# Practical Perl Tools

## With Just a Little Bit of a Swagger

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 systems 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

I've already come out as an API-phile in this column, so I suspect no one will be shocked that we're going to dive into yet another API-related topic this column. Just like the TV show where you recognize everyone in the bar (and they all know your name), we'll be back among old friends like REST and JSON. The one thing I perhaps should provide a trigger warning for is we're going to be mentioning Java in the column. If that's not your cup of you know what, then you may want to skip ahead in the magazine. If it is any comfort, we won't see any actual Java code in the column, just a bit of the tooling.

### Why Are APIs Important?

Even though this column is a bit of a drinking game where every time I say "API," you drink, I don't think we've ever discussed why APIs are important. A (good) API can be seen as a contract between the person who is writing the code to provide a service and the person who is writing the code to consume that service. This is true even if that turns out to be the same person, because all you need is a little time to pass for it to be easy to forget just how two components were supposed to work together. Essentially the contract says, "If you send me a request of this form, I promise to respond (ideally with the data that was requested) in a way that you will expect." It helps to ensure the principle of least surprise, leading to (more) stable and reliable software. An API also encourages software authors to think up front about how pieces of a system should interact. I say "encourages" just because we have all dealt with an API at one time or another that wasn't as well defined or thought out as we might like.

APIs also make it possible to write "loosely coupled" components that interact only through their API, à la the microservices concept that is all the rage. I won't go into more detail here about why loosely coupled services make for a better system, but if you haven't heard that gospel yet, be sure to take some time to look up "microservices." I joke, but this idea is super serious. If you haven't read Steve Yegge's post [1] that included Jeff Bezos' big mandate about APIs, be sure to do so.

And finally, in an ideal world, part of creating a good API is the process of documenting it well. A well-documented API makes things better for everyone (the people who wrote it, the people who use it, the people who are thinking about using it, people who want to learn from it, and so on). And this is where Swagger comes in.

### And Now We Swagger

Here's how the official Web site [2] defines it:

> The goal of Swagger™ is to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined via Swagger, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interfaces have done for lower-level programming, Swagger removes the guesswork in calling the service.

> Technically speaking - Swagger is a formal specification surrounded by a large ecosystem of tools, which includes everything from front-end user interfaces, low-level code libraries and commercial API management solutions.

So what does this standard look like? At the moment, you can write Swagger specifications in either JSON (the original format) or YAML (recently added). To get a sense of what it actually looks like, here's the Hello World-ish sample in YAML format from the "Getting Started with Swagger—What Is Swagger?" article on the official Web site:

```
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting to the user!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name of the user to greet.
      responses:
        200:
          description: Returns the greeting.
          schema:
            type: string
        400:
          description: Invalid characters in "user" were provided.
```

This defines a REST interface that has exactly one endpoint, `/hello/{user}` (as in /hello/rik). The username at the end is defined to be a required string. If a valid username was given, the API promises to return a 200 return code (success) followed by a greeting to that user (in string form). If there is a problem with the username, an error code (400) is returned.

This by itself, besides being a reasonable documentation format, isn't the cool part. The cool part is when you bring the tools written around the Swagger specification into the picture. Let's quickly mention the non-Perl-related tools and then take a look at how Swagger plays in the Perl space.

Two of the more interesting non-Perl tools are Swagger Editor (running sample at http://editor.swagger.io/) and Swagger UI (running sample at http://petstore.swagger.io/). Swagger Editor lets you compose your YAML or JSON in real time and see how it will look as a fully formatted (and purdy, they've done a nice job with the design) documentation page generated on the fly. The editor also has some options for code generation—more on that in a moment.

Equally interesting is the Swagger UI tool, which generates a Web application that lets people not only read the documentation, but try API calls right from the documentation page. If you've ever tried something like Google's API Explorer or Spotify's API Console [3] you'll have a sense of what Swagger UI provides. And if you haven't, you really should because they are both very useful tools.

## Generating Code

So now we step closer to the promise of Perl code. It's cool that we now have a good format for specifying our REST API. It is even cooler that we can process that specification and produce good-looking (and even interactive) documentation. But even better would be to run that specification through a post-processor that actually writes the code for us to make use of the specification. Why is this cool? It means that your API documentation and your API code won't get out of sync, because one begets the other. As an aside before we go deeper into this: I have seen efforts that allow people to take existing code and generate a Swagger spec (i.e., go the other way). I think it is cleaner to write the doc first, but I can see how going in the opposite direction could be beneficial in certain cases.

There are two kinds of code we could think about generating: client and server. We'll look at both separately. If we are continuing our look at "official" tools, we should start with Swagger Codegen (http://swagger.io/swagger-codegen/). Swagger Codegen is primarily meant to produce client code in a wide range of languages/frameworks from a Swagger spec. It manages this by making it relatively easy to add your own modules/templates.

At the moment, it knows how to output clients using these languages/frameworks:

```
[
  "android",
  "async-scala",
  "csharp",
  "dart",
  "flash",
  "java",
  "objc",
  "perl",
  "php",
  "python",
  "qt5cpp",
  "ruby",
  "scala",
  "dynamic-html",
    "html",
    "swagger",
    "swagger-yaml",
    "swift",
    "tizen",
    "typescript-angular",
    "typescript-node",
    "akka-scala",
    "CsharpDotNet2"
  ]
```

This is output from the online Codegen tool at https://generator.swagger.io, essentially a pretty-printed version of the output of:

```
curl -X GET --header "Accept: application/json"
  "https://generator.swagger.io/api/gen/clients"
```

To use Swagger Codegen, you have to install a particular version of Java (7 as of this writing), Apache maven, and the tool itself. I used homebrew on my Mac to install all of these components, including Java. Java 7 gets installed in a homebrew-specific place via its Cask mechanism since downloading that version from Oracle's Web site isn't easy. All in all, the process of bringing up the necessary Java toolchain wasn't as painful as I expected, but your mileage may vary.

Once you have everything installed, you can process a Swagger specification. Swagger ships with sample specs (for example, one based on an API for a pet store because, um, every modern pet store needs an API, I guess?) and scripts that process them to generate sample code for each language. Rather than using that sample spec, let's stick to the simpler "hello world" example shown earlier. To process it, we might type something like:

```
$ swagger-codegen generate -i ./test.yaml -l perl -o perl-test
```

The output will look something like this:

```
reading from ./test.yaml
[main] INFO io.swagger.codegen.DefaultCodegen -
        generated operationId helloUserGet
        for Path: get /hello/{user}
writing file /tmp/perl-test/lib/WWW/SwaggerClient/DefaultApi.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/ApiClient.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/
        Configuration.pm
writing file /tmp/perl-test/lib/WWW/SwaggerClient/Object
        /BaseObject.pm
```

As you can see, four separate files have been generated to form a module we can use (WWW::SwaggerClient). Of these, three are "support" files and one has the code specific to the defined REST API. That info is in DefaultApi.pm. In it we find the following code (slightly reformatted to fit on the page):

```
#
# hello_user_get
#
#
#
# @param string $user The name of the user to greet. (required)
# @return string
#
 sub hello_user_get {
   my ($self, %args) = @_;

   # verify the required parameter 'user' is set
   unless (exists $args{'user'}) {
     croak("Missing the required parameter 'user' when calling
         hello_user_get");
   }

   # parse inputs
   my $_resource_path ='/hello/{user}';
   # default format to json
   $_resource_path =~ s/{format}/json/;

   my $_method ='GET';
   my $query_params = {};
   my $header_params = {};
   my $form_params = {};

   # 'Accept' and 'Content-Type' header
   my $_header_accept =
    $self->{api_client}->select_header_accept();
   if ($_header_accept) {
      $header_params->{'Accept'} = $_header_accept;
   }
   $header_params->{'Content-Type'} =
     $self->{api_client}->select_header_content_type();

   # path params
   if ( exists $args{'user'}) {
      my $_base_variable = "{" . "user" . "}";
      my $_base_value =
         $self->{api_client}->to_path_value($args{'user'});
      $_resource_path =~ s/$_base_variable/$_base_value/g;
   }


   my $_body_data;


   # authentication setting, if any
   my $auth_settings = [];

   # make the API Call
   my $response =
     $self->{api_client}->call_api($_resource_path,
                                   $_method,
                                   $query_params,
                                   $form_params,
                                   $header_params,
                                   $_body_data,
                                   $auth_settings);
    if (!$response) {
       return;
    }
    my $_response_object =
       $self->{api_client}->deserialize('string', $response);
    return $_response_object;


 }
```

This code is a little gnarly (as are the other files). The generated code is meant to handle more sophisticated specs, so it looks a bit like overkill at first glance. It definitely represents a certain set of opinions and programming choices of the template author. The generated code includes a bunch of Perl modules you may or may not have installed (e.g., Log::Any), so be prepared to work a bit if you are going to use the code right out of the box.

Given all of this, let me highlight one small part of the code above. In it you can see that it has defined a hello_user_get subroutine. This is the one you are going to call as a method to perform the actual call from the Swagger spec. To use all of this, we would write code like this:

```
use lib 'perl-test/lib';
use WWW::SwaggerClient::DefaultApi;
''
my $api = WWW::SwaggerClient::DefaultApi->new();

my $greet = $api->hello_user_get('user' =>'rik' );
```

If I just run this code from my laptop without any other preparation, I get the following error:

```
API Exception(500): Can't connect to localhost:443 at
perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100.
```

because there is nothing currently listening on my laptop for connections from a client (i.e., no server). If I ran this under the debugger, I'd see more detail about what was being attempted (here's the key line of the output):

```
API Exception(500): Can't connect to localhost:443 at
perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100.
 at perl-test/lib/WWW/SwaggerClient/ApiClient.pm line 124.
WWW::SwaggerClient::ApiClient::call_api(WWW::SwaggerClient::
 ApiClient=HASH(0x7fc1339f7d98), "/hello/rik", "GET",
 HASH(0x7fc133a76fb0), HASH(0x7fc133a76f98),
 HASH(0x7fc133a76fc8), undef, ARRAY(0x7fc1320083c0))
 called at perl-test/lib/WWW/SwaggerClient/DefaultApi.pm line 100
```

From this line you can see that it was going to attempt to connect to a server and issue a GET request for the path /hello/rik just as we'd hoped it would. If you'd like to see a more sophisticated example, I recommend you dissect the sample apps that come with Swagger Codegen (e.g., the pet store one) to see how it works. If the generated code isn't to your liking, you may want to consider creating a custom plugin that outputs the kind of code you seek.

Another possibility is to use the module we are about to see for server code: Swagger2. Swagger2 ships with a Swagger2::Client module, which lets you write code that looks like this:

```
use Swagger2::Client;

$ua = Swagger2::Client->generate("/tmp/test.yaml");
$ua->base_url->host("other.server.com");

# yes, this is ugly. If our spec had an operationId parameter,
# the name of the method would be based on it instead
$ua->_hello__user_({'user'=>'rik'})
```

But let's move away from the client code possibilities and think a little bit about the server side of things instead. Swagger Codegen has limited support for server code (e.g., it can create server stubs for NodeJS, Python Flask, Ruby Sinatra, and so on) but nothing for Perl-based servers. For that we're going to have to go a little further off the ranch and use the Swagger2 module.

Probably the easiest path is to use Mojolicious::Plugin::Swagger2, which ships with the Swagger2 Perl module. With this plugin, we can use the Mojolicious Web framework we've seen in past columns. If you add code like this to the startup routine of your Web app:

```
$app->plugin(Swagger2 =>
          {url => "file:///path/to/test.yaml"});
```

it will automatically add routes and validation to your Web app (providing it has operationId info in the spec). The paths defined in the Swagger spec will automatically become routes that require the parameters mentioned in the spec. There's a lovely example of how this works in the author's tutorial on his blog [4]. Swagger2 with Mojolicious isn't the only game in town for Swagger (for example, there is the REST API framework "raisin" that also integrates with Swagger), but I think it is a lovely combination.

So with that, I think you've got at least a small peek at Swagger and how it can improve your API life. Take care, and I'll see you next time.

***Resources***

[1] Steve Yegge's Google Platform Rant: https://plus.google.com/+RipRowan/posts/eVeouesvaVX.

[2] Swagger: http://swagger.io.

[3] Google's API explorer: https://developers.google.com/apis-explorer; Spotify's API Console: https://developer.spotify.com/web-api/console/.

[4] Mojolicious Swagger2 tutorial: http://thorsen.pm/perl/programming/2015/07/05/mojolicious-swagger2.html.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

## USENIX Awards

USENIX honors members of the community with two prestigious awards which recognize public service and technical excellence:

- **The USENIX Lifetime Achievement (Flame) Award**
- **The LISA Award for Outstanding Achievement in System Administration**

The winners of these awards are selected by the USENIX Awards Committee. The USENIX membership may submit nominations for either or both of the awards to the committee.

### The USENIX Lifetime Achievement (Flame) Award

The USENIX Lifetime Achievement Award recognizes and celebrates singular contributions to the UNIX community in both intellectual achievement and service that are not recognized in any other forum. The award itself is in the form of an original glass sculpture called "The Flame," and in the case of a team based at a single place, a plaque for the team office.

Details and a list of past recipients are available at www.usenix.org/about/flame.

### The LISA Award for Outstanding Achievement in System Administration

This award goes to someone whose professional contributions to the system administration community over a number of years merit special recognition.

Details and a list of past recipients are available at www.usenix.org/lisa/awards/outstanding.

### Call for Award Nominations

USENIX requests nominations for these two awards; they may be from any member of the community. Nominations should be sent to the Chair of the Awards Committee via awards@usenix.org at any time. A nomination should include:

1. Name and contact information of the person making the nomination
2. Name(s) and contact information of the nominee(s)
3. A citation, approximately 100 words long
4. A statement, at most one page long, on why the candidate(s) should receive the award
5. Between two and four supporting letters, no longer than one page each