practical perl

Web Automation

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Introduction

Web service protocols like XML-RPC and SOAP are great for automating common tasks on the Web. But these protocols aren't always available. Sometimes interacting with HTML-based interfaces is still necessary. Thankfully, Perl has the tools to help you get your job done.

In my last column, I introduced Web services using XML-RPC. Web services are commonly used as a high-level RPC (remote procedure call) mechanism to allow two programs to share data. They enable programs to exchange information with each other by sending XML documents over HTTP.

There are many advantages to using Web service tools like XML-RPC and its cousin, SOAP. First, all of the low-level details of writing a client and server programs are handled by reusable libraries. No longer is it necessary to master the arcana of socket programming and protocol design to implement or use a new service or daemon. Because information is exchanged as text, Web services are programming-language agnostic. You could write a service in Perl to deliver weather information, and access it with clients written in Python, Tcl, Java, or C#. Or vice versa.

Yet for all of the benefits Web services bring, they are hardly a panacea. Protocols like SOAP and XML-RPC focus on how programs interact with each other, not on how people interact with programs. For example, I cannot scribble down the address of an XML-RPC service on a napkin and expect someone to use that service easily. Nor can I send a link to an XML-RPC service in the body of an email message.

Generally speaking, in order to use a Web service, I need to write some code, and have an understanding of how to use that particular service. This is why after about five years, Web services are still a niche technology. They work great if you want to offer programmatic access to a service like, say, eBay, Google, or Amazon.com. But if you want to publish information or offer a service to the widest possible audience, you still need to build a Web site.

The HTML Problem

Before Web services, automatic processing of data from the Web usually involved fetching HTML documents and scanning them to find new or interesting bits of data. Web services offer a more robust alternative, but do not eliminate the need to sift through HTML documents and "screen scrape" data off a Web page.

Processing HTML is the worst possible solution, but it is often the only solution available. HTML is a difficult format to parse. Many documents contain invalid or otherwise broken formatting. Using regular expressions to extract information from HTML documents is a common coping strategy, but it is quite error-prone and notoriously brittle.

Nevertheless, HTML is the universal format for data on the Web. Programmers who are building systems may consider alternatives like XML-RPC or SOAP Web services. But publishers and service providers are still focused on HTML, because it is the one format that everyone with a Web browser can always use.

Automating the Web

Since the early days of the Web, people have used programs that automatically scan, monitor, mirror, and fetch information from the Web. These programs are generally called robots or spiders. Today, other kinds of programs traverse the Web, too. Spammers use email harvesters to scour Web pages for email addresses they can spam. In the semantics of the Web community, "scutters" follow links to metadata files to build up databases of information about who's who and what's what on the Web.

There are many other mundane uses for Web automation programs. Link checkers rigorously fetch all the resources on a Web site to find and report broken links. With software development moving to the Web, testers use scripts to simulate a user session to make sure Web applications behave properly.

Fortunately, there are a great many Perl modules on CPAN to help with all of these tasks.

Most Web automation programs in Perl start with libwww-perl, more commonly known as LWP. This library of modules is Gisle Aas's Swiss Army knife for interacting with the Web. The easiest way to get started with LWP is with the LWP::Simple module, which provides a simple interface to fetch Web resources:

```
#!/usr/bin/perl -w

use strict;
use LWP::Simple;

## Grab a Web page, and throw the content in a Perl variable.
my $content = get("http://www.usenix.org/publications/login/");

## Grab a Web page, and write the content to disk.
getstore("http://www.usenix.org/publications/login/", "login.html");

## Grab a Web page, and write the content to disk if it has changed.
mirror("http://www.usenix.org/publications/login/", "login.html");
```

LWP has other interfaces that enable you to customize exactly how your program will interact with the Web sites it visits. For more details about LWP's capabilities, check out the documentation that comes with the module, including the lwpcook and lwptut man pages. Sean Burke's book *Perl & LWP* also provides an introduction to and overview of LWP.

Screen Scraping

Retrieving Web resources is the easy part of automating Web access. Once HTML files have been fetched, they need to be examined. Simple Web tools like link checkers only care about the URLs for the clickable links, images, and other files embedded in a Web page. One easy way to find these pieces of data is to use the HTML::LinkExtor module to parse an HTML document and extract only these links. HTML::LinkExtor is another of one of Gisle's modules that can be found in his HTML::Parser distribution.

```
#!/usr/bin/perl -w

use strict;
use LWP::Simple;
use HTML::LinkExtor;

my $content = get("http://www.usenix.org/publications/login/");

my $extractor = new HTML::LinkExtor;

$extractor->parse($content);

my @links = $extractor->links();

foreach my $link (@links) {
    ## $link is a 3-element array reference containing
    ## element name, attribute name, and URL:
    ##
    ## 0 1 2
    ## <a href="http://....">
    ## <img src="http://....">
print "$link->[2]\n";
}
```

Most modern Web sites have common user interface elements that appear on every page. These are elements like page headers, page footers, and navigation columns. The actual content of a page is embedded inside these repeating interface elements that appear on every page of a Web site. Sometimes, a screen scraper will want to ignore all of the repeatable elements and focus instead on the page-specific content for each HTML page it examines.

For example, the O'Reilly book catalog (*http://www.oreilly.com/catalog/*) has each of these three common interface elements. The header, footer, and navigation column on this page all contain links to ads and to other parts of the O'Reilly Web site. A program that monitors the book links on this page is only concerned with a small portion of this Web page, the actual list of book titles.

One way to focus on the meaningful content is to examine the structure of the URLs on this page, and create a regular expression that matches only the URLs on the list of titles. But when the URLs change, your program breaks. Another way to solve this problem is to write a regular expression that matches the HTML content of the *entire* book list, and throw out the extraneous parts of this

Web page. Both of these approaches can work, but they are error-prone. Both will fail if the page design changes in a subtle or a significant manner.

Of course, this is Perl, so there's more than one way to do it. Many Web page designs are built using a series of HTML tables. A better way to find the relevant content on this Web page is to parse the HTML and focus on the portion of the page that contains what we want to examine. This approach isn't foolproof, but it is more robust than using a regular expression to match portions of a Web page and fixing your program each time the Web page you are analyzing changes.

There are a few modules on CPAN that handle parsing HTML content. While HTML::Parser can provide a good general-purpose solution, I prefer Simon Drabble's HTML::TableContentParser, which focuses on extracting the HTML tables found in a Web page. This technique will break if the HTML layout changes drastically, but at least it is less likely to break when insignificant changes to the HTML structure appear.

```
#!/usr/bin/perl -w

use strict;
use LWP::Simple;
use HTML::TableContentParser;

my $content = get("http://www.oreilly.com/catalog/");
my $parser = new HTML::TableContentParser;
my $tables = $parser->parse($content);

## $tables is an array reference. Select to the specific table
## content and process it directly.
```

Interacting with the Web

Most Web automation techniques, like the ones described above, focus on fetching a page and processing the result. This kind of shallow interaction is sufficient for simple automation tasks, like link checking or mirroring. For more complicated automation, scripts need to be able to do all the things a person could do with a Web browser. This means entering data into forms, clicking on specific links in a specific order, and using the back and reload buttons.

This is where Andy Lester's WWW::Mechanize comes in. Mechanize provides a simple programmatic interface to script a virtual user navigating through a Web site or using a Web application.

Consider a shopping cart application. A user starts by browsing or searching for products, and periodically clicks on "Add to Shopping Cart." On the shopping cart page, the user can click on the "Continue shopping" button, click on the back button, browse elsewhere on the Web site, or search for products.

If you were developing this application, how would you test it? Would you write down detailed instructions for the people on your test team to repeat by rote? Or would you write a program to simulate a user, checking each and every intermediate result along the way? Mechanize is the tool you need to write your simulated user scripts. That user script might look something like this:

```
#!/usr/bin/perl -w

use strict;
use WWW::Mechanize;

my $mech = new WWW::Mechanize;

## Start with the homepage.
$mech->get("http://localhost/myshop.cgi");

## Browse for a book.
$mech->follow_link( text => "Books" );
$mech->follow_link( text_regex => qr/Computers/ );
$mech->follow_link( text_regex => qr/Perl/ );

## Put "Programming Perl" in the shopping cart.
```

```
$mech->follow_link( text_regex => qr/Programming Perl/);

## Add this to the shopping cart.
$mech->click_button( name => "AddToCart");

## Click the "back button."
$mech->back();

## Check out.
$mech->click_button( name => "Checkout");

## Fill in the shipping and billing information.
....
```

Mechanize is also an excellent module for scripting common actions. Every other week, I need to use a Web-based time-tracking application to tally up how much time I've worked in the current pay period. I could fire up a browser and type in the same thing I typed in two weeks ago. Or I could use Mechanize:

```
#!/usr/bin/perl -w

use strict;
use WWW::Mechanize;

my $mech = new WWW::Mechanize;

$mech->get('...');

## Log in.
$mech->set_fields(
    user => "my_username",
    pass => "my_password",
);
$mech->submit();

## Put in a standard work week.
## Log in manually later if this needs to be adjusted.
## (Timesheet is the 2nd form. Skip the calendar.)

$mech->submit_form (
    form_number => 1,
    fields => {
        0 => 7.5,
        1 => 7.5,
        ...
        9 => 7.5,
    },
    button => "Save",
);

## That's it. Run this again in two weeks.
```

Mechanize is also a great module for writing simple Web automation. Scripts that rely on HTML layout or specific textual artifacts in HTML documents are prone to breaking whenever a page layout changes. For example, whenever I am reading a multi-page article on the Web, I invariably click on the "Print" link to read the article all at once.

I could use regular expressions, or modules like HTML::LinkExtor or HTML::TableContentParser, to examine the content of a Web page to find the printable version of an article. But these techniques are both site-specific and prone to breakage. With Mechanize, I can analyze the text of a link — the stuff that appears underlined in blue in my Web browser. Using Mechanize, I can look for the "Print" link and just follow it:

```
#!/usr/bin/perl -w

use strict;
use WWW::Mechanize;

my $mech = new WWW::Mechanize;

my $url = shift(@ARGV)
$mech->get();

if ($mech->find_link(text_regex => qr/^2|Next$/i)) {
    ## This is a multipage document.
    ## Open the "print" version instead.
    $url = $mech->find_link(text_regex => qr/Print/);
}

## Open the file in a browser (on MacOS X).
system("open '$url'");
```

Conclusion

Perl is well known for automating the drudgery out of system administration. But Perl is also very capable of automating Web-based interactions. Whether you are using Web service interfaces like XML-RPC and SOAP or interacting with standard HTML-based interfaces, Perl has the tools to help you automate frequent, repetitive tasks.

Perl programmers have a host of tools available to help them automate the Web. Simple automation can be accomplished quickly and easily with LWP::Simple and a couple of regular expressions. More intensive HTML analysis can be done using modules like HTML::LinkExtor, HTML::Parser, HTML::TableContentParser, or WWW::Mechanize, to name a few. Whatever you need to automate on the Web, there's probably a Perl module ready to help you quickly write a robust tool to solve your problem.