Practical Perl Tools

H-T...TP—That's What It Means to Me

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

Socket to me, socket to me...Okay, sorry, enough with an imitation of Aretha Franklin and Tim Berners-Lee's love-child. Instead, let's continue with a thread we began in last issue's column. In that column, we discussed using various protocols like FTP, rsync, and SSH from Perl to move data from one place to another. The one protocol that we were able to press our nose onto the glass to see but not really touch was HTTP, the Hypertext Transfer Protocol. As promised, here's an entire column on just that subject to help make up for the omission.

Allow me to get some caveats out onto the table before we go much further. First off, I'm probably not going to spend much time describing the internals of the protocol. If I said that the standard request consists of the request itself, some headers that provide some context for the request (such as preferring a response that's compressed or in a specific language) and sometimes a message body, that about covers what you need to know. We'll talk a bit about the different request types ("methods") that communicate using this format, but we're going to stick to only the top three kinds of types. Funky request types like PATCH, defined just last year in RFC5789, won't have even a cameo appearance (unless this sentence counts).

That's the "what we won't be talking about" caveats; here's the "what we will be talking about" set: in the Perl world, there is one group of modules that almost totally dominates the space: Library for WWW in Perl, more commonly known as LWP. Gisle Aas deserves tremendous credit for actively maintaining such a useful set of modules, for over 16 years at this point. It's the first module people turn to for doing lower-level HTTP stuff. There are some others at the periphery that are useful for edge cases, but I'll only mention them in passing. Largely this is going to be "The LWP Show."

Lest you worry there won't be enough material based on just this set of modules, let me point out that there is an excellent book by Sean Burke called *Perl & LWP* (O'Reilly, 2002). I'm just going to skim the surface of the topic here rather than try to rewrite Sean's book. If you'd like to get more advanced in your LWP work, you may want to hunt down a copy. One thing you'll find in that book that is totally absent from this column is the parsing of any data that gets returned via HTTP. We've touched on this subject in other columns, but unfortunately there is not enough room to explore that topic here.

Simple Pleasures Are the Best

I'd like to start out with the simplest way to use LWP. It is entirely possible that this section will cover the vast majority of your day-to-day needs. If that's the case, you can probably bail at the end of the section and just check out the marvelous proof I give for why it is impossible to separate a cube into two cubes or a fourth power into two fourth powers, or, in general, any power higher than the second into two like powers (assuming it fits in the margin of this column). But if you do bail, you're going to miss a pointer to some of the cooler things that ship with LWP that many people miss.

The key to all of this magic-fairy-simple-dust is the LWP::Simple module. It lets you write code that looks like this:

```
use LWP::Simple;

my $page = get('http://www.usenix.org');
die "Could not retrieve page" unless defined $page;

#... do something with $page
```

That's how hard it is to fetch a Web page. If we wanted to print the information we received instead of storing it in a scalar variable, we could have used getprint() instead of get(). There is a similar function, getstore(), that lets you drop that information into a file instead.

LWP::Simple is mostly useful for HTTP GET operations, but it also can make a HTTP HEAD request (often used to see if a document has changed since the last time you fetched it). The head() function returns a list with the following information:

- Content type
- Document length
- Modified time
- Expiration
- Server

So, for example, when I ran:

```
use LWP::Simple;
use Data::Dumper;

my @results = head('http://www.usenix.org');

print Dumper \@results;
```

the output was:

```
$VAR1 =  [
            'text/html',
            '65735',
            1296087147,
            undef,
            'Apache/1.3.41 (Unix) mod_perl/1.31 mod_ssl/2.8.31 OpenSSL/0.9.8k'
         ];
```

Be the Agent of Your User

If your needs extend beyond those that LWP::Simple can handle—for example, you need to do more than just a generic HTTP GET or HEAD request—then it is time to move up the food chain. The next most commonly used part of LWP is the

LWP::UserAgent module. With this jump, we leave the land of naive function calls behind and enter LWP's object-oriented framework. You won't need to be an OOP ace to make use of the modules we're going to talk about, but I thought it best to mention this switch before I start to use words like "class" and "method" for the first time, in the next paragraph:

The LWP::UserAgent module provides what the LWP tutorial calls the two main classes you have to understand from LWP. They are the eponymous LWP::UserAgent class and the HTTP::Response class. The LWP::UserAgent class lets you write a "Web user agent…to dispatch Web requests" (to use the terms from the documentation). Answers to these requests come back to us in HTTP::Request objects. You don't have to explicitly load the HTTP::Request module to work with these objects, since the LWP::UserAgent module will do that for you. This all probably sounds more complex than it is. Let's look at some code to calm any jitters:

```
use LWP::UserAgent; # or use LWP;
my $ua = LWP::UserAgent->new;
my $resp = $ua->get('http://usenix.org/');
die "Could not retrieve page" unless $resp->is_success;
# do something with $resp->decoded_content;
```

To get us started with LWP::UserAgent, I just rewrote the first LWP::Simple code sample from above. First, we create a LWP::UserAgent object, and then we call the get() method from that object to make our request. It returns a response (HTTP::Response, to be precise) object that gives us methods to test the success of the request and return the contents of the reply. No biggee so far, right?

Now let's do something we couldn't do with LWP::Simple. Let's tune the request a bit. Some Web sites will tailor their responses to a query based on the browser making the request. If we want to pretend to be another browser, we can change some of the default settings, such as the header the browser uses to identify itself, before we actually make the get() request. Let's make your favorite Web designer twitch a little by pretending to be an Internet Explorer 5.0 session on a Solaris box:

```
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent('Mozilla/4.0 (compatible; MSIE 5.0; SunOS 5.10 sun4u; X11)');
```

Perhaps an even more useful method you might use is credentials(). This lets you authenticate to a Web site that is using basic authentication (ideally, over HTTPS, something we'll mention in just a bit). The documentation shows the parameters it requires:

```
$ua->credentials( $netloc, $realm, $uname, $pass );
```

with this as the example:

```
$ua->credentials("www.example.com:80", "Some Realm", "foo", "secret");
```

This will allow you to authenticate to example.com (i.e., at the point where it requests a username and password for "Some Realm") with that username and password.

If you need to send other headers along with your get() request (e.g., the language you expect a response in or a desire to get the result back in a compressed form), get() takes a set of field/value pairs after the URL. In addition to real headers such as Accept-Language and Accept-Charset, it also takes "special" fields (e.g., :content_file) which let you redirect the contents of the request to a file. This is quite important in cases where you will be slinging lots of data around. For example:

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

my $resp = $ua->get('http://usenix.org/TerabytesOfFun.html',
                        ':content_file' => 'TerabytesOfFun.html');
```

All GET and No POST Makes Jack a Dull Browser

So far we've only covered writing the kind of HTTP requests that retrieve information, but we've said nary a word about how one can submit information over HTTP. For those requests, we'll have to learn how to submit form elements using both GET and POST requests. Let's take it in that order.

GET request submissions are those that crowd up your browser's URL bar with bunches of parameters sent as part of the URL:

```
http://forecast.weather.gov/MapClick.php?CityName=Boston&state=MA&site=BOX
&textField1=42.3583&textField2=-71.0603&e=0
```

Constructing a huge URL like this programmatically isn't necessarily much harder than just string concatenation, but there are a few catches. These catches are largely centered on what characters are legal for a URL and which need to be escaped before they can be used. There are a few Perl modules that will pay attention to those details so we don't have to. Here's a slightly modified version of an example given in the LWP tutorial:

```
use URI;
use LWP::UserAgent;

# makes an object representing the URL
my $url = URI->new( 'http://us.imdb.com/Tsearch' );

# And here are the form data pairs:
$url->query_form(
    'title'    => 'Blade Runner',
    'restrict' => 'Movies and TV',
);

my $ua = LWP::UserAgent->new;
my $resp = $ua->get($url);

# do something with $resp->decoded_content;
```

It uses the URI module's query_form() method to create the form submission in URL form, so we can perform a get() request just like before. If we were to print `$url` after it was constructed, it would look like this:

```
http://us.imdb.com/Tsearch?title=Blade+Runner&restrict=Movies+and+TV
```

This isn't particularly complex, but I've seen some monster URLs that you wouldn't want to construct by hand. It is far safer to use query_form() when possible.

Okay, so if that is a GET submission, how would we go about doing a POST? It turns out that simple POSTs are actually easier than the GET request we just saw. LWP::UserAgent has a post() method that directly takes the field/value pairs in an array right after the submission URL. Here's an example that demonstrates a US Postal code lookup:

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

my $resp = $ua->post('http://zip4.usps.com/zip4/zcl_1_results.jsp',
                       ['city'         =>'Boston',
                        'state'        => 'MA',
                        'pagenumber'  => 0 ] );

# do something with $resp->decoded_content;
```

The hardest part of that code is probably determining the names of the form's fields. For this, you can look at the source code of the page (although I cheated and used the `--forms` option to the mech-dump utility we discussed in this column back in February of 2009). In the interests of full disclosure, the code is so simple because the form is simple. HTML forms can get considerably more complex when you start wanting to do things like file uploads. *Perl & LWP* and the lwpcook (LWP Cookbook) man page are two good resources for handling the more advanced cases.

I do want to offer one hint on the subject of forms before we start to leave our discussion of LWP::UserAgent. I was surprised ,when preparing for this column, just how skewed the balance of GET to POST forms is on the major Web sites. It took me over an hour to find a POST form for the previous example that would take in input and return a page of useful output. The vast majority of sites used GET-based forms on their front pages.

A significant number of the rest that did have POST forms used them in their search boxes, but didn't return a page of results directly. Instead, when you posted a search request, the response would be an HTTP Redirect (302 status code) pointing at some other page on the site. By default, LWP::UserAgent only chases referrals for you on GET requests. If you are dealing with a POST request, you can either:

1. Tell LWP::UserAgent to also chase referrals for POST requests, despite RFC 2616 suggesting that this is a baaaad idea:

```
push @{ $ua->requests_redirectable }, 'POST';
```

2. Grab the referral and choose whether to chase it yourself:

```
if ($resp->is_referral) {
   # go chase the value of $resp->header('Location')
}
```

Now that we've seen how to use LWP::UserAgent to POST data, what haven't we covered? A number of things: LWP::UserAgent can handle cookies with just a few lines of initialization. It can deal with proxies. It can use callbacks to give your code more precise control over how it will handle the data that comes in. Please see the LWP::UserAgent documentation, the lwpcook/lwptut manual pages, and *Perl & LWP* for more details.

There is one thing I do have to mention before we leave this section, if only to avoid harshing the mellow of my security-conscious editor. We haven't talked about how LWP::UserAgent handles HTTPS requests. I'm afraid this may be a bit of a letdown, because the answer is "it just does." As long as you have either the Crypt::SSLeay or IO::Socket::SSL modules installed, any URL that begins with https instead of http just works as you'd expect.

What Else?

We're just about at the end of this column, but it is worthwhile mentioning some of the additional capabilities of the LWP module group and two of their competitors. In LWP itself, you can find LWP::RobotUA, basically a robots.txt-compliant version of LWP::UserAgent, and LWP::ConnCache, available to let you use HTTP/1.1 "Keep-Alive" to improve performance of multiple requests to the same destination.

And here's the tip you've been waiting for ever since the foreshadowing in the LWP::Simple section: LWP comes with a number of really useful command-line scripts that get installed when the module group itself is installed: lwp-download, lwp-dump (only included in later versions of the LWP distribution), lwp-mirror, lwp-request, and lwp-rget. You can also choose to install aliases for lwp-request called GET, HEAD, and POST to make, for example, HTTP HEAD requests from the command-line really easy. If you've never noticed any of these utilities before, be sure to check them out. I just recently used lwp-download on a shared host of an ISP that didn't have wget or curl installed, so I know they can be a great help at times.

Even though LWP is the most widely used framework for doing the sort of stuff we've been looking at in this column, it doesn't work for everyone. For example, the people who enjoyed my October 2010 column on ::Lite modules might also want a replacement for this framework that isn't quite as heavyweight. There is a HTTP::Lite module which attempts to provide that, but judging by the HTTP::Tiny documentation (which claims it is more correct and more complete than HTTP::Lite), HTTP::Tiny may be the module for you.

Take care, and I'll see you next time.