16 Chapter

RSS and XML

16.1 XML and RSS Overview

The Extensible Markup Language, commonly called XML, is a markup language for structured documents. A structured document is one that contains various elements, such as images and text, as well as an indication of what that element is. For example, a letter has elements such as address, body, and footer that all have different roles in the document. When there is some indicator on the letter as to what each part of each element does, then there is a structured document. Most documents do have structure, and XML can be used to define that structure in a usable, and standard, way. HTML documents are structured documents because they have elements and tags defining the role of that element.

HTML and XML are not the same, however. HTML has a known set of tags. XML, on the other hand, does not. XML isn't just a way to mark up documents; it is really a meta-language giving developers a way to describe markups. The developers of the XML documents and applications that use them define the tags for the document as well as the relationship between those tags. XML was created to allow developers to use their tags to create structured documents for the Web. Through this chapter you will learn some basics of an XML document and see how to use an XML variant, RSS, for the Web.

16.1.1 Structure of an XML Document

The look of an XML document is very close to that of HTML and should be easy to follow and understand. In Listing 16-1 you will see a simple XML document.

Listing 16-1 Example XML document

```
<?xml version="1.0"">
<Zappa>
<quote>Good night Cleveland, wherever you are!</quote>
<quote>Shoot low, they're riding Shetlands.</quote>
</Zappa>
```

Well, there is nothing too mystical looking in there. The first line declares that this is an XML document as well as the version of XML being used. This line is not obligatory, but it is a good practice to have it and it helps make the document well formed. Next a container is created. The name of the container is “Zappa”. Inside of this container, there are two elements that are being tagged as “quote”. Finally, the “Zappa” container is closed, containing two quote elements. The preceding example is a very simple one, and XML goes much deeper than what you see there, but it would take a book to show all the aspects (in fact, there are books that do). However, the preceding example shows enough to help you understand the coming examples and quickly get started using XML and its variants.

16.2 News Portals with RSS

A few short years ago, Netscape created what could be called the first Web portal. They developed the My News Network,[1] or MNN, which gave a facility for its users to get much of the news and search capabilities from their own starting page. Users can choose sites from which they would like to see news summaries and have them displayed. Netscape dubbed these summaries “channels,” and that is now the common name. For Netscape channels, the backend server would periodically fetch structured XML documents from contributing sites and update the content of the channels. In order to ensure that all contributors' files were structured in the same way, Netscape developed the RDF Site Summary (RSS) format. This format uses XML and Resource Description Format (RDF)[2] to define a markup language for developers to use. The RSS format is not just for Netscape channels anymore. Using the XML::RSS module a programmer can use the same channel files to format data for the Web.

1. *http://my.netscape.com*
2. This is a format used to describe Web-based meta-data.

Before we jump into using XML::RSS, let's first cover the RSS format markup language. There is a finite set of tags to use with RSS. The main container for the document is the channel. Within this containter there are a few elements that can be used to define the content. Three main elements are title, link, and description. The channel container can also contain image, textinput, and other containers. Let's take a moment to break down an RSS document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
            "http://my.netscape.com/publish/formats/rss-0.91.dtd">
<rss version="0.91">
```

This section is doing three declarations. The first is declaring that this is an XML document, as you saw in the previous section. The second declaration is the DOCTYPE for the document. The third is declaring that this is a RSS document, using version 0.91. Version 0.90 of RSS was introduced by Netscape in 1999, so this is all still very new. Now that we have the type of document we will be creating, the next step is to create it.

```
<channel>
```

This line opens the *channel* container. Everything up until this containter is closed will be a part of this container. The main container in all RSS documents is *channel*.

```
<title>My News</title>
<link>http://news.me.com</link>
<description>My news, for you!</description>
<language>en</language>
<copyright>Copyright 2000++, Me</copyright>
<pubDate>Sun May 21 15:43:45 2000</pubDate>
<lastBuildDate>Sun May 21 15:43:45 2000</lastBuildDate>
<managingEditor>me@me.com</managingEditor>
<webMaster>me@me.com</webMaster>
```

Here a list of elements is being defined. The three main ones, which are also required, are title, which is the title of the channel; link, which is the location of the Web site for this channel; and description, which is how to describe the channel. The remaining elements listed are optional for inclusion in the document but can provide useful information. We will not go into individual descriptions of them, since the name of each element does a good job of that on its own.

```
<image>
<title>My News</title>
<url>http://news.me.com/my_news.gif</url>
<link>http://news.me.com</link>
<width>119</width>
<height>30</height>
</image>
```

The preceding snippet shows the optional *image* container. This container will hold information about the image logo for the channel. Again, the simplicity of the RSS format makes each element name self-explanatory for what the element data is used for. The last line of the snippet closes the container.

```
<item>
<title>Man eats cheese, MPEG at 11.</title>
<link>http://news.me.com/news/story2.html</link>
</item>
```

The *item* container is extremely important, since it defines the content of the channel. This container is required, and there can be multiple *item* containers per channel. For example, if there were five news stories, there would be five *item* containers. There are two elements in an *item* container, and both are required. These elements define the title of the item as well as the link to the full story.

```
<textinput>
<title>Search My News</title>
<description>Search the Archives</description>
<name>text</name>
<link>http://news.me.com/search.cgi</link>
</textinput>
```

The “textinput” containter will create a text box that people can use to search your site. This container is optional, since some sites may not have search capabilities. If the site does have search capabilities, then this is an excellent way to allow users to search the site from wherever the channel is displayed.

```
</channel>
</rss>
```

The document ends by closing the *channel* container and closing the RSS document. This is equivalent to closing an HTML document with

</HTML>. That's the entire RSS document. With this document, a Netscape channel can be created[3] and you can share it with others around the world who want to customize their Web pages and applications with your information. Now that you know the structure of an RSS document, we will show you how to use that file and then create your own using Perl.

16.2.1 A Home Page News Portal

Now that you have a general understanding what an RSS document is and how one in structured, it is time to see how a developer can use these files to create dynamic content and create a customized portal with them. The easiest way to use RSS formatted files in Perl is with the XML::RSS module, written by Jonathan Eisenzopf. This module allows a programmer to easily get the data that is inside an RSS formatted file in an object-oriented fashion. In this section we will walk through an application that fetches RSS files, displays the channels to a Web page, and allows a user to add channels to fetch and choose what channels are displayed on the Web page.

The first thing to know is where to get channel files to use! The xmlTree Web site[4] is attempting to categorize much of the XML content on the Web, and much of this content is RSS files. Using this Web site, you can search for the type of channel you wish you have and find the location of the RSS files[5] for those channels. Now that you know where to look for channel files, let's get on with the script.

We begin by creating a table that will hold information about the channels. We will be keeping information on three things: the URL of the RSS file, the name of the channel, and whether the channel is to be viewed on the Web page.

Listing 16-2 SQL to create the RDF table

```
CREATE TABLE rdf (
 URL varchar(250) NOT NULL,
 Name varchar(250) NOT NULL,
 Selected int(11)
);
```

To help you use this script as you read along, Listing 16-2 shows the insert statements of four channels. This will also give you an idea of what the data in the table looks like.

3. The RDF file needs to be registered with Netscape so they know where to fetch it.
4. *www.xmltree.com*
5. Many of these have an .rdf extension, although this is not required.

```
INSERT INTO rdf VALUES ('http://slashdot.org/slashdot.rdf',
                        'Slashdot',0);
INSERT INTO rdf VALUES ('http://www.news.perl.org/perl-news-
                        short.rdf','Perl News',1);
INSERT INTO rdf VALUES ('http://freshmeat.net/backend/fm.rdf',
                        'Freshmeat',1);
INSERT INTO rdf VALUES ('http://www.securityfocus.com/topnews-
                        rss.html','Security Focus',1);
```

The first thing that is needed is a way to get the RSS files locally so the XML::RSS module can parse it. To accomplish this, we create the script fetch, which can be run from the command line or via cron at regular intervals.

```
01: #!/usr/bin/perl -w
02: # fetch
03: use strict;
04: use File::Basename;
05: use DBI;
06: use LWP::Simple qw(mirror);
```

Lines 1–6 define the path to Perl and *use()* the needed modules. The *mirror()* method from LWP::Simple will be used to get the remote RSS files. When the file will be saved locally, they will be saved with the same name of the remote file. The *basename()* method from File::Basename will be used to easily get that information for us.

```
07: my $RDF_DIR = './rdf';
```

Line 7 initializes the $RDF_DIR variable. Its value will be used as the directory to store the retrieved RSS files.

```
08: my $dbh = DBI->connect("dbi:mysql:book", "user", "password");
09: my $sth = $dbh->prepare(qq{select URL from rdf});
10: $sth->execute or die $DBI::errstr;
```

Line 8 connects to the database. **Line 9** then prepares a query that selects all the URLs for the RSS files from the table. **Line 10** executes this statement or dies with the error from DBI.pm.

```
11: while (my $url = $sth->fetchrow) {
12:     my $name = basename($url);
13:     mirror($url, "./$RDF_DIR/$name");
14: }
```

Lines 11–14 loop through the results set from the database. The value returned and stored in $url will be a single URL. In **line 12** $url is passed to the *basename()* method, which will return the filename from the end of the URL. This value is stored in $name. **Line 13** does the real work. The *mirror()* method takes the URL as its first argument and in turn fetches the remote Web page. The second argument is the location of where the new file is to be saved. When this loop is done, all the available RSS files will be stored locally. This script would be most helpful if run in intervals to make sure the latest RSS files are local.

```
15: $dbh->disconnect;
```

Line 15 closes the database connect.

The next part of the application is to take the RSS files and make them useful. The index.cgi script, to be explained next, will create channel boxes on a Web page displaying the data from the RSS files.

```
01: #!/usr/bin/perl -wT
02: # index.cgi
03: use strict;
04: use CGI qw(:standard end_ul end_table);
05: use CGI::Carp qw(fatalsToBrowser);
06: use File::Basename;
07: use DBI;
08: use XML::RSS;
09: my $RDF_DIR = './rdf';
10: my $dbh = DBI->connect("dbi:mysql:book", 'user','password') or
                           print $DBI::errstr;
11: my $sth = $dbh->prepare(qq{select URL from rdf where
                           Selected = 1});
12: $sth->execute;
```

Lines 1–12 only introduce one new thing: using XML::RSS. This is the module that will be used to retrieve the wanted information from the RSS files in $RDF_DIR. We are also selecting all the URLs from the database where Selected is 1, which indicates that channel should be displayed on the Web page.

```
13: print header,
14:       start_html("My Home Page"),
15:       h2("My Favorite Sites");
16: print start_table({cellpadding=>0, cellspacing=>0, border=> 0,
                  width => '100%'}),
17:       td;
```

Lines 13–17 start off the HTML for the page. All the HTML is being printed out using methods from the CGI.pm module.

```
18: my $count = 1;
19: my @html = ('</TD><TD>', '</TD><TR><TD>');
```

Line 18 and 19 initialize two variables that will be used together as a sort of toggle. The channels will be displayed in two columns, and one way or another we have to know if a <TR> is to be printed to start a new row. Since every other column will have the <TR>, a simple little toggle can be used to switch between the two HTMLs.

```
20: while (my $url = basename($sth->fetchrow)) {
21:      my $rss = new XML::RSS;
```

Line 20 starts iterating through the results set, which will be URLs. As the results row is fetched, it is also put through the *basename()* method to get only the filename. For example, the URL *http://slasdhot.org/slashdot.rdf* will be reduced to "slashdot.rdf." That filename is then stored in $url. **Line 21** then creates a new XML::RSS object. The resulting $rss variable will be an object reference.

```
22:      eval {$rss->parsefile("$RDF_DIR/$url")};
23:      warn "$url will not parse $@" and next if $@;
```

Line 22 evaluates the *parsefile()* method. The *parsefile()* method takes the location of the RSS file on disk, opens it, and parses it. This is being wrapped in an *eval()* because if the RSS is broken, an exception may be thrown. By using *eval(),* the exception can be caught, and the script will continue. **Line 23** will print a warning to STDERR and move on to the next iteration of the loop if an exception is caught.

```
24: my $last_mod = scalar localtime((stat("$RDF_DIR/$url"))[9]);
```

Line 24 initializes $last_mod with the scalar value of the last modified time of the RSS file.

```
25:     print start_table({cellpadding=>0, cellspacing=>2,
                          border=> 5, width=>'75%'}),
26:     td({valign=>'CENTER', bgcolor => '#C0C0C0'});
```

Lines 25 and 26 begin the HTML table for the channel.

```
27:    $rss->{image}{url}
28:    ? print img({src=>$rss->{image}{url}})
29:    : print strong($rss->{channel}{title});
```

Lines 27, 28, and 29 are one line broken up for clarity. **Line 27** wants to see if there is a true value for $rss->{image}{url}. If there is, then the RSS file has an *image* container. If it is true, **line 28** then displays that image to the browser. If there is no *image* container, the title for the channel is displayed instead.

```
30: print ul;
```

Line 30 prints the <UL> tag. The items in the RSS file will be displayed as an unordered list.

```
31: for (@{$rss->{items}}) {
32:         print li(a({href=>$_->{link}}, $_->{title}));
33: }
```

Lines 31–33 iterate over the items in the channel container. $rss->{items} is a reference to an array and is being dereferences as such. **Line 32** prints out the item to the browser. The item is displayed as a hyperlink to the URL for the story. The text for the hyperlink is the items title. These values come from the link and title elements of the item container.

```
34: print end_ul;
```

Line 34 prints the </UL> tag.

```
35: if ($rss->{textinput}{link}) {
36:         print $rss->{textinput}{description},
                start_form(-method => 'GET',
37:                        -action => $rss->{textinput}{link}),
38:             textfield(-name => $rss->{textinput}{name}),
39:             end_form;
40: }
```

Lines 35–40 handle an occurrence of a textinput container. If **line 35** finds that the *link* element of the textinput container has a true value, the rest of the block prints the appropriate form.

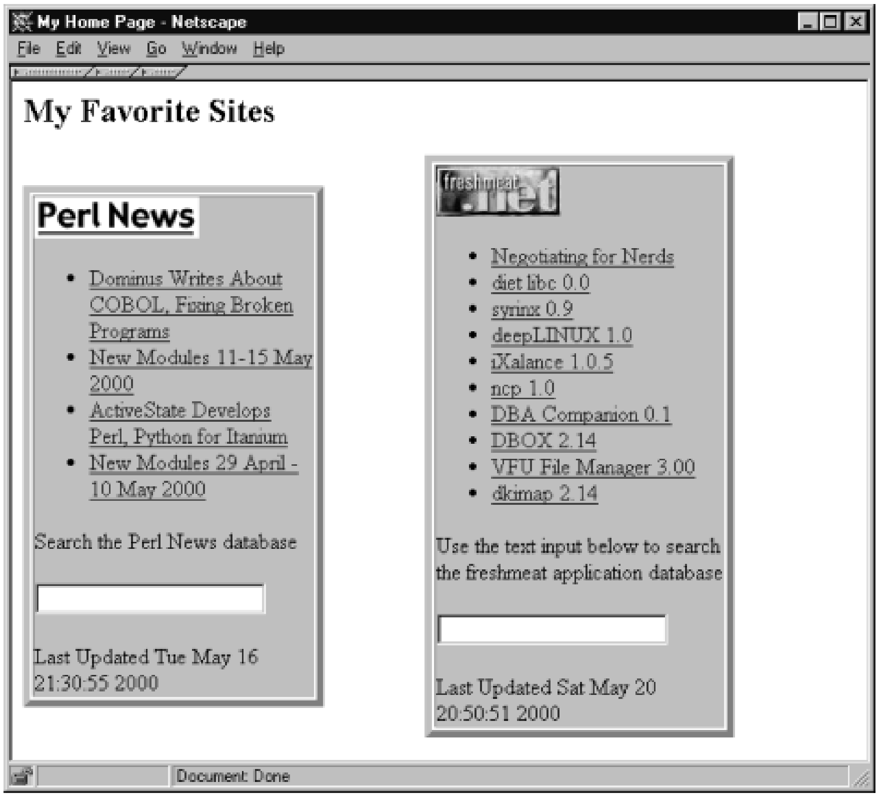
```
41:     print qq(Last Updated $last_mod<BR>),
42:        end_table;
43:     $html[$count^=1];
44: }
```

Line 41 displays the last modified date that was retrieved in line 24. **Line 42** closes the table for the channel, but **line 43** does something fun. Remember lines 18 and 19 when we initialized @html and $count so they would act as a toggle? Line 43 is that toggle. We want to have two columns of channels being displayed, and by XORing the values of $count and 1, we can print the desired HTML to either end the table row with a <TR> tag or not. The higher the value to initialize $count with, the more columns you will have. Finally, **line 44** ends the *while()* loop.

```
45: print end_table,
46:       end_html;
```

Lines 45 and 46 end the script by printing the closing tags for the main table and the HTML. When this script is run, the Web page that is generated is similar to that shown in Figure 16-1.

Figure 16-1 Channels on a Web page



My Favorite Sites

Perl News

- Dominus Writes About COBOL, Fixing Broken Programs
- New Modules 11-15 May 2000
- ActiveState Develops Perl, Python for Itanium
- New Modules 29 April - 10 May 2000

Search the Perl News database

Last Updated Tue May 16 21:30:55 2000

freshmeat.net

- Negotiating for Nerds
- diet libc 0.0
- syrinx 0.9
- deepLINUX 1.0
- iXalance 1.0.5
- ncp 1.0
- DBA Companion 0.1
- DBOX 2.14
- VFU File Manager 3.00
- dkimap 2.14

Use the text input below to search the freshmeat application database

Last Updated Sat May 20 20:50:51 2000

To this point, the application has a means to fetch updated RSS files and display the channel data to a Web page. However, the application still needs a way to add RSS files to the database, as well as choose which ones to display and which ones not to. To accomplish this, we will show the script admin.cgi.

```
01: #!/usr/bin/perl -wT
02: # admin.cgi
03: use strict;
04: use CGI qw(:standard);
05: use CGI::Carp qw(fatalsToBrowser);
06: use DBI;
07: my $dbh = DBI->connect("dbi:mysql:book", "user", "password");
08: param('Submit') ? add_new() : show_form();
09: $dbh->disconnect;
```

Lines 1–9 introduce nothing new. Line 8 is what is really important here. **Line 8** checks if there was a "Submit" parameter sent to the script. If there was, this means that the form to make changes was submitted and to call the *add_new()* subroutine. The *add_new()* subroutine will make the needed changes to the database, then reshow the form with the *show_form()* subroutine. If there was no "Submit" parameter sent to the script, the *show_form()* subroutine is immediately called. **Line 9** disconnects from the database.

```
10: sub show_form {
11:    my $sth = $dbh->prepare(qq{select * from rdf});
12:    $sth->execute;
13:    print header,
14:          start_html("My Home Page Options"),
15:          h2("Choose My Favorite Sites");
16:    print start_form(-method => 'POST', -action => 'admin.cgi');
```

Lines 9–16 begin the *show_form()* subroutine. This subroutine will do one thing: display the Web form to the browser. The form will consist of a list of all the channels in the database, each with a checkbox denoting whether it has been selected to be viewable on the main Web page. The page also will have a place to add a new channel by entering the channels' name, URL to the RSS file, and a checkbox to select if the channel should be shown on the main Web page. Figure 16-2 shows what the final form will look like. **Line 11** is the query that will select all the information from the database. This data will be used starting in **line 17** to display the channels. The other line of note is **line 16**, which begins the Web form with itself as the script to which the form will be submitted.

```
17:    while (my $data = $sth->fetchrow_hashref) {
18:         my $checked = $data->{Selected} ? "CHECKED" : "";
19:         print checkbox(-name => 'Selected',
20:                        -checked => $checked,
21:                        -value => $data->{Name},
22:                        -label => $data->{Name},
23:                        ),
24:                p;
25:    }
```

Lines 17–25 are a loop that iterates over the data set returned from the SQL executed on line 12. The data returned is stored in the $data variable, which is a hash reference containing the data for the specific row being returned from the *fetchrow_hashref()* method. Each channel in the database has a Selected field, which denotes if the channel is to be displayed on the main Web page. This field will either contain a 0 or 1 to not display or display the channel. **Line 18** checks to see if the channel is selected or not to determine if the CHECKED attribute of the HTML checkbox input tag should be shown. If $data->{Selected} is a true value, the $checked variable is initialized as the string "CHECKED" and is an empty string if there is a false value. **Lines 19–23** display the checkbox for the channel. Passing parameters to the *checkbox()* method from CGI.pm, we can define the checkbox's name, which will be "Selected," whether or not the checkbox is to be displayed selected, as well as the value and label for the checkbox. The value and label parameters are both defined as the name of the channel, referred to as $data->{Name}. This continues for each of the channels in the database. **Line 25** closes the *while()* block we have been in.

```
26:    print h2("Add new channel") , p,
27:        "RDFs URL: ", textfield(-name => 'URL', -size => 50), p
28:        "Name of channel: ", textfield(-name => 'Name',
                                         -size => 50), p
29:        "Display on Home Page: ",
30:         checkbox(-name => 'new-Selected', -label => ''), p,
31:         submit(-name => 'Submit', -value => 'Make Changes'),
32:         end_form, end_html;
33: }
```

Lines 26–32 are a long *print()* to display three form elements that will allow a user to enter a new channel into the database. The first form element, aptly named "URL," is a text box (created with the *textfield()* method) that will be used to gather the URL of the RSS file. **Line 28** creates the second text box, “Name,” which is where the user-defined name of the channel is entered. **Line 30** is a checkbox, named “new-Slected,” where the user can select if the channel is to be displayed or not. You may have noticed that all the form elements except this one is the name of a field in the database. This is to make sure that when the form is submitted, this value doesn’t want to get mixed in with the other checkboxes that were selected. The other selected checkboxes will be seen as an array when they are retrieved in the *show_form()* subroutine, and this value will be separate. **Line 31** prints the submit button for the form. A value is defined for the button so the value will have a true value when line 8 checks to see if this form is being submitted for an update or being called directly. **Line 32** ends the *print()* by ending the form and Web page.

```
34: sub add_new {
35:     my $qry_select = qq(update rdf set Selected = 1
                             where );
36:     my $qry_deselect = qq(update rdf set Selected = 0
                             where );
```

Line 34 begins the *add_new()* subroutine. **Lines 35 and 36** define two variables that are the beginnings of SQL UPDATE statements. When the data is parsed to see what channels were selected, there is no real way to know which ones are *not* selected. This means that after knowing the selected channels, two queries will be executed. One will update the Selected field of those selected to 1, and those that are not selected will have it set to 0. These queries will be built in lines 39 and 40.

```
37:     my @selected = param('Selected');
```

Line 37 builds the @selected array by getting all the parameters submitted with the name “Selected.” The elements of the @selected array will be the names of the channels the user checked to be viewable from the form.

```
38:     $qry_select .= qq(Name = '$_' or ) for @selected;
39:     $qry_deselect .= qq(Name <> '$_' and ) for @selected;
```

Lines 38 and 39 build the update statements. Both scalars are concatenated as *for()* iterates through @selected.

```
40:     $qry_deselect =~ s! and $!!;
41:     $qry_select =~ s! or $!!;
```

Lines 40 and 41 clean up the queries being made. At the end of each string, there will be an extra "and" or "or" and they need to be removed before trying to execute them.

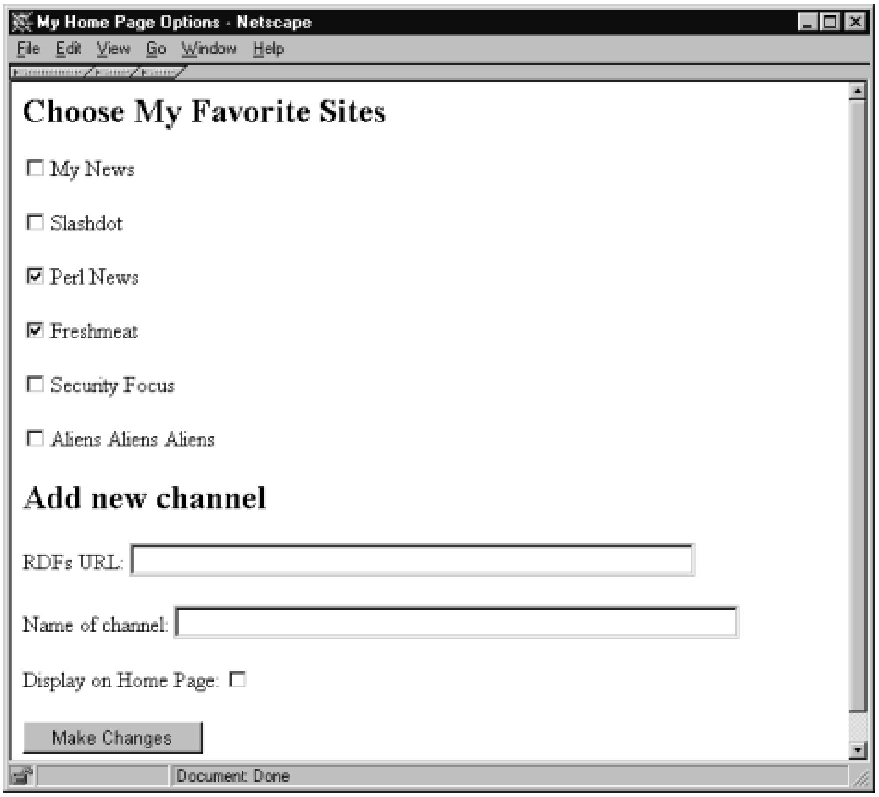
```
42:     my $sth = $dbh->prepare($qry_select);
43:     $sth->execute or print $DBI::errstr;
44:     $sth = $dbh->prepare($qry_deselect);
45:     $sth->execute or print $DBI::errstr;
```

Lines 42–45 *prepare()* and *execute()* the queries. When this is done, the database will match what the user wanted as far as what is Selected and what is not.

```
46:     if (param('URL') && param('Name')) {
47:             my $url = param('URL');
48:             my $name = param('Name');
49:             my $display = param('new-Selected') ? 1 : 0;
50:             $sth = $dbh->prepare(qq{insert into rdf (URL, NAME,
                   SELECTED) values ('$url', '$name', $display)});
51:             $sth->execute or print $DBI::errstr;
52:     }
53:     show_form;
54: }
```

Lines 46–54 complete the script. **Line 46** is checking to see if a URL and Name parameter have been passed to the script. Both are checked because we do not want to add a channel with no Name or no URL. If those parameters are true, the block continues and adds a record into the database, thereby creating a new channel. When this is all finished, the *show_form()* subroutine is called, and the form, with the updated information, is displayed to the screen.

Figure 16-2 Channel admin page


My Home Page Options - Netscape
File Edit View Go Window Help
Document: Done

Choose My Favorite Sites

☐ My News

☐ Slashdot

☑ Perl News

☑ Freshmeat

☐ Security Focus

☐ Aliens Aliens Aliens

Add new channel

RDFs URL:

Name of channel:

Display on Home Page: ☐

The three scripts explained in this section showed you how to create an RSS file as well as how to parse, read, and display them. The tools taught in this section can be used to provide, or receive other, information in Web applications. There are other features that can be added to this application, which will be suggested reader exercises at the end of this chapter. Using XML is growing in popularity and is an excellent tool for the programmer's toolbox. With the basic techniques taught in this chapter, you can already use Perl and XML to enhance Web applications.

16.3 Creating an RSS File

Now that you have seen how to use RSS files and are familiar with their structure, it is a good time to explain how a developer can create his or her own channel file. To make an RSS file useful in an application, it is necessary to have the essential element a channel needs—news to share with others—that is on the Web. For example's sake, we will assume that the information is kept in a text file. Of course, your information may be in a database or are HTML files in a special directory or some other type of data source.

The following example, make_rss, is not a CGI script. It is meant to be used as a command line script, which could be run from a schedule to automatically create the RSS file. This code, of course, could be run as a CGI, but unless your news changes by the minute, having it scheduled to run on intervals should suffice.

```
01: #!/usr/bin/perl -wT
02: # make_rss
03: use strict;
04: use XML::RSS;
```

Lines 1–4 define the path to Perl as well as pull in the strict pragma and XML::RSS module.

```
05: my $FILE = 'news.txt';
06: my $RDF_DIR = './rdf';
```

Lines 5 and 6 create two scalar variables we will be using later. **Line 5** initializes $FILE with the location of the text file that has the news information in it. This file, which we called news.txt, is a pipe delimited file with the URL for the news story to the left of the pipe and the description of the story on the right. On **line 6** the $RDF_DIR variable is created. This variable holds the location of the directory in which the RSS file is to be created.

```
07: my $rdf = new XML::RSS;
```

Line 7 creates a new XML::RSS object. The reference to that object is being stored in $rdf.

```
08: $rdf->channel(title => 'My News',
09:              link => 'http://news.me.com',
10:              language => 'en',
11:              description => 'My news, for you!',
12:              copyright => 'Copyright 2000++, Me',
13:               pubDate => scalar localtime(time),
14:               lastBuildDate => scalar localtime(time),
15:               managingEditor => 'me@me.com',
16:               webMaster => 'me@me.com'
17:              );
```

Lines 8–17 use the *channel()* method to define some of the main information about the channel itself and begin the “channel” container. Earlier in this chapter you were shown a complete RSS file, and you can see how each of these pairs is represented in the final document.

```
18: $rdf->image(title => 'My News',
19:             url => 'http://news.me.com/my_news.gif',
20:             link => 'http://news.me.com',
21:             height => 30,
22:             width => 119
23:            );
```

Lines 18–23 create the “image” container. This is optional, but if there is an icon available for the channel, this is how to add it.

```
24: open(FILE, $FILE) || die "Can't open $FILE ($!)";
25: while (<FILE>) {
26:    my ($url, $desc) = split /\|/;
27:    $rdf->add_item(title => $desc,
28:                   link => $url
29:                  );
30: }
```

Lines 24–30 get the data from the text file and use it to create the “item” containers. **Line 24** opens the file for reading or dies with an error. **Lines 25–30** are a *while()* loop that iterates over each line of the file. **Line 26** splits the input by the pipe (|) character and stores the values in the $url and $desc variables. **Lines 27 and 28** use the *add_item()* method to add an item into the “channel” container. As you saw in the RSS file example, title and link are the two elements in the “item” container. By simply creating named value pairs as arguments to *add_item()*, we can create a new container. When this loop is complete all of the items will be in the “channel” container.

```
31: $rdf->textinput(title => 'Search My News',
32:                 description => 'Search the Archives',
33:                 name => 'text',
34:                 link => 'http://news.me.com/search.cgi'
35:                );
```

Lines 31–35 use the *textinput()* method to create a “textinput” container. This is an optional container that will create a text box for people to use to search your site. The URL of the search script is the value of the “link” element, in which the name, title, and description of the link are the values of their corresponding key. At this point, the channel is created in the XML::RSS object.

```
36: $rdf->save("$RDF_DIR/my_news.rdf");
```

Line 36 ends the script by writing the file to disk using the *save()* method—and that easily a RSS file is created! This is the type of script that can be written and implemented in short time and needs little to no maintenance.

Now you are ready to take RSS full cycle from creating the data source, creating the RSS format file, and using that file for the Web.

16.4 Reader Exercises

- This application allows you to add and list channels. However, it doesn’t have the functionality to delete them. Building on the admin.cgi script, add in delete functionality.
- Create your own channel. If your company has news items or you just want to link to your favorite sites, use the XML::RSS module to create your own RSS file. Then add that channel to your database and take a look.
- Use the application created in this chapter, along with other topics from this book so far, and make this a multi-user system.

16.5 Listings

Listing 16-3 fetch script

```
#!/usr/bin/perl -w
# fetch
use strict;
use File::Basename;
use DBI;
use LWP::Simple qw(mirror);
my $RDF_DIR = './rdf';
my $dbh = DBI->connect("dbi:mysql:book", "user", "password");
my $sth = $dbh->prepare(qq{select URL from rdf});
$sth->execute or die $DBI::errstr;
while (my $url = $sth->fetchrow) {
```

(continued)

```
    my $name = basename($url);
    mirror($url, "$RDF_DIR/$name");
}
$dbh->disconnect;
```

Listing 16-4 index.cgi script

```
#!/usr/bin/perl -wT
# index.cgi
use strict;
use CGI qw(:standard end_ul end_table);
use CGI::Carp qw(fatalsToBrowser);
use File::Basename;
use DBI;
use XML::RSS;
my $RDF_DIR = './rdf';
my $dbh = DBI->connect("dbi:mysql:book", 'user','password')
          or print $DBI::errstr;
my $sth = $dbh->prepare(qq{select URL from rdf where
          Selected = 1});
$sth->execute;
print header,
      start_html("My Home Page"),
      h2("My Favorite Sites");
print start_table({cellpadding=>0, cellspacing=>0, border=> 0,
                   width => '100%'}),
      td;
my $count = 1;
my @html = ('</TD><TD>', '</TD><TR><TD>');
while (my $url = basename($sth->fetchrow)) {
    my $rss = new XML::RSS;
    eval {$rss->parsefile("$RDF_DIR/$url")};
    warn "$url will not parse $@" and next if $@;
    my $last_mod = scalar localtime((stat("$RDF_DIR/$url"))[9]);
    print start_table({cellpadding=>0, cellspacing=>2,
                       border=> 5,   width=>'75%'}),
    td({valign=>'CENTER', bgcolor => '#C0C0C0'});
    $rss->{image}{url}
    ? print img({src=>$rss->{image}{url}})
    : print strong($rss->{channel}{title});
    print ul;
    for (@{$rss->{items}}) {
        print li(a({href=>$_->{link}}, $_->{title}));
```

(continued)

```
    }
    print end_ul;
    if ($rss->{textinput}{link}) {
        print $rss->{textinput}{description},
                     start_form(-method => 'GET',
                           -action => $rss->{textinput}{link}),
            textfield(-name => $rss->{textinput}{name}),
            end_form;
    }
    print qq(Last Updated $last_mod<BR>),
        end_table;
    print $html[$count^=1];
}
print end_table,
      end_html;
```

Listing 16-5 **admin.cgi script**

```
#!/usr/bin/perl -w
# admin.cgi
use strict;
use CGI qw(:standard);
use CGI::Carp qw(fatalsToBrowser);
use DBI;
my $dbh = DBI->connect("dbi:mysql:book", "user", "password");
param('Submit') ? add_new() : show_form();
$dbh->disconnect;
sub show_form {
my $sth = $dbh->prepare(qq{select * from rdf});
$sth->execute;
print header,
      start_html("My Home Page Options"),
      h2("Choose My Favorite Sites");
print start_form(-method => 'POST', -action => 'admin.cgi');
 while (my $data = $sth->fetchrow_hashref) {
        my $checked = $data->{Selected} ? "CHECKED" : "";
        print checkbox(-name => 'Selected',
                               -checked => $checked,
                               -value => $data->{Name},
                               -label => $data->{Name},
                              ),
                p;
     }
```

(continued)

```
 print h2("Add new channel") , p,
                "RDFs URL: ", textfield(-name => 'URL', -size =>
                            50), p
                "Name of channel: ", textfield(-name => 'Name', -
                                   size => 50), p
                "Display on Home Page: ",
                checkbox(-name => 'new-Selected', -label => ''), p,
                submit(-name => 'Submit', -value => 'Make
                       Changes'),
                end_form, end_html;
}
sub add_new {
    my $qry_select = qq(update rdf set Selected = 1 where );
    my $qry_deselect = qq(update rdf set Selected = 0 where );
    my @selected = param('Selected');
    $qry_select .= qq(Name = '$_' or ) for @selected;
    $qry_deselect .= qq(Name <> '$_' and ) for @selected;
    $qry_deselect =~ s! and $!!;
    $qry_select =~ s! or $!!;
    my $sth = $dbh->prepare($qry_select);
    $sth->execute or print $DBI::errstr;
    $sth = $dbh->prepare($qry_deselect);
    $sth->execute or print $DBI::errstr;
    if (param('URL') && param('Name')) {
            my $url = param('URL');
            my $name = param('Name');
            my $display = param('new-Selected')  ? 1 : 0;
            $sth = $dbh->prepare(qq{insert into rdf (URL, NAME,
               SELECTED) values ('$url', '$name', $display)});
            $sth->execute or print $DBI::errstr;
    }
    show_form;
}
```

Listing 16-6 make_rss script

```
#!/usr/bin/perl -wT
# make_rss
use strict;
use XML::RSS;
my $FILE = 'news.txt';
my $RDF_DIR = './rdf';
my $rdf = new XML::RSS;
```

(continued)

```
$rdf->channel(title => 'My News',
              link  => 'http://news.me.com',
              language => 'en',
              description => 'My news, for you!',
              copyright => 'Copyright 2000++, Me',
              pubDate => scalar localtime(time),
              lastBuildDate => scalar localtime(time),
              managingEditor => 'me@me.com',
              webMaster => 'me@me.com'
             );
$rdf->image(title => 'My News',
            url => 'http://news.me.com/my_news.gif',
            link => 'http://news.me.com',
            height => 30,
            width => 119
           );
open(FILE, $FILE) || die "Can't open $FILE ($!)";
while (<FILE>) {
my ($url, $desc) = split /\|/;
$rdf->add_item(title => $desc,
                       link => $url
                      );
}
$rdf->textinput(title => 'Search My News',
                description => 'Search the Archives',
                name => 'text',
                link => 'http://news.me.com/search.cgi'
               );
$rdf->save("$RDF_DIR/my_news.rdf");
```