Practical Perl Tools
Perl to the Music

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

Given all of the APIs and their Perl interactions we've discussed in this column, it is a little surprising it has taken me this long to get around to one of my favorite subjects: music. I started to pay closer attention to the APIs of the streaming music services right around the time one of my favorites (Rdio) was shuttered. I had amassed a pretty large collection of artists and albums I wanted to replicate on another service and was concerned about extracting the info from their service before it closed down. Luckily, the engineers at Rdio were equally concerned for their users and did a superb job of providing each user with an export of their data. But that started me down the path of wondering just what I could do in Perl to interact with my music data.

Several of the major streaming services have a decent API. Here's the rundown as of this writing:

- Spotify: https://developer.spotify.com
- Rhapsody: https://developer.rhapsody.com
- Deezer: http://developers.deezer.com/api
- Pandora: (only official partners can use it)
- Tidal: none
- Guevara: none
- Apple Music: none
- Google Play Music: none (really, Google? No API?)

Some of the services above have "unofficial APIs" where a random developer has reverse-engineered how the service works and published code that uses that information. We're not going to touch any of those APIs with a very large pole for any number of reasons, most of which I bet you can guess. Instead, in this column we'll pick the top one of the list above, Spotify, and dive into it. Spotify has a particularly mature API. Note: for some of these examples, you will need a Spotify account (and indeed, may need to be a subscriber).

Do I Know Who You Are?

The Spotify API distinguishes between authorized requests and public, non-authorized requests. The former is less rate-limited than the latter and (depending on the kind of authorization) also allows for querying of more sensitive data. But we can still get some good stuff from the API using public requests, so let's start there before we get into the auth game.

Given all of the past columns on APIs, I expect no gasps of astonishment when I say that the API is REST-based and that the results returned are in JSON format. The one Spotify API-specific piece we haven't really seen before (but is actually pretty common) is that Spotify has a unique resource identifier and ID for pointing to a specific object in their system. So, for example, here in their docs list:

```
spotify:track:6rqhFgbbKwnb9MLmUQDhG6
```

is a Spotify URI and

```
6rqhFgbbKwnb9MLmUQDhG6
```

is a Spotify ID. The URI includes what kind of thing is being referenced, the ID just simply provides which specific thing (i.e., that track) is being referenced. And in case you were curious, that URI in their doc points to the first track of an iconic album. I'll let you paste it in to the Spotify client to see just which one.

Dither, Dither, Dither

Right about now in the column the hero has a small crisis where he frets about which Perl module/approach he should use. Should he build something up using the minimalist but elegant modules that only do one basic thing really well (like performing an HTTP request or parsing JSON)? Should he instead use the all-singing, all-dancing REST request module that does both of these things and four other things besides? Perhaps he should use the module specifically made for this Web service. Or maybe show all three? Decisions, decisions.

It may shorten this column, but let's go right for the purpose-built module this time: WebService::Spotify. I'll explain this decision in more depth (complete with a dash of dithering) when we come back to authenticated/authorized requests. This module appears to be the most up-to-date (there is a module available called WWW::Spotify, but it calls the deprecated Spotify API). The difference between WebService::Spotify and something more lightweight becomes apparent when you install it. Because it is using Mouse (the smaller version of the modern object-oriented framework called Moose), it requires a whole slew of dependencies. cpanminus or CPANPLUS (discussed in a previous column) will handle this for you, but it can still be a bit disconcerting to watch the module names scroll by when you install it.

Once installed, using the module for non-authorized API calls is super simple:

```
use WebService::Spotify;

my $s = WebService::Spotify->new;

my $r = $s->search( 'chloe', type => 'artist' );

foreach my $artist ( @{ $r->{artists}->{items} } ) {
   print "$artist->{name} ($artist->{uri})\n";
}

print "total=$r->{artists}->{total}\n";
print "previous=$r->{artists}->{previous}\n";
print "next=$r->{artists}->{next}";
print "\nlimit=$r->{artists}->{limit}\n";
print "offset=$r->{artists}->{offset}\n";
```

Here's the output when I run it (which we will explain in a moment):

```
Chloe Angelides (spotify:artist:79A4RmgwxYGIDkqQDUHLXK)
Chloe (spotify:artist:71P0UphzXd95FKPipXjtEO)
Chloë (spotify:artist:2pCYsqZMqjA345dkjNXEct)
Chloe Martini (spotify:artist:6vhgsnZ2dLDaLDog3pqP2d)
Chloe Agnew (spotify:artist:34sL9HIOU5Ot8uOIQMZeze)
Chløë Black (spotify:artist:OIfnpflOVEmRGxCKaCYPX4)
Chloe (spotify:artist:2hgOg48H7GvAlTzkt3z5Vo)
Chloe Kaul (spotify:artist:35BBadnzA39iYkbQWLOr3p)
Chlöe Howl (spotify:artist:1hvPdvTeY6McdTvN4DyKGe)
Chloe Dolandis (spotify:artist:2SfamMWWSDbMcGpSuaO6o4)
total=340
previous=
next=https://api.spotify.com/v1/search?query=chloe&offset
=10&limit=10&type=artist
limit=10
offset=0
```

The code creates a new object, executes a search, and then prints key parts of the response. The response comes back in JSON form that is then parsed into Perl data structures. Here are some excerpts from a dump of that data structure:

```
0  HASH(0x7fe70ca94dc8)
  'artists' => HASH(0x7fe70ca94b88)
    'href' => 'https://api.spotify.com/v1/search?
query=chloe&offset=0&limit=10&type=artist'
    'items' => ARRAY(0x7fe70b003718)
...
        1  HASH(0x7fe70e129500)
         'external_urls' => HASH(0x7fe70e1157c8)
           'spotify' => 'https://open.spotify.com/artist/
                71P0UphzXd95FKPipXjtEO'
         'followers' => HASH(0x7fe70e115138)
           'href' => undef
           'total' => 38
         'genres' => ARRAY(0x7fe70e129a40)
            empty array
         'href' => 'https://api.spotify.com/v1/artists/
                71P0UphzXd95FKPipXjtEO'
         'id' => '71P0UphzXd95FKPipXjtEO'
         'images' => ARRAY(0x7fe70e0d6780)
           0  HASH(0x7fe70e0d67c8)
             'height' => 640
             'url' => 'https://i.scdn.co/image/20620033bdf1
               c86e83cb3f18f97172aa89ee6eca'
             'width' => 640
           1  HASH(0x7fe70e42e388)
             'height' => 300
             'url' => 'https://i.scdn.co/image/5c0a9f8cb3bb
               f55e15c305720d6033090c04c136'
             'width' => 300
           2  HASH(0x7fe70a7646e0)
             'height' => 64
             'url' => 'https://i.scdn.co/image/767603633a02
```

```
                6e73551c6c98d366b8cf08a1adec'
              'width' => 64
          'name' => 'Chloe'
          'popularity' => 39
          'type' => 'artist'
          'uri' => 'spotify:artist:71P0UphzXd95FKPipXjtE0'
...
       'limit' => 10
     'next' => 'https://api.spotify.com/v1/search?query=chloe
           &offset=10&limit=10&type=artist'
     'offset' => 0
     'previous' => undef
     'total' => 331
```

We get back a list of artists, each with their own sub-data structure (a list of hashes). Each artist that is returned has a number of fields. Our script prints out just the name and the URL, but you can see there's lots of interesting stuff here, including pointers to images for the artist. Already we are having some fun.

In addition to the list of artists, we also get back some data about the query itself, which I thought was so important to discuss, I print it explicitly in the script. This data helps us paginate through the large quantity of info in Spotify's database as needed. The "total" field tells us there are actually 331 artists with this name, but by default the module asks the API to limit the output to sending back 10 records at a time. This query started at the beginning of the data set (an offset of 0), and there is nothing before it (previous is "undef"). There is, however, the URI we should be querying to get the next 10 records (which includes an offset of 10, and an explicit limit of 10).

If we weren't using the WebService::Spotify module, we would call that URI to get the next set of 10. To do this explicitly with the module, we could add an "offset" parameter to the search() method like this:

```
my $r = $s->search( 'chloe', type => 'artist', offset => 10);
```

but WebService::Spotify makes it even easier by providing a next() method that consumes a results object and "does the right thing," as in:

```
my $s = WebService::Spotify->new;

my $r = $s->search( 'chloe', type => 'artist' );

while ( defined $r->{artists}->{next} ) {
   print_artists($r);
   $r = $s->next($r->{artists});
}

sub print_artists {
   my $r = shift;
   foreach my $artist ( @{ $r->{artists}->{items} } ) {
      print "$artist->{name} ($artist->{uri})\n";
   }
}
```

This code pages through the data using next() to pull the next result set if there is any.

So now that we've seen how to query for artists with a particular name, what can we do with that artist's info? As a start, with the artist ID, we can look up that artist's albums and her or his top tracks in a particular country:

```
# same search as before, let's pick the second result
my $artist = $r->{artists}->{items}->[1];
my $albums = $s->artist_albums( $artist->{id} );
my $top_tracks = $s->artist_top_tracks( $artist->{id},
                    'country' => 'US' );
```

Here's a small excerpt from the $albums data structure:

```
0  HASH(0x7fea614a86a8)
  'href' => 'https://api.spotify.com/v1/artists/
    71P0UphzXd95FKPipXjtE0/albums?
    offset=0&limit=20&album_type=single,album,compilation
,appears_on,ep'
  'items' => ARRAY(0x7fea614a8780)
    0  HASH(0x7fea613279d8)
      'album_type' => 'album'
      'available_markets' => ARRAY(0x7fea61268880)
        0  'AR'
        1  'AU'
        2  'AT'
        3  'BE'
        4  'BO'
        5  'BR'
        6  'BG'
        7  'CA'
        8  'CL'
        9  'CO'
        10  'CR'
...
        58  'ID'
      'external_urls' => HASH(0x7fea614a94c8)
        'spotify' => 'https://open.spotify.com/album
              /3nVaq2gmNw8Z7k7rgVB961'
      'href' => 'https://api.spotify.com/v1/albums/3nVaq
              2gmNw8Z7k7rgVB961'
      'id' => '3nVaq2gmNw8Z7k7rgVB961'
      'images' => ARRAY(0x7fea61bd6768)
        0  HASH(0x7fea614a9438)
          'height' => 640
          'url' => 'https://i.scdn.co/image/d64f75bc4f
              b474478b904b7e4058bf369d2373e1'
          'width' => 640
        1  HASH(0x7fea61bd68a0)
          'height' => 300
          'url' => 'https://i.scdn.co/image/d2e5b264f4
              87ac3bf1e3c32d48b1296247e99455'
          'width' => 300
        2  HASH(0x7fea61bd6f00)
          'height' => 64
```

```
            'url' => 'https://i.scdn.co/image/c20d9975b9
                 253255d879c9a10d8f3a8deab077e5'
            'width' => 64
          'name' => 'Only Everyone'
          'type' => 'album'
          'uri' => 'spotify:album:3nVaq2gmNw8Z7k7rgVB961'
```

We've received info about an album called "Only Everyone." The response tells us it is available in 58 markets. Album covers are available at the specified URLs. Another interesting field is "album_type." If we had wanted to, we could have narrowed down the type of album we were seeking (for example, if we wanted to see all of the singles available from this artist). To do that, we'd add an extra parameter to the query, as in:

```
my $albums = $s->artist_albums( $artist->{id},
                'album_type' => 'single' );
```

The result of our artist_top_tracks() method call is equally fun. In our code, we've asked what the top tracks are for that artist in the US market (remember the list of markets in the previous output?). Here's an excerpt from what we get back:

```
    'tracks' => ARRAY(0x7f841614be30)
      0  HASH(0x7f841486e0c0)
        'album' => HASH(0x7f84131f7978)
          'album_type' => 'album'
          'available_markets' => ARRAY(0x7f841227ea98)
'AR'

...

            58  'ID'
          'id' => '5mwk4GspWSXQHIkZGGdnhm'
          'name' => 'Boys and Girls Soundtrack'
          'type' => 'album'
          'uri' => 'spotify:album:5mwk4GspWSXQHIkZGGdnhm'
        'artists' => ARRAY(0x7f841250d5a8)
          0  HASH(0x7f841347d2e0)
            'id' => '71POUphzXd95FKPipXjtEO'
            'name' => 'Chloe'
            'type' => 'artist'
            'uri' => 'spotify:artist:71POUphzXd95FKPipXjtEO'
        'available_markets' => ARRAY(0x7f841347d5b0)
          0  'AR'

...

          58  'ID'
        'disc_number' => 1
        'duration_ms' => 203800
        'explicit' => JSON::PP::Boolean=SCALAR(0x7f8412196d08)
          -> 0
        'external_ids' => HASH(0x7f841347c8a8)
          'isrc' => 'USAK10000397'
        'id' => '600tF3aqxRjwJ0tdjxEwzY'
        'name' => 'Get You Off my Mind'
        'popularity' => 15
        'preview_url' => 'https://p.scdn.co/mp3-preview/4d99d5
                6fad8d2b31eb9fb7fa5c9a603fa23adb16'
        'track_number' => 14
        'type' => 'track'
        'uri' => 'spotify:track:600tF3aqxRjwJ0tdjxEwzY'
```

I've chopped a bunch of the fields of the data structure just to save space but left enough so that you can see that for each track, you get its name, the album it was on (and full info on that album), markets available, the artist info for that track, what disc it is (for multi-disc sets), what track it is on that disk, how long the actual track is, and even a URL to a 30-second MP3 preview of the track. Feel free to check out the preview, although I wish to insert a caveat that this example wasn't picked for its artistic merit.

There are a few more API calls we can make without authorization listed in the WebService::Spotify doc (for example, to return the tracks found on an album). I hope you have a sense of how you might build a more sophisticated script to do things like search for all of the albums and tracks by an artist.

What We Do in Private

There's a lot of fun that can be had using non-authorized API calls. I could probably end the column right here and you would have enough to play with for a long time. But one important part of these streaming music services is the ability to manipulate the service's music in various ways. Spotify is all about the playlist, so as the last item in this column, we're going to take a brief look at how to work with them from the API. In particular, we are going to look at how we retrieve our users' private playlists.

The very mention of users and ownership should perk up your ears because it means we get to get back into the authentication and authorization business. Spotify uses a method we discussed in detail a few columns back, namely OAuth2, to allow a user to delegate the privileges to a script/app to perform operations on your behalf. And this brings us back to the reason why using WebService::Spotify had such appeal. Yes, we certainly could have used LWP::Authen::OAuth2 as we did in that previous column, and I suspect it would work, but I was also perfectly pleased to use the OAuth2 support built into WebService::Spotify instead. There are good arguments for going either route, so I would say you should follow your own best judgment on this call.

Let's take a look at some code that uses OAuth2 behind the scenes to extract the contents of one of my private playlists. In order to use this code, I first had to register for an application at https://developer.spotify.com/my-applications/#!/applications. I gave the application a name, a description, and a redirect URI (I used "http://localhost:8888/callback"; more on that in a moment). Upon creation, the application was assigned a client ID and a client secret. Let's see the code and then we'll take it apart:

```
use WebService::Spotify;
use WebService::Spotify::Util;
```

```
my $username = "yourusername";

$ENV{SPOTIFY_CLIENT_ID}     = 'YOUR_CLIENT_ID_HERE';
$ENV{SPOTIFY_CLIENT_SECRET} = 'YOUR_CLIENT_SECRET_HERE';
$ENV{SPOTIFY_REDIRECT_URI}  = 'http://localhost:8888/
callback';

my $token = WebService::Spotify::Util::prompt_for_user_
     token($username);

my $s = WebService::Spotify->new( auth => $token );

my $playlist = $s->user_playlists($username)->{items}->[2];

# note, as of this writing, the module had a bug that
# causes the use of 'fields' to fail. It may be fixed
# by the time you read this.
#
# If not, line 133 of lib/WebService/Spotify.pm should read:
# return $self->get("users/$user_id/$method", fields =>
$fields);
my $tracks = $s->user_playlist(
   $username,
   'playlist_id' => $playlist->{id},
   'fields'      => 'tracks.items(track(name,album(name),
                 artists(name)))',
);

print "Playlist: $playlist->{name}\n";

foreach my $t ( @{ $tracks->{tracks}->{items} } ) {
   print "Track: $t->{track}->{name}\n";
   print "Album: $t->{track}->{album}->{name}\n";
   print "Artists: $t->{track}->{artists}->[0]->{name}\n";
   print "\n";
}
```

The first interesting part of the code is the "prompt_for_user_token" call. Here we are asking the module to do the necessary OAuth2 dance to get us a token that can be presented to the service by WebService::Spotify to show we have authorization.

When this line of the code runs, it spits out a URL that you need to paste into a browser and then prompts for the URL that will be returned to us. Spotify shows you a standard OAuth2 authorization request screen. If you authorize the request, the service attempts to redirect to the redirect URI we supplied during the application creation stage but adds on a few parameters. These parameters include a representation of a token we will need later.

Here's where I have taken a less than elegant shortcut. Unless you are doing something interesting on the machine running the browser (localhost), chances are you don't have a server running on port 8888 to handle this redirect. As a result, the browser displays a "Sorry, I can't go to that URL" error page. That's just fine, we don't actually need to complete the redirect, we just need the URL being used for that redirect. We can just copy the URL it attempted from the browser's URL field and paste it to the prompt from our running script so it can continue. I suppose if we wanted to be cooler we could indeed spin up a tiny server (e.g., using something like Mojolicious) to catch the redirect and print out the URL, but that level of coolness is not required for this particular application.

Now that we've done the OAuth2 auth dance, the code in this example uses the appropriate token when creating the WebService::Spotify object. WebService::Spotify will make sure to handle getting the right token to Spotify during the rest of the transactions with the service. It is also smart enough to cache the token (and ask for renewals if necessary), so we won't have to go through that initial authorization step again.

The first thing we do with our newfound authorization is request one of the playlist records from the list of playlists our user owns. I'm picking the third playlist in my account just because I know it is short and hence useful for this example. You could easily write code that iterates through all of them.

Now to get the tracks for this playlist—we ask for the contents of my user's playlist with the ID retrieved from the playlist record. To be fancy, we're adding an additional parameter to our request that limits the fields returned from our query. That field says "tracks.items(track(name,album(name),artists(name)))", which means "given all of the items in a playlist record, pull out the tracks, and from within the tracks pull out the track name, the name of the album that track comes from, and the artists' name." We could have left this parameter off and just grabbed the fields from the large response we get back, but this is a bit more efficient, and the returned object is easier to look at in a debugger. We iterate through the tracks in the playlist, printing out these items. Here's what the output looks like for my shortest playlist:

```
Playlist: Pre-class
Track: I Zimbra - 2005 Remastered Version
Album: Fear Of Music (Deluxe Version)
Artists: Talking Heads

Track: Default
Album: Default
Artists: Django Django
```

Working with all of this data is done just the way we saw in the previous sections. Creating (user_playlist_create) and manipulating playlists (user_playlist_add_tracks) is supported by WebService::Spotify as well. Both are straightforward given what we already know about playlist IDs and track URIs.

I hope you have fun experimenting with this API. Take care, and I'll see you next time.