to `/image.jpg` in `Product.aspx` would be translated to `http://seoasp/image.jpg` if read from `Product.aspx?ProductID=10`, or to `http://seoasp/Products/image.jpg` if read through a rewritten URL such as `http://seoasp/Products/P-10.html`. To avoid such problems, you should use at least one of the following two techniques:

* Always use absolute paths. Creating a URL factory library, as shown later in this chapter, can help with this task.
* Use the ~ syntax supported by ASP.NET controls. The ~ symbol always references the root location of your application, and it is replaced by its absolute value when the controls are rendered by the server.

Problems Rewriting Doesn't Solve

URL rewriting is not a panacea for all dynamic site problems. In particular, URL rewriting in and of itself does not solve any duplicate content problems. If a given site has duplicate content problems with a dynamic approach to its URLs, the problem would likely also be manifest in the resulting rewritten static URLs as well. In essence, URL rewriting only obscures the parameters—however many there are, from the search engine spider's view. This is useful for URLs that have many parameters as we mentioned. Needless to say, however, if the varying permutations of obscured parameters *do not* dictate significant changes to the content, the same duplicate content problems remain.

A simple example would be the case of rewriting the page of a product that can exist in multiple categories. Obviously, these two pages would probably show duplicate (or very similar content) even if accessed through static-looking links, such as:

```
http://www.example.com/College-Books-C1/Some-Book-Title-P2.html
http://www.example.com/Out-of-Print-Books-C2/Some-Book-Title-P2.html
```

Additionally, in the case that you have duplicate content, using static-looking URLs may actually exacerbate the problem. This is because whereas dynamic URLs make the parameter values and names obvious, rewritten static URLs obscure them. Search engines are known to, for example, attempt to drop a parameter it heuristically guesses is a session ID and eliminate duplicate content. If the session parameter were rewritten, a search engine would not be able to do this at all.

There are solutions to this problem. They typically involve removing any parameters that can be avoided, as well as excluding any of the remaining the duplicate content. These solutions are explored in depth in the chapter on duplicate content.

A Last Word of Caution

URLs are much more difficult to revise than titles and descriptions once a site is launched and indexed. Thus, when designing a new site, special care should be devoted to them. Changing URLs later requires one to redirect all of the old URLs to the new ones, which can be extremely tedious, and has the potential to influence rankings for the worse if done improperly and link equity is lost. Even the most trivial changes to URL structure should be accompanied by some redirects, and such changes should only be made when it is absolutely necessary.

This is relatively simple process. In short, you use the URL factory that we just created to create the new URLs based on the parameters in the old dynamic URLs. Then you employ what is called a "301-redirect" to the new URLs. The various types of redirects are discussed in the following chapter.

So, if you are retrofitting a web application that is powering a web site that is already indexed by search engines, you must redirect the old dynamic URLs to the new rewritten ones. This is especially important, because without doing this every page would have a duplicate and result in a large quantity of duplicate content. You can safely ignore this discussion, however, if you are designing a new web site.

Summary

We covered a lot of material here! We detailed how to employ static-looking URLs in a dynamic web site step-by-step. Such URLs are both search engine friendly and more enticing to the user. This can accomplished through several techniques, and you've tested the most popular of them in this chapter. A "URL factory" can be used to enforce consistency in URLs. It is important to realize, however, that URL rewriting is not a panacea for all dynamic site problems—in particular, duplicate content problems.

This article is excerpted from chapter 3 "Provocative SE-Friendly URLs" of the book *Professional Search Engine Optimization with ASP.NET: A Developer's Guide to SEO* *(Wrox, 2007, ISBN: 978-0-470-13147-3) by Cristian Darie and Jaimie Sirovich. Cristian Darie is a software engineer with experience in a wide range of modern technologies, and the author of numerous books and tutorials on AJAX, ASP.NET, PHP, SQL, and related areas. Cristian currently lives in Bucharest, Romania, studying distributed application architectures for his PhD. He's getting involved with various commercial and research projects, and when not planning to buy Google, he enjoys his bit of social life. If you want to say "Hi," you can reach Cristian through his personal web site at http://www.cristiandarie.ro. Jaimie Sirovich is a search engine marketing consultant. He works with his clients to build them powerful online presences. Officially Jaimie is a computer programmer, but he claims to enjoy marketing much more. He graduated from Stevens Institute of Technology with a BS in*

Computer Science. He worked under Barry Schwartz at RustyBrick, Inc., as lead programmer on all eCommerce projects until 2005. At present, Jaimie consults for several organizations and administrates the popular search engine marketing blog, SEOEgghead.com.