Beginning Web Development with Perl

From Novice to Professional

■ ■ ■

Steve Suehring

Apress®

Beginning Web Development with Perl: From Novice to Professional

Copyright © 2006 by Steve Suehring

ISBN (pbk): 1-59059-531-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: James Lee
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Managers: Laura Cheu, Richard Dal Porto
Copy Editors: Marilyn Smith, Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: Rebecca Plunkett
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

CHAPTER 8

Perl and RSS

RSS (an abbreviation for Rich Site Summary or RDF Site Summary) is used to syndicate content from a web site. RSS is helpful for gathering headlines and other news-related items from web sites or getting recent changes to a web page. It's common for an end user to use news aggregation software to consume RSS feeds. Web browsers such as Mozilla Firefox also enable RSS feeds to be used as bookmarks.

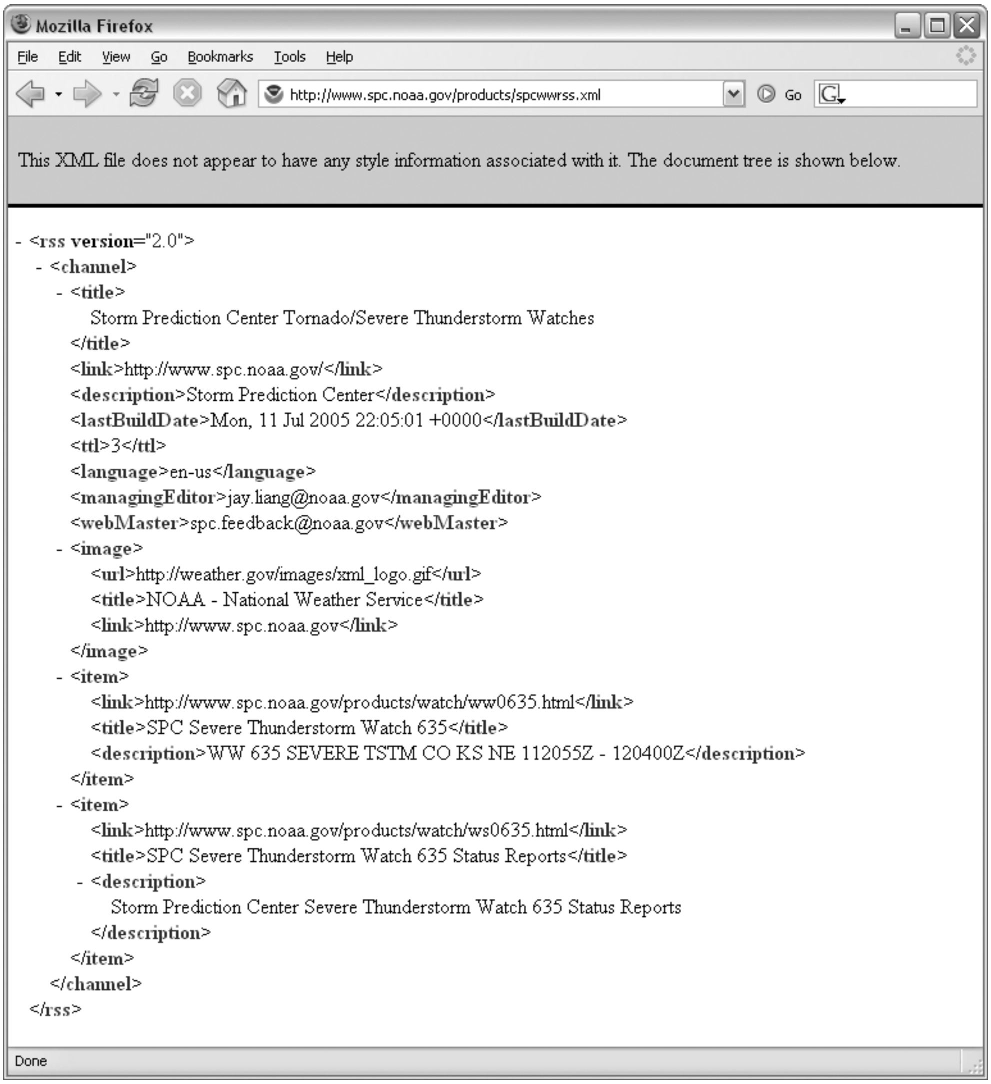
Various Perl modules handle RSS feeds. Some of the modules, such as `XML::RSS`, are general and designed to work with most any RSS feed; others are specific to a particular site's RSS feed. For example, `XML::RSS::Headline::PerlJobs` gets the headlines from `jobs.perl.org`, and `XML::RSS::Headline::Fark` gets headlines from the popular Fark web site.

This chapter looks at RSS from a Perl perspective. Specifically, you'll see how to consume and create RSS feeds using the `XML::RSS` module.

RSS: Versioning Fun

At the time of this writing, there are four versions of the RSS protocol: 0.90, 0.91, 1.0, and 2.0. Some aggregation software works with only certain versions of the protocol. The aggregators may support a limited subset of a newer version, or they may not support a newer version at all. Similarly, the Perl modules may or may not support every version of the RSS protocol. Some RSS modules handle the versions well, simply ignoring things that they don't implement, while others don't fail so gracefully. The best method for determining whether the module you're using works with a particular version of RSS is to read the documentation for that particular module.

For those not familiar with RSS, you can pull up an RSS feed through your web browser. You can point your browser at the example used throughout the chapter: `http://www.spc.noaa.gov/products/spcwwrss.xml`. You should be able to view it in a manner similar to Figure 8-1.


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.spc.noaa.gov/products/spcwwrss.xml
Go
Done

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <rss version="2.0">
  - <channel>
    - <title>
        Storm Prediction Center Tornado/Severe Thunderstorm Watches
      </title>
      <link>http://www.spc.noaa.gov/</link>
      <description>Storm Prediction Center</description>
      <lastBuildDate>Mon, 11 Jul 2005 22:05:01 +0000</lastBuildDate>
      <ttl>3</ttl>
      <language>en-us</language>
      <managingEditor>jay.liang@noaa.gov</managingEditor>
      <webMaster>spc.feedback@noaa.gov</webMaster>
    - <image>
        <url>http://weather.gov/images/xml_logo.gif</url>
        <title>NOAA - National Weather Service</title>
        <link>http://www.spc.noaa.gov</link>
      </image>
    - <item>
        <link>http://www.spc.noaa.gov/products/watch/ww0635.html</link>
        <title>SPC Severe Thunderstorm Watch 635</title>
        <description>WW 635 SEVERE TSTM CO KS NE 112055Z - 120400Z</description>
      </item>
    - <item>
        <link>http://www.spc.noaa.gov/products/watch/ws0635.html</link>
        <title>SPC Severe Thunderstorm Watch 635 Status Reports</title>
      - <description>
          Storm Prediction Center Severe Thunderstorm Watch 635 Status Reports
        </description>
      </item>
    </channel>
  </rss>
```

Figure 8-1. *An RSS feed as seen through a web browser*

Reading RSS with XML::RSS

The previous chapter described how to consume a SOAP-based web service from the United States National Weather Service. The National Weather Service has a division called the Storm Prediction Center (SPC), which handles the forecasting of severe or extreme weather events for the United States. The SPC home page is `http://www.spc.noaa.gov/`. Among the many products offered for current conditions and forecasting is an RSS feed of mesoscale discussions, convective outlooks, and watches. Here, you'll see how to consume the RSS feed for weather watches offered by the National Weather Service.

You'll use the XML::RSS module for reading an RSS feed. XML::RSS is available within many Linux distributions or from your favorite CPAN mirror. XML::RSS includes methods to both parse (read) and create (write) RSS feeds.

Parsing RSS Feeds

This section looks specifically at parsing an RSS feed for interesting items. While XML::RSS can work with RSS feeds, it does not include methods to retrieve the actual RSS from the Internet. For this functionality, you can turn to LWP::Simple, which was covered in Chapter 5.

Parsing an RSS feed can be broken into three basic steps:

1. Get the RSS.
2. Parse the RSS.
3. Do something with the RSS.

To accomplish the first task, LWP::Simple will retrieve the RSS feed and place it into a variable. XML::RSS can perform the second task. The third task is accomplished by you, doing whatever it is that you would like to do with the program, assisted by XML::RSS methods.

Here are the beginning bits of code for the program to be built in this section:

```
use strict;
use XML::RSS;
use LWP::Simple;
```

These lines of code import the modules into the program's namespace and also enable the strict pragma.

The URL for the SPC's Weather Watch RSS feed is http://www.spc.noaa.gov/products/spcwwrss.xml. Therefore, you can retrieve it by using the get() method of LWP::Simple, as shown here:

```
my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");
```

The data from the RSS feed is saved into the $url variable for later use. If you would like to debug to ensure that the RSS was actually retrieved, you can use a simple print statement:

```
print $url;
```

For example, here's the output from that print statement:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>Storm Prediction Center Tornado/Severe Thunderstorm Watches</title>
    <link>http://www.spc.noaa.gov/</link>
    <description>Storm Prediction Center</description>
    <lastBuildDate>Mon, 11 Jul 2005 22:01:02 +0000</lastBuildDate>
    <ttl>3</ttl>
    <language>en-us</language>
    <managingEditor>jay.liang@noaa.gov</managingEditor>
    <webMaster>spc.feedback@noaa.gov</webMaster>
    <image>
      <url>http://weather.gov/images/xml_logo.gif</url>
      <title>NOAA - National Weather Service</title>
      <link>http://www.spc.noaa.gov</link>
    </image>

    <item>
      <link>http://www.spc.noaa.gov/products/watch/ww0635.html</link>
      <title>SPC Severe Thunderstorm Watch 635</title>
      <description>WW 635 SEVERE TSTM CO KS NE 112055Z - 120400Z</description>
    </item>
    <item>
      <link>http://www.spc.noaa.gov/products/watch/ws0635.html</link>
      <title>SPC Severe Thunderstorm Watch 635 Status Reports</title>
      <description>Storm Prediction Center Severe Thunderstorm Watch 635 Status
          Reports</description>
    </item>
  </channel>
</rss>
```

■**Note** Unless all you want to do is print the RSS to STDOUT, you would obviously use the `print` statement only for debugging. Otherwise, comment it out or remove it entirely.

The first order of business is to instantiate an RSS object, which I'll call `$rss` in this code. Next, you parse the RSS that was retrieved by `LWP::Simple` using the `parse()` method of `XML::RSS`.

```
my $rss = XML::RSS->new;
$rss->parse($url);
```

With the RSS parsed, it's now a matter of gleaning from the RSS whatever information is useful for your program using one of the `XML::RSS` methods, iterating through a hash of items, and so on. For example, you might use the `channel()` method, which enables you to print the title, the link the description, and other information about the RSS feed:

```
print $rss->channel('title'), "\n";
```

Each item within the RSS feed is placed into an array and can be referenced by iterating through the array, as shown in this example:

```
foreach my $item (@{$rss->{'items'}}) {
        print "Title: $item->{'title'}\n";
        print "Desc: $item->{'description'}\n";
        print "Link: $item->{'link'}\n";
}
```

In the example, each item is placed into the scalar variable `$item`, and then attributes of this scalar variable are called, as you see by the `print` statements to output the title, description, and link. The entire program is shown in Listing 8-1.

Listing 8-1. *An Initial RSS Example*

```
#!/usr/bin/perl

use strict;
use XML::RSS;
use LWP::Simple;

my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");

# Debugging - Comment out when not debugging #
#print $url;

my $rss = XML::RSS->new;
$rss->parse($url);

print $rss->channel('title'), "\n";
foreach my $item (@{$rss->{'items'}}) {
        print "Title: $item->{'title'}\n";
        print "Desc: $item->{'description'}\n";
        print "Link: $item->{'link'}\n";
}
```

When executed, the program produces this output:

```
Storm Prediction Center Tornado/Severe Thunderstorm Watches
Title: SPC Severe Thunderstorm Watch 587
Desc: WW 587 SEVERE TSTM KS NE 031245Z - 031800Z
Link: http://www.spc.noaa.gov/products/watch/ww0587.html
Title: SPC Severe Thunderstorm Watch 587 Status Reports
Desc: Storm Prediction Center Severe Thunderstorm Watch 587 Status Reports
Link: http://www.spc.noaa.gov/products/watch/ws0587.html
```

Be aware that some sites monitor the number of RSS retrievals performed from a specific IP address within a certain time period. This is because of abuse by some people who retrieve the RSS looking for updates too frequently. For example, some people have been known to configure their RSS aggregator to request the RSS feed every few seconds. While one user doing this wouldn't likely cause a performance degradation, if 1,000 users requested constant updates, that would quickly lead to a distributed denial of service. Therefore, be careful when debugging the script that you don't request the RSS feed too many times and get blocked in the process! In the next section, you'll see a method for developing your script so that it doesn't cause the site operator to lose sleep.

Debugging RSS Scripts

While you're developing a script to retrieve an RSS feed, it's not uncommon to run that script multiple times within a short period. As I just mentioned, this can, on certain sites, cause your IP address to get blocked for abusive RSS requests. The site operator who owns the RSS feed likely won't be able to tell the difference between someone debugging a script and someone requesting constant RSS updates.

Here's how I solve the problem: When developing the script, I use `LWP::Simple` to retrieve the RSS and print the resulting RSS to STDOUT using the `print` statement shown in the preceding section. Then I save the output to a file by using a simple shell redirect.

For example, assume you created a script to retrieve an RSS feed called `get_rss.pl`. That script looks like this:

```
#!/usr/bin/perl

use strict;
use XML::RSS;
use LWP::Simple;

my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");

print $url;
```

You run the script, and the output is printed to STDOUT. Redirect that output to a file:

```
./get_rss.pl > spc_rssfeed.xml
```

With the RSS contained in the file, you'll be able to use the `XML::RSS parsefile()` method to parse the RSS feed, rather than requesting the RSS from the site again. Recall the example shown in the previous section. Instead of using the `get()` method, you comment that out and use the `parsefile()` method, along with an argument of the filename containing the RSS feed, as shown in Listing 8-2.

Listing 8-2. *Debugging an RSS Feed Script*

```
#!/usr/bin/perl

use strict;
use XML::RSS;
use LWP::Simple;

#my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");

# Debugging - Comment out when not debugging #
#print $url;

my $rss = XML::RSS->new;
$rss->parsefile("spc_rssfeed.xml");
```

```
print $rss->channel('title'), "\n";
foreach my $item (@{$rss->{'items'}}) {
        print "Title: $item->{'title'}\n";
        print "Desc: $item->{'description'}\n";
        print "Link: $item->{'link'}\n";
}
```

Once the script has been developed and debugged, you uncomment the `get()` method and change the `parsefile()` method to the `parse()` method. This extra bit of work makes the site operator of the RSS feed happy and can prevent you from getting blocked.

Writing RSS with XML::RSS

Along with parsing an RSS feed, `XML::RSS` can also write RSS. Some of the same methods used for parsing an RSS feed with `XML::RSS` are used in creating one, and others are simply reversed; instead of reading with the methods, you write with them.

As an example, let's see how to build an RSS feed using repackaged data from the National Weather Service's Weather Watch RSS. I live in Wisconsin. Therefore, I'm interested in weather events in and around the state of Wisconsin. It would be nice to be able to ignore watches for other states and produce an RSS file that contains only items relevant to my area. (Fortunately, there are no weather watches for my area on the day that I'm writing this chapter. Therefore, I'll also be checking in Kansas, since I know there's a weather watch there, and I'll also include the states of Minnesota, and Iowa.)

When creating an RSS feed with `XML::RSS`, you can set the RSS version. If not specified, the default version is 1.0. Since this program will repackage data from the SPC's RSS feed, the opening bits of the program are much the same as the previous examples:

```
use strict;
use XML::RSS;
use LWP::Simple;
my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");
my $rss = XML::RSS->new;
$rss->parse($url);
```

The existing `XML::RSS` object, `$rss`, is used to parse the incoming RSS feed. Therefore, you need a new `XML::RSS` object to create the new feed. I'll call this new object `$rsswriter`:

```
my $rsswriter = XML::RSS->new;
```

Since no version is specified, version 1.0 will be used. However, if you wanted to specify the version, the statement would look like this for a version 0.91 feed:

```
my $rsswriter = XML::RSS->new(version => '0.91');
```

Instead of reading the channel information as in the previous example, this time, you're creating your own RSS channel.

```
$rsswriter->channel(
    title => "My Watch Summary",
    link => "http://www.braingia.org/",
    description => "Weather Watches for KS, IA, MN, and WI"
);
```

As in the previous examples, you iterate over each item of the incoming RSS feed. Instead of printing all of the items to STDOUT, this time, each one is examined to see if the description contains one of the four states that you're interested in for this example. If one of those states is listed within the incoming item's description, the `add_item()` method is called on the `$rsswriter` object:

```
foreach my $item (@{$rss->{'items'}}) {
    if ($item->{'description'} =~ /KS|WI|IA|MN/) {
        $rsswriter->add_item(
            title => $item->{'title'},
            description => $item->{'description'},
            link => $item->{'link'}
        );
    }
}
```

Once each item in the incoming feed has been examined, it's time to write the RSS feed. You can do this by using the `save()` method or by printing the feed with the `as_string()` method. I chose to save the RSS feed to a file called `mywatchsummary.xml`:

```
$rsswriter->save("mywatchsummary.xml");
```

If you would rather print the RSS to STDOUT, use a `print` statement with the `as_string()` method:

```
print $rsswriter->as_string;
```

Regardless of which method you use, the resulting file or output looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns="http://purl.org/rss/1.0/"
 xmlns:taxo="http://purl.org/rss/1.0/modules/taxonomy/"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:syn="http://purl.org/rss/1.0/modules/syndication/"
 xmlns:admin="http://webns.net/mvcb/"
>

<channel rdf:about="http://www.braingia.org/">
<title>Watch Summary for Western Great Lakes</title>
<link>http://www.braingia.org/</link>
```

```
<description>Weather Watches for IA, MN, and WI</description>
<items>
 <rdf:Seq>
  <rdf:li rdf:resource="http://www.spc.noaa.gov/products/watch/ww0587.html" />
 </rdf:Seq>
</items>
</channel>

<item rdf:about="http://www.spc.noaa.gov/products/watch/ww0587.html">
<title>SPC Severe Thunderstorm Watch 587</title>
<link>http://www.spc.noaa.gov/products/watch/ww0587.html</link>
<description>WW 587 SEVERE TSTM KS NE 031245Z - 031800Z</description>
</item>

</rdf:RDF>
```

If you wanted to, you could also use the `output` attribute to convert between RSS versions. For example, the previously shown output is version 1.0. However, using the `output` attribute of the `XML::RSS` object, `$rsswriter`, you can change this to a different version on the fly. For example, the code to change the version just prior to printing the output looks like this:

```
$rsswriter->{'output'} = '0.91';
print $rsswriter->as_string;
```

The resulting output would show the change:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
            "http://my.netscape.com/publish/formats/rss-0.91.dtd">

<rss version="0.91">

<channel>
<title>My Watch Summary</title>
<link>http://www.braingia.org/</link>
<description>Weather Watches for KS, IA, MN, and WI</description>

<item>
<title>SPC Severe Thunderstorm Watch 587</title>
<link>http://www.spc.noaa.gov/products/watch/ww0587.html</link>
<description>WW 587 SEVERE TSTM KS NE 031245Z - 031800Z</description>
</item>

</channel>
```

The final program is shown in Listing 8-3.

Listing 8-3. *Retrieving Weather Watches with RSS*

```
#!/usr/bin/perl

use strict;
use XML::RSS;
use LWP::Simple;

my $url = get("http://www.spc.noaa.gov/products/spcwwrss.xml");

my $rss = XML::RSS->new;
$rss->parse($url);

my $rsswriter = XML::RSS->new;

$rsswriter->channel(
    title => "My Watch Summary",
    link => "http://www.braingia.org/",
    description => "Weather Watches for KS, IA, MN, and WI"
);

foreach my $item (@{$rss->{'items'}}) {
    if ($item->{'description'} =~ /KS|WI|IA|MN/) {
        $rsswriter->add_item(
            title => $item->{'title'},
            description => $item->{'description'},
            link => $item->{'link'}
        );
    }
}

$rsswriter->save("mywatchsummary.xml");
```

The `XML:RSS` module contains other methods and attributes that you may find helpful for your own RSS writing projects. Look over the `perldoc` for `XML::RSS` for more information about these methods and attributes.

Security Considerations with RSS

Creation of your own RSS feeds doesn't pose any great security risk in itself. Of course, you don't want to release any sensitive information through an RSS feed, just as you wouldn't want to allow access to certain data through a web page or CGI program.

Consuming an RSS feed carries with it the same risks inherent in any external data source. You should be sure that all data external to your program is safe for use within the program. If an RSS feed contains malicious data, using it within your program puts the program and the system at risk.

Summary

This chapter dealt with RSS feeds through Perl, covering both creation and consumption of RSS feeds. Specifically, you saw how to parse and write RSS using the `XML::RSS` module. Other modules for parsing and writing RSS with Perl, such as `XML::RSS::Feed`, are available.

When parsing an RSS feed, you create a new RSS object to parse an RSS file. Retrieval of the RSS file is left for another module. In this chapter, you saw how to use `LWP::Simple` to retrieve an RSS feed from an Internet site, but you can use any means to get an RSS feed into the parser, including using a local file. A local file is recommended when developing and debugging the RSS feed, so that the site operator doesn't misinterpret the repeated retrieval requests.

Many of the same methods are used for both parsing and writing an RSS feed. You can choose and change the version for writing RSS by specifying it at instantiation time or with the `output` attribute.

This and the previous chapter have both touched on XML-related services in one form or another and provided a good introduction into XML applications in the real world. In the next chapter, you'll finally look at straight XML parsing with Perl.