Practical Perl Tools
Give it a REST

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 29+ 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

Believe it or not, there's a good reason that this column returns to the subject of Web API programming on a fairly regular basis. As time goes on, much of the work of your average, ordinary, run-of-the-mill sysadmin/devops/SRE person involves interacting/integrating with and incorporating services other people have built as part of the infrastructure we run. I think it is safe to say that a goodly number of these interactions take place or will take place via a REST-based API. Given this, I thought it might be a good idea to take a quick look at some of the current Perl modules that can make this process easier.

Nice Thesis, Pal

Before we get into the actual Perl code, it is probably a good idea to take a brief moment to discuss what REST is. People used to argue about what is and what isn't REST, but I haven't heard those arguments in years. (I suspect enough people abused the term over the years that those who cared just threw up their hands.) Back in 2012 I wrote a column that included an intro description about REST; let me quote from myself now:

REST stands for "Representational State Transfer." The Wikipedia article on REST at http://en.wikipedia.org/wiki/Representational_State_Transfer is decent (or was on the day I read it). Let me quote from it:

"REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource....

Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: presented with a network of Web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

(That last quote comes from Roy Fielding's dissertation, which actually defined the REST architecture and changed Web services forever as a result.)

That's a pure description of REST. In practice, people tend to think about REST APIs as those that use a combination of HTTP operations (GET/PUT/DELETE) aimed at URLs that represent the objects in the service. Here's an overly simplified example just for demonstration purposes:

```
GET /api/shoes  - return a list of available shoes
GET /api/shoes/shoeid - return more detailed info on that shoe
DELETE /api/shoes/shoeid - delete that kind of shoe
PUT /api/shoes/shoeid - update inventory of that kind of shoe
```

This just gives you a fleeting glance at the idea of using HTTP operations as the verb and intentionally constructed URLs as the direct objects for these operations. But to quote the Barbie doll, "APIs are hard, let's go shopping!"

What Do We Need

As a good segue to the modules that are out there, let's chat about what sort of things we might like in a module that will assist with REST programming. I'm focusing on REST clients in this column, but who knows, we might get crazy in a later column and talk about the server side of things as well.

The first thing we'll clearly need is an easy way to construct HTTP requests. The messages being passed to and fro over these requests is likely to be in JSON format (the current *lingua franca* for this sort of thing) or XML format, so it would be great if that didn't cause the module to break a sweat. Beyond this, it can be helpful to have the module understand the usual request-reply-request more workflow and perhaps add a little syntactic sugar to the process to make the code easier to read. Okay, let's see what Perl can offer us.

HTTP Me

We are going to be looking at modules that do lots more hand-holding than this category, but I feel compelled to start with something a little lower level. Sometimes you will want to whip out a very small script that makes a few HTTP calls. The classic module for this sort of thing was LWP::Simple or LWP::UserAgent (as part of the libwww package). Recently I've found myself using two other modules instead.

The first is one of the ::Tiny modules. You may recall from a previous column that I love that genre of modules. These are the relatively recent trend in Perl modules to produce a set of modules that are mean and lean and do one thing well with a minimum of code. The ::Tiny module in play for this column is HTTP::Tiny. Here's a small excerpt from the doc:

```
use HTTP::Tiny;

my $response = HTTP::Tiny->new->get('http://example.com/');
die "Failed!\n" unless $response->{success};

print "$response->{status} $response->{reason}\n";
…
print $response->{content} if length $response->{content};
```

As you can see, performing a GET operation with HTTP::Tiny is super easy (the same goes for a HEAD, DELETE, or POST) as is getting the results back. HTTP::Tiny will also handle SSL for you if the required external modules are also available. I'd also recommend you check out the small ecosystem of available modules that attempt to build on HTTP::Tiny (e.g., HTTP::Tiny::UA, HTTP::Tiny::SPDY, HTTP::Retry, and HTTP::Tiny::Paranoid).

Besides using HTTP::Tiny, I've also been enjoying using some of the fun stuff that comes with Mojolicious, the Web programming framework we've seen in past columns. For simple operations, it can look a lot like LWP::UserAgent:

```
use Mojo::UserAgent;

$ua = Mojo::UserAgent->new;
print $ua->get('www.google.com')->res->body
```

That's not all that exciting. More exciting is when you combine this with some of the great Mojolicious DOM processing tools. Even more fun is when you use the "ojo" module to construct one-liners. (Quick explanatory aside: The module is called "ojo" because it gets used with the Perl runtime flag -M used to load a module from the command line. So that means you get to write Mojo on the command line.) Once again, let me borrow from the Mojolicious documentation to show you a couple of cool one-liners:

```
$ perl -Mojo -E 'say g("mojolicio.us")->dom->at("title")->text'
Mojolicious - Perl real-time web framework
```

This uses the g() alias to get the Web page at http://mojolicio.us, find the title element in the DOM, and print the text in that element (i.e., the title of the page).

```
$ perl -Mojo -E 'say r(g("google.com")->headers->to_hash)'
```

This code performs a GET of google.com, returns the headers it gets back as a hash, and then performs a data dump (r()) of them. The end result looks something like this:

```
{
  "Alternate-Protocol" => "80:quic,p=0.002",
  "Cache-Control" => "private, max-age=0",
  "Content-Length" => 19702,
  "Content-Type" => "text/html; charset=ISO-8859-1",
  "Date" => "Fri, 28 Nov 2014 03:56:53 GMT",
  "Expires" => -1,
  "P3P" => "CP=\"This is not a P3P policy! See
 http://www.google.com/support/accounts/bin/answer.py?hl=en
&answer=151657 for more info.\"",
  "Server" => "gws",
  "Set-Cookie" =>
 "PREF=ID=fdfedfe972efadfb:FF=0:TM=1417147013:LM=141714701
3:S=hy1EI4K1BJDEX3to; expires=Sun, 27-Nov-2016 03:56:53 GMT;
path=/; domain=.google.com, NID=67=kCOtIKopo0mStqWp3xSj7
nM0iQkt-GoL9D3Ena9y8EcAm95Z2Ki-c7-NGjWYG878nHQ6tVE-Y3
JkqAM68YR1B6IsGuDL2Cd4UCYI2N35VMM66RcywTTGo6hAH8_Al8Wq
; expires=Sat, 30-May-2015 03:56:53 GMT; path=/; domain
=.google.com; HttpOnly",
 "X-Frame-Options" => "SAMEORIGIN",
 "X-XSS-Protection" => "1; mode=block"
 }
```

I find the ease of working with the structure of the page (via the DOM or CSS selectors) to be particularly handy, but do check out the rest of the documentation for the many other neat tricks Mojolicious can perform.

Get On with the REST

So let's start looking at the sorts of modules that are trying to help us with our REST work. We'll take this in the order of least hand-holdy (is that a word?) to most hand-holdy. You'll find that the earlier modules look very much like the HTTP request modules we've already seen. For example, let's see some sample code that uses REST::Client. For almost all of the examples in this column, we are going to use the handy sample REST service the developer Thomas Bayer has been kind enough to provide (as a demo for his sqlREST package found at http://sqlrest.sourceforge.net).

```
use REST::Client;

my $rc = REST::Client->new(
  host => 'www.thomas-bayer.com',
  timeout => 10, );

$rc->GET('/sqlrest/CUSTOMER/');
print $rc->responseContent(),"\n---\n";
$rc->GET('/sqlrest/CUSTOMER/3');
print $rc->responseContent();
```

The result of running this code looks something like this:

```
<?xml version="1.0"?><CUSTOMERList xmlns:xlink=
"http://www.w3.org/1999/xlink">
 <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/0/">0</CUSTOMER>
 <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/1/">1</CUSTOMER>
 <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/2/">2</CUSTOMER>
 <CUSTOMER xlink:href="http://www.thomas-bayer.com/sqlrest
/CUSTOMER/3/">3</CUSTOMER>
 …
 ---
 <?xml version="1.0"?><CUSTOMER xmlns:xlink=
 "http://www.w3.org/1999/xlink">
 <ID>3</ID>
 <FIRSTNAME>Michael</FIRSTNAME>
 <LASTNAME>Clancy</LASTNAME>
 <STREET>542 Upland Pl.</STREET>
 <CITY>San Francisco</CITY>
```

We've performed a GET to receive a set of URLs in XML format that represent the available customer list and then performed a second GET to pull information for the customer with ID 3. This second step was all manually done (i.e., I picked #3 at random), but you can easily imagine using something like XML::LibXML or XML::Simple to parse the initial list that was returned, and then use some complicated process to determine which customer ID (or all of them) for the second step.

So I bet you are wondering why REST::Client is any better than HTTP::Tiny in this case. It is only a hair more helpful. The helpful part comes largely in the new() call where we could set a default host (meaning we only have to put the path into the GET requests), a timeout, and settings for SSL/redirects (which we didn't use). Responses are a little easier to retrieve, but on the whole, nothing exciting.

A step up from this is something like WebService::CRUST. This module is a step further in the direction of "more hand-holdy" for a few reasons:

- It can take more "default" settings in the object constructor, so the actual query lines only have to contain the parameters explicit to the query.
- It knows how to hand the results back to a parser of some sort (i.e., to decode the XML or the JSON we get back).
- It adds some syntactic sugar, which makes the actual queries look more intuitive.

Let's do a quick rewrite of the REST::Client code to use WebService::Crust instead:

```
use WebService::Crust;
use Data::Dumper;

my $wc = new WebService::CRUST(
  base     => 'http://www.thomas-bayer.com/sqlrest/',
  timeout  => 10,
  # params => { appid => 'SomeID' },
);

# same as $wc->get('CUSTOMER')
# same as $wc->get_CUSTOMER();
my $reply = $wc->CUSTOMER;

print Dumper $reply->result;
print "\n---\n";
my $reply = $wc->get('CUSTOMER/3');
print Dumper $reply->result;
```

The first thing to note in this code is the constructor takes a base for the API so we never have to repeat the URL in our code. It also can take a params hash that will be used to add parameters to every call. For example, if your API required you to send along some API-specific key in each call, you could easily do it here. Our test service doesn't call for this, so I placed a commented-out version there instead. And, in the spirit of making the actual API calls easier, you can see that WebService::CRUST lets us write things as a method call (->CUSTOMER or get_CUSTOMER), which makes the code even more readable.

Now on to the more interesting part. If we were to actually dump out the contents of $reply at this point, we'd find it contained not the XML that the call returned, but a Perl data structure that represented the parsed version of that XML, hence the use of Data::Dumper to display it. The actual data structure looks like this:

```
DB<1> x  $reply->result
0 HASH(0x7fb96cbedbe0)
 'CUSTOMER' => ARRAY(0x7fa1abd96e40)
   0  HASH(0x7fa1abc674b8)
   'content' => 0
   'xlink:href' => 'http://www.thomas-bayer.com/sqlrest
    /CUSTOMER/0/'
   1  HASH(0x7fa1abd82cc0)
   'content' => 1
   'xlink:href' => 'http://www.thomas-bayer.com/sqlrest
    /CUSTOMER/1/'
   2  HASH(0x7fa1abb29570)
   'content' => 2
   'xlink:href' => 'http://www.thomas-bayer.com/sqlrest
    /CUSTOMER/2/'
   3  HASH(0x7fa1a9ee7848)
   'content' => 3
   'xlink:href' => 'http://www.thomas-bayer.com/sqlrest
    /CUSTOMER/3/'
...
```

There's a hash with a single key called "result". The value of this key is a reference to a hash that contains a single key (CUSTOMER). That key has a reference to an array of hashes, each containing the XML elements we'll need to access. If you think this particular data structure is kind of icky, I'm in your corner. What you are seeing here is the default parse rules from XML::Simple at work. Sometimes they work well given a hunk of XML, sometimes not as well. They work better for the second query (the one where we request the info for customer #3):

```
{
'LASTNAME' => 'Clancy',
'FIRSTNAME' => 'Michael',
 'ID' => '3',
 'xmlns:xlink' => 'http://www.w3.org/1999/xlink',
 'CITY' => 'San Francisco',
 'STREET' => '542 Upland Pl.'
};
```

The two ways to deal with the icky data structure takes us too far afield to look at in depth, but just to give you a head start on the problem, you could either:

- write a subroutine that takes in the unpleasant data structure and returns one that is easier to use, or
- use the "opts" constructor option in the new() call to pass along options to XML::Simple. XML::Simple is quite willing to do your bidding, you'll just have to tell it exactly what you need.

Now, just so we don't lose track of one of the desired qualities of REST modules we mentioned earlier, I want to make sure JSON gets at least a brief mention. So far our test code has talked to APIs that return XML; what would we do if we had to talk to something that spoke only JSON? A couple of possibilities leap right to mind. First, we could switch modules. There are modules like REST::Consumer that behave similarly to WebService::CRUST. REST::Consumer offers a little less syntactic sugar than WebService::CRUST, but it does expect to receive (and send) JSON data as a default. Since I have a sweet tooth sometimes that craves the sugar (for readability purposes, I assure you), a second possibility is to continue using WebService::CRUST. It allows you to write:

```
my $wc = new WebService::CRUST(
 format => [ 'JSON::XS', 'decode', 'encode', 'decode' ]);
```

and from now on WebService::CRUST will speak JSON by using the JSON::XS (the faster JSON module) to decode and encode messages for you.

By the Way

As a way of winding down this column, I want to point out two other REST-related module types that may be interesting to you. The first takes the sugar part of the last section a wee bit further. There are modules like Rest::Client::Builder that let you inherit from them the capability to build OOP modules. In your module you spend a little time mapping out the API in your code and in return you get to write code that uses the API operations as if they were native calls. This is like the ->CUSTOMER stuff from above only a little cleaner because you've been explicit up front.

The last module I want to show you is some combination of fun and debugging (or maybe debugging fun). The module App::Presto installs a command line tool called "presto" that provides an interactive shell for working with REST services. If you are used to debugging them using CURL, you may find that presto will make your life a little easier. Let's see a couple of quick sessions using it:

```
$ presto http://www.thomas-bayer.com/sqlrest/CUSTOMER/
http://www.thomas-bayer.com/sqlrest/CUSTOMER/> GET 3
{
   "STREET" : "542 Upland Pl.",
   "ID" : "3",
   "FIRSTNAME" : "Michael",
   "CITY" : "San Francisco",
   "xmlns:xlink" : "http://www.w3.org/1999/xlink",
   "LASTNAME" : "Clancy"
}
http://www.thomas-bayer.com/sqlrest/CUSTOMER/> quit
```

Here's a JSON-based API session:

```
$ presto  http://date.jsontest.com
http://date.jsontest.com> GET /
{
   "time" : "04:02:11 AM",
   "milliseconds_since_epoch" : 1417320131828,
   "date" : "11-30-2014"
}
```

Having a tool that lets you walk around an API like this can be mighty handy at times. And with that, let's bring this column to a close. Take care and I'll see you next time.