Practical Perl Tools
Blog, Can We Talk?

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson). He has spent close to 30 years in the system administration/DevOps/SRE field in large multiplatform environments, including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard "g."
dnblankedelman@gmail.com

According to some figures (those at http://w3techs.com/technologies/details/cm-wordpress/all/all to be precise), WordPress powers 24.2% of the sites on the Internet. I don't have any reason to doubt that number. WordPress (WP) has lots of great things going for it when you are looking to bring up a Web site containing dynamic content. Beginners can grasp it fairly quickly, it has a huge and vibrant ecosystem, a strong development effort, oh, and it's free. It's my "goto" tool when someone comes to me and needs my help building a Web site for their aardvark repair company or whatnot. It may have started as blogging software, but it has evolved far beyond that over the years into a reasonable Web development platform.

So why am I giving you a sales pitch for WP in a Perl column? In addition to making my bias clear, I figure if it powers close to a third of the Web sites on the planet, it could be a good idea to learn to interact with it via Perl. Why would you want to do this? For me, the best reasons center around being able to easily extract information posted on a WordPress site or, even better, the ability to post external sources of information right to a WordPress site. For example, let's say you had a process for generating sales reports that took hours of heavy-duty computation on a massive data warehouse. It might be very handy to post the results to an internal WordPress site every day for people to be able to easily access.

The good news is we are going to be able to draw strongly on past columns and knowledge for this effort. One quick prerequisite: I'm going to make the assumption that you have at least a passing familiarity with WordPress (you know it has posts, pages, and users, and you know how to install plugins) and administrative access to a working up-to-date WP site.

Here's What We Are Not Going to Do

There are lots of inelegant ways we could interact with WP (some of which we've explored in this column). For example, we could use something like WWW::Mechanize or Selenium to pretend to be Web browsers to screenscrape the pants off the site or fake like we are typing/clicking. I could make you more nauseous by noting that WordPress has a MySQL backend (plus access to a file system) so we could just whip out DBI and go to town. Nope, not going to do it.

A much more reasonable approach might be to use the closest thing WordPress has had to an external API: the XML-RPC interface it provides via the xmlrpc.php file. And, indeed, there have been modules written in days of yore like WordPress::XMLRPC that use this API. Even though XML-RPC has been around for quite a while, it doesn't seem to get much love or respect from the WordPress community these days. Part of this could be because XML-RPC isn't the simplest of protocols: at the very least you need to understand and know how to manipulate XML. But another large part is likely how incomplete the API support is. It exposes certain WordPress operations, but it omits whole classes of things you might want to do remotely over an API. So what's a better option if we want to stick with the magical three letters "API"?

There are two choices. Once upon a time, Automattic, the commercial entity that runs the WordPress hosting at wordpress.com, made available a JSON-based REST service their customers could use. This was available for wordpress.com, but self-hosted WordPress sites couldn't use it. Later this functionality was added to their kitchen-sink plugin Jetpack (http://jetpack.me), which "supercharges your self-hosted WordPress site with cool functionality from WordPress.com." I've not used Jetpack on any site I've set up, largely because it always seemed a bit heavyweight to me even if it does do a ton of cool stuff simultaneously out of the box. Plus it introduces some dependencies on the wordpress.com backend infrastructure I didn't really want. That takes this option out of the running for me.

The second choice, better in some ways (worse in others, more on that in a moment), is a plugin that provides a similar JSON-based REST API. The later version of the plugin (v2, in beta) is meant to be a reference implementation merged into WordPress core in short order. This means the functionality will eventually be available out of the box without having to install a plugin. I'm not entirely sure if this is still the plan for WordPress roadmap, but the intent to add this to core is a pretty strong indicator of support. That's the good part of this option. There are two aspects that I am less enamored of: v2 of this plugin's implementation is relatively new, so information about installing and using it is much less mature than what is available for v1 (e.g., the API documentation at http://v2.wp-api.org is more a collection of section headings than actual documentation). This leads to lots of peeking back and forth between v1 and v2 docs and more hunting down of arcana/reading of the source than I would prefer. In this column, I will largely try to cut through all of that and provide some more direct instruction. There is, however, one place I'm going to punt on how to do things (my second negative); we'll come to that a little later on.

WP-API Install

Assuming again that you have a functioning and up-to-date WordPress install to work with, let's see how to get the WP-API stuff functional. There are 3–4 steps; let's start with the first two and bring the others in when we need them.

First off, you will want to install and activate the "WordPress REST API (Version 2)" plugin. You can either do this by entering that phrase into the search box in Plugins -> Add New (be sure to get the Version 2 one), or if you want to flex your dev chops, you can change to the wp-content/plugins directory of your WP installation and clone the plugin from its GitHub repo right into place:

```
git clone git://github.com/WP-API/WP-API.git
```

(Be sure to activate the plugin once you've installed it.)

The second step is to confirm you have a compatible permalinks scheme selected (Settings -> Permalinks in the dashboard). Any scheme except for the one listed as "Default" will work. Switch it away from Default to something else and save the change if this is not the case.

To confirm that the installation works, the v1 Getting Started guide (http://wp-api.org/guides/getting-started.html) suggests you can type the following:

```
curl -I {URL of your WP site}
```

The -I tells cURL to make a HEAD request because all we really need to see is the headers this returns. If everything is hunky-dory, you should see something like this:

```
$ curl -I http://local.wordpress.dev
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 30 Jul 2015 03:17:23 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
X-Powered-By: PHP/5.5.9-1ubuntu4.11
X-Pingback: http://local.wordpress.dev/xmlrpc.php
Link: <http://local.wordpress.dev/>; rel=shortlink
Link: <http://local.wordpress.dev/wp-json>; rel="https://
github.com/WP-API/WP-API"
```

The second Link: header we get back above is the key: it shows that WP-API is installed and ready to take requests at the wp-json endpoint. As a quick aside, the examples in this column will all be using a local WordPress install I have on my laptop provided by the Varying Vagrant Vagrants package (https://github.com/Varying-Vagrant-Vagrants/VVV). If you use Vagrant, be sure to check VVV out because it is quite well done.

Now That It's Installed, What Can We Do?

Now that we know it is working, what can we do with it? Let's actually ask it:

```
$ curl http://local.wordpress.dev/wp-json/
{"name":"Local WordPress Dev","description":"Just
another WordPress site","url":"http:\/\/local.wordpress
.dev","namespaces":["wp\/v2"],"authentication":[],"routes":{"
\/":{"namespace":"","methods":["GET"],"_links":{"self":"http:
\/\/local.wordpress.dev\/wp-json\/"}},"\/wp\/v2":{"namespace":
"wp\/v2","methods":["GET"],"_links":{"self":"http:\/\/local
.wordpress.dev\/wp-json\/wp\/v2"}},"\/wp\/v2\/posts":
{"namespace":"wp\/v2","methods":["GET","POST"],"_links":
{"self":"http:\/\/local.wordpress.dev\/wp-json\/wp\/v2\/
posts"}},"\/wp\/v2\/posts\/(?P<id>[\d]+)":{"namespace":"wp\/v2"
,"methods":["GET","POST","PUT","PATCH","DELETE"]},"\/wp\/v2\/
posts\/schema":{"namespace":"wp\/v2","methods":["GET"],"
_links":{"self":"http:\/\/local.wordpress.dev\/wp-json\/
wp\/v2\/posts\/schema"}},"\/wp\/v2\/posts\/{parent_id}\/
meta":{"namespace":"wp\/v2","methods":["GET","POST"]},"\/
wp\/v2\/posts\/{parent_id}\/meta\/{id}":{"namespace":"wp\/
v2","methods":["GET","POST","PUT","PATCH","DELETE"]},"\/
wp\/v2\/posts\/meta\/schema":{"namespace":"wp\/
v2","methods":["GET"],"_links":{"self":"http:\/\/local
.wordpress.dev\/wp-json\/wp\/v2\/posts\/meta\/schema"}},
...
```

Note: I could have made this request via Perl (perhaps used GET from the LWP::Simple package, HTTP::Tiny, or any of the modules we've discussed in the past for this sort of thing) but cURL was already in my shell history.

Egads, that's one big blob of JSON we get back (I cut it off at an arbitrary point; the whole thing is 6443 characters total). It is kind of hard to read, so let's run it through a JSON pretty-printer to make it more legible. Again, we could write some Perl code to parse and pretty print, but in command-line cases like this, I tend to use one of two really great JSON tools: underscore-cli (https://github.com/ddopson/underscore-cli) or jq (http://stedolan.github.io/jq/). Both are excellent, so if you haven't encountered them before, I highly recommend you go check them out. Let's run that last request through jq (and show an excerpt from the reply):

```
$ curl -s http://local.wordpress.dev/wp-json/|jq .
{
  "name": "Local WordPress Dev",
  "description": "Just another WordPress site",
  "url": "http://local.wordpress.dev",
  "namespaces": [
    "wp/v2"
  ],
  "authentication": [],
  "routes": {
    "/": {
      "namespace": "",
      "methods": [
        "GET"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/"
      }
    },
    "/wp/v2": {
      "namespace": "wp/v2",
      "methods": [
        "GET"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2"
      }
    },
    "/wp/v2/posts": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2/posts"
      }
    },
    "/wp/v2/posts/{id}": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST",
        "PUT",
        "PATCH",
        "DELETE"
      ]
    },
...
   "/wp/v2/users": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST"
      ],
      "_links": {
        "self": "http://local.wordpress.dev/wp-json/wp/v2/users"
      }
    },
    "/wp/v2/users/{id}": {
      "namespace": "wp/v2",
      "methods": [
        "GET",
        "POST",
        "PUT",
        "PATCH",
        "DELETE"
      ]
    },
...
```

Let's take a closer look at some of this output. Specifically, I want to draw your attention first to the info it printed regarding the route available to query post info:

```
"/wp/v2/posts": {
  "namespace": "wp/v2",
  "methods": [
    "GET",
    "POST"
  ],
  "_links": {
    "self": "http://local.wordpress.dev/wp-json/wp/v2/posts"
  }
},
"/wp/v2/posts/{id}": {
  "namespace": "wp/v2",
  "methods": [
    "GET",
    "POST",
    "PUT",
    "PATCH",
    "DELETE"
  ]
},
```

This says I can either make a GET or a POST request for http://local.wordpress.dev/wp-json/wp/v2/posts to read or change the list of posts on the site. If I want to address an individual post (to GET, submit a new one with POST, DELETE it, and so on), I can do so at the same URL with the ID for that post tacked on to the path. This pattern repeats itself in the previous output for users, so we now know how to with users of the system. Let's try to get the list of users on the site:

```
$ curl -s http://local.wordpress.dev/wp-json/wp/v2/users|jq .
[
  {
    "code": "rest_forbidden",
    "message": "You don't have permission to do this.",
    "data": {
      "status": 403
    }
  }
]
```

Whoops, that didn't work—and good thing too! We really don't want anyone with cURL to be able to pull a list of users. That leads to the second part of the WP-API install/setup and a bit of a screed.

WP-API Authentication

In order for authentication of any type to work, there has to be an existing user defined on your site that you will authenticate to do the work. If you plan to query information that only an admin-level user should have access to (e.g., a list of site users), this user will have to be created as an admin. If you don't need that level of access from the API, I encourage you to create a user at a lower role or just send unauthenticated requests for publicly viewable information. New users for WP-API are created using the normal WordPress process (Users -> Add New). For this column, I created an admin user with the user name "api" and the password "api" (yup, security by alliteration, yay!) on my local test site.

WP-API has two contexts it operates in, one I'll call "internal," where code running on the site (e.g., a PHP-based WordPress theme), the other "external" (some outside code calls the API remotely). We're going to totally ignore the former and only look at the external context. In this context, there are two, maybe three mechanisms for authentication.

The first is the least secure one and is only recommended for development and testing. This is using the HTTP Basic Authentication found in RFC 2617. To use this, you need to git clone the Basic Authentication plugin into place as we did earlier:

```
$ cd wp-content/plugins
$ git clone git://github.com/WP-API/Basic-Auth.git
```

and then activate the plugin in the dashboard.

The second and third options are to use OAuth. OAuth is a mildly complicated protocol that comes in two incompatible versions (1.0a and 2.0) and that is designed to allow a third-party client to be given permissions to act on the behalf of a user. So, for example, if you install a new Twitter or Gmail client, it is likely that the first thing it will do is ask you to authenticate to those services and then permit that client to act on your behalf to perform certain operations (post tweets, manipulate your mail, etc.). This is OAuth in action.

Here's where it starts to get tricky and we quickly descend down a rabbit hole. The WP-API docs suggest that you install an OAuth 1.0a plugin from GitHub ("git clone git@github.com:WP-API/OAuth1.git content/plugins/oauth-server"; see https://github.com/WP-API/OAuth1) and use that for authentication. It is suggested that this plugin will also eventually be incorporated into WP core. Ordinarily at this point in the column, we'd go off and talk about how OAuth works and how to work with it in Perl. I won't be doing that here for two reasons:

1. The protocol/framework is a wee bit complicated and needs a little bit of explanation before you can dive into using it, and I don't have the space.
2. I'm annoyed that WP-API's suggested plugin implements 1.0a of OAuth and not 2.0. As far as I can tell, there isn't a major service provider using 1.0a instead of 2.0, so the value of going deeper into a barely used protocol is unclear to me. Some say that the older version was a stronger protocol, but I'm not sure that pragmatically justifies the column space.

Now let me make things even more interesting. There exists a commercial plugin (or at least one that would like to charge licensing fees) that implements OAuth2. It can be found at https://wp-oauth.com. It claims to support WP-API (at least in part of the doc, while in another part it claims it doesn't, sigh). I'm also not clear whether it supports the 2.0 WP-API beta version either. Because OAuth2 leans on SSL/TLS for some of its security, you would want that set up on your site before truly using it. I have yet to test it.

Given these complications, I'm going to punt on the more secure methods (even though I know it means that somewhere an angel isn't going to get its wings) and just go with Basic Authentication in our examples. Just so you don't feel I'm hanging you out to dry, I will mention that the Net::OAuth and Net::OAuth2 modules (plus a couple others like OAuth::Lite) do exist, so you can definitely perform OAuth operations (from both protocol versions) from Perl. If you'd like to see another column about just OAuth, please drop me a line and I will see about writing one.

To review as we leave the rabbit hole: to use WP-API operations of a certain level, you need a suitably empowered WordPress user and a way to authenticate as that user installed. We'll be using the Basic Authentication plugin for the latter (boo hiss).

Perl Time

In a previous column about using REST interfaces from Perl, we tiptoed up to using Perl modules that provided lots of "do what I mean" syntactical sugar. In this column, I'm just going to put the pedal to the metal and go right for using that kind of module.

Let's start off with getting the list of pages on a site. Our first attempt to write code to this would probably look a bit like this:

```
use WebService::CRUST;

my $w = new WebService::CRUST(
   base       => 'http://local.wordpress.dev/wp-json/wp/v2/',
   format     => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

# this is the equivalent of
# $w->get('pages');
# we could also write $w->pages;
# yummy syntactic sugar!
my $result = $w->get_pages;

print "Total items: " .
  $result->crust->{response}->{_headers}->{'x-wp-total'},
  "\n";

foreach my $page ( @{ $result->result } ) {
      print $page->{'id'} . ':' .
            $page->{'title'}->{'rendered'} .
            " (" . $page->{'link'} . ")\n";
}
```

The code creates a WebService::CRUST object and tells it that all of our requests are going to start with that URL. It also specifies that we will want to use JSON::XS (the faster JSON parser) to decode the responses we get back. The next step is to query for all of the pages on the system. As you can see in the comments, WebService::CRUST allows us to write code that makes it look like pages() or get_pages() is a real method call. This is one of the things I like about this module: it makes for very readable code, even if it is doing a bit of autoload magic behind the scenes.

For fun (or actually, for foreshadowing), we reach deep into the WebService::CRUST::Response object using the crust method to pull out one of the headers WordPress sends us in response to our query (X-WordPress-Total, which gets downcased when stored in the object). This header provides the number of items we should expect back from our query. Then we proceed to iterate over the response we got back in that WebService::CRUST::Response object (via the result method) to print out the ID, title, and the URL for each page on the system. Here are the results on my local test instance (which I've preloaded with a bunch of example pages):

```
Total items: 248
2:Sample Page (http://local.wordpress.dev/sample-page/)
5434:2008 Festival (http://local.wordpress.dev/archive
/2008-festival/)
5433:2007 Festival (http://local.wordpress.dev/archive
/2007-festival/)
5432:2006 Festival (http://local.wordpress.dev/archive
/2006-festival/)
5409:2012 Festival (http://local.wordpress.dev/archive
/2012-festival/)
5407:2011 Festival (http://local.wordpress.dev/archive
/2011-festival/)
5405:2010 Festival (http://local.wordpress.dev/archive
/2010-festival/)
5403:2009 Festival (http://local.wordpress.dev/archive
/2009-festival/)
```

```
5305:Pick-Up Band 2014 (http://local.wordpress.dev/archive
/2014-festival/pick-up-band-2014/)
5280:Saturday Schedule by Location (2014) (http://local
.wordpress.dev/archive/2014-festival/saturday-in-davis-square
/saturday-schedule-by-location-2014/)
```

Hey, wait a second, something is wrong. WordPress says there are 248 pages on the system, but it has only returned 10. Welcome to the world of pagination. Perhaps showing its blogging roots, WordPress wants to hand back the reply one "page" at a time. This isn't entirely out of the ordinary because other servers (e.g., LDAP servers) often have a max size for data returned that you can only deal with by requesting a chunk at a time. We could try to get around this pagination by asking WordPress to create pages that are big enough to hold all of the items or even turn off pagination, but I think it is better to work within the system than try to hack around it.

So how do we get more pages past the first one? If we were to peek more closely at what was returned from our request, we would notice that WordPress has sent us a "link" header (remember that from the beginning of the column?). Here's what it looks like from the request above (it is all one long line):

```
'http://local.wordpress.dev/wp-json/wp/v2/pages?page=
2>; rel="next"'
```

That is the URL we will have to request to get the next set of results (i.e., the next page). We'll need to write code that parses this header and extracts the next page number, then repeats the request. Here's what that code looks like:

```
use WebService::CRUST;

my $w = new WebService::CRUST(
   base     => 'http://local.wordpress.dev/wp-json/wp/v2/',
   format   => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

my $nextpage = 1;

my $result = $w->get_pages( 'page' => $nextpage, );

print "Total items: " .
     $result->crust->{response}->{_headers}->{'x-wp-total'},
     "\n";
print "Total pages of content: " .
      $result->crust->{response}->{_headers}->{'x-wp-totalpages'},
      "\n";

while ( defined $result and $nextpage ) {
   foreach my $page ( @{ $result->result } ) {
      print $page->{'id'} . ':' .
           $page->{'title'}->{'rendered'} .
           " (" . $page->{'link'} . ")\n";
   }

   ($nextpage) =
    $result->crust->{response}->{_headers}->{'link'} =~
    /\?page=(\d+)>; rel="next"/;

   last unless ( defined $nextpage );

   $result = $w->get_pages( 'page' => $nextpage, );
}
```

Let's focus for a moment on how this differs from the previous code. WordPress is willing to tell us how many pages it will take to provide the entire result set, so I print that for informational purposes. For the real work, our get_pages requests now take an argument that is the parameter and the value to be sent with that request. Adding this argument means we'll be requesting:

```
http://local.wordpress.dev/wp-json/wp/v2/pages?page=N
```

where N is the value of $nextpage. We print the information returned for that page, determine if there are more pages (as specified in the link header), and if so, we perform another request for the next page. As a quick aside, we could have taken the number of pages returned in the X-WP-TotalPages header and iterate from page 1 to that value, but I believe it is less likely to cause a race condition if we work from the "here's the next page" info we get back on each query instead.

This is the basic pattern for most things we can pull back from the API. For example, if we wanted a list of users:

```
use WebService::CRUST;
my $w = new WebService::CRUST(
   base     => 'http://local.wordpress.dev/wp-json/wp/v2/',
   basic_username => 'api',
   basic_password => 'api',
   format   => [ 'JSON::XS', 'decode', 'encode', 'decode' ],
);

my $nextpage = 1;

my $result = $w->get_users( 'page' => $nextpage, );

while ( defined $result and $nextpage ) {
   foreach my $user ( @{ $result->result } ) {
      print $user->{'id'} . ':' . $user->{'name'} . "\n";
   }

   ($nextpage) =
    $result->crust->{response}->{_headers}->{'link'} =~
    /\?page=(\d+)>; rel="next"/;

   last unless ( defined $nextpage );

   $result = $w->get_users( 'page' => $nextpage, );
}
```

Almost identical, yes? The only differences are that we add some initial parameters to send along authentication with every request (in an insecure manner, don't rub it in) and later on pull out different fields from the returned information.

Where Do We Go from Here?

We're almost out of room, but there are a few important things to mention. First off, the examples we've seen here all pull a collection of items (pages, users, etc.). If we want to retrieve a single item, we can reference that item as a part of the path we request by appending the ID we need—for example:

```
    $result = $w->get( "pages/$id" );
```

Second, we've only seen examples that retrieve content. If we want to create or modify content on the site, we use the REST idea of using other HTTP operations as verbs. Want to create a new page or edit a page? Perform a PUT request (details found in the v1 documentation) with the right parameters specified as arguments to the put() method.

And lastly, one more advanced topic we didn't discuss is how to use more of the power of WordPress in our interactions. v1 of the API had a working "filter" parameter which allowed you to pass in a specification the WordPress WP_Query class could work with. This means that you could say to WordPress "return all of the posts by this author" or "only return a list of publicly published posts." I had difficulty using this facility with the v2 API because I believe it is still very much a work in progress as of this writing. Hopefully, this facility will be up to snuff by the time you read this.

In the meantime, enjoy, and I'll see you next time.