Practical Perl Tools
OAuth2 in Situ

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

In the last column I threw a little hissy fit around the authentication options for working with the WordPress REST API. In it, I made noises about the adoption of OAuth 2.0 over version 1.0a and further grumbled about the backwards incompatibility (not to mention some of the politics around the changes) between the two versions. All of this piqued the interest of my editor who asked me to write some more on the topic. I'm still not thrilled about the OAuth situation, but I thought I would try to redeem myself by providing a column around the subject based on an actual piece of code that had to authenticate using OAuth2. This still won't address the OAuth 1.0a questions, but perhaps future columns will drag me kicking and screaming in that direction as well. One brief aside about version 1.0a because I need to make a slight correction: in the previous column I had suggested that 2.0 had all but supplanted 1.0a in the world. I've recently been discovering a few pockets of 1.0a (for example, Twitter's API, probably for historical reasons, seems to consist of this strange mishmash of the two), so I don't think 1.0a can be considered dead quite yet. Maybe we'll make with the Twitter in a future column.

Background

For those of you who haven't had the pleasure of diving into either of the OAuth versions, let's take a quick moment to describe the beast and what purpose it serves. In many of the cases where you will initially encounter it, it will be used as an authentication protocol (albeit one that appears to be more complex than it needs to be). But OAuth was designed to be much more than that. Here's the very best description [1] of the intent I have ever seen, from a blog post by Eran Hammer, one of the former lead authors of the spec, which is also quoted on oauth.net, the canonical home for OAuth material:

> Many luxury cars today come with a valet key. It is a special key you give the parking attendant and unlike your regular key, will not allow the car to drive more than a mile or two. Some valet keys will not open the trunk, while others will block access to your onboard cell phone address book. Regardless of what restrictions the valet key imposes, the idea is very clever. You give someone limited access to your car with a special key, while using your regular key to unlock everything.
>
> Every day new websites launch offering services which tie together functionality from other sites. A photo lab printing your online photos, a social network using your address book to look for friends, and APIs to build your own desktop application version of a popular site. These are all great services—what is not so great about some of the implementations is their request for your username and password to the other site. When you agree to share your secret credentials, not only do you expose your password to someone else (yes, that same password you

also use for online banking), you also give them full access to do as they wish. They can do anything they wanted—even change your password and lock you out.

This is the problem OAuth solves. It allows you, the User, to grant access to your private resources on one site (which is called the Service Provider), to another site (called Consumer, not to be confused with you, the User). While OpenID is all about using a single identity to sign into many sites, OAuth is about giving access to your stuff without sharing your identity at all (or its secret parts).

As an aside, in the process that led to 2.0, Hammer subsequently left the OAuth working group and withdrew his name from the specification because he thought 2.0 was deeply flawed. See his subsequent post, "OAuth 2.0 and the Road to Hell," [2] for more details. He's also got an entertaining, albeit NSFW, talk on the flaws of OAuth and the OAuth spec process [3]. I told you that OAuth was politically messy...

So a key thing to note about the OAuth intention statement above (that is brought out nicely in the first chapter of the upcoming book *OAuth2 in Action* by Justin Richer and Antonio Sanso) is that OAuth is less of an authentication protocol and more of a delegation protocol. The idea is that it can provide you a way to say, "Let this program/service/etc. have the access to do the following things on my behalf." If you've ever signed into a new mail or Twitter client and found yourself logging first into Gmail or Twitter to be faced with a screen of permissions to grant, you've likely been part of an OAuth transaction (of some version or other).

The Goal

Now that you know what sort of protocol we are dealing with, let's look at the actual goal of the script we're going to write together. Let's just say, hypothetically, that you work with an organization that uses Google Calendar to maintain its "in-out" listing (i.e., who is going to be out of the office, who is working from home, who is on vacation, etc.). Each person who is not going to be in the office marks this by making an event in this shared calendar. It can be a bit cluttered, but on the whole it works fine. There's just one niggling problem: sometimes people add calendar entries for their absences but forget to turn off notifications for those entries. This means that every day, everyone in the organization who subscribes to this calendar receives notifications (whether they like it or not) for those entries. We're going to write some code that connects to the Google calendar service, reads that calendar, and displays the events which have notifications still set. Then presumably we can go visit either the entries or the individuals in question and take corrective action.

Where does OAuth2 come in? Google forces you to use/interact with their (respected, even by Hammer) OAuth2.0 implementation to gain access to private calendars. Let's dive in.

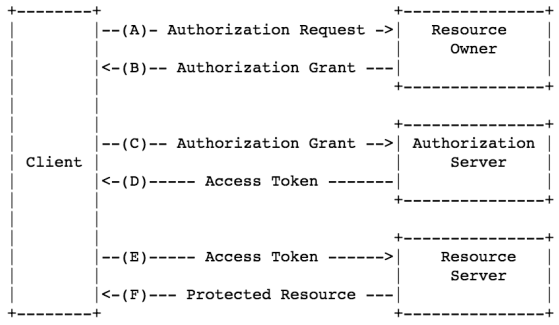

Figure 1: OAuth2 protocol flow, from https://tools.ietf.org/html/rfc6749

How Does It Work?

Before we get into looking at actual code, we probably should take at least a few steps up the OAuth2 learning curve (a fairly steep slope, I might add). Even if some magic Perl module will take care of all of the details behind the scenes, it is really important to get at least a general sense of what is going on. To do that, I'm going to elide a whole bunch o' details because OAuth2 has a number of different "flows" whereby different steps get taken depending on just what context it is being used in (e.g., a Web application, some application running on your computer, an application running in the browser, etc.) and just what kind of access the person interacting with the system has to a real Web browser for one of the steps (e.g., is it being used from a machine with a browser? in some device with no keyboard? etc.). In this column, we're only going to show one flow/scenario that works for the task at hand. If you plan to get deeper into this stuff, I highly recommend you read and reread and reread the documentation of the vendor that you will be talking OAuth2 to/with. Some (e.g., Google [4]) have quite decent documentation, so perhaps you will get lucky.

Okay, let's toss a stone and skim the surface. The basic OAuth2 protocol flow looks like the diagram from RFC6749 in Figure 1.

The dance goes roughly like this: you (the client) send an "I'd like my app to have the following access for this service" request to the resource holder of your choice (Google, in our case). That resource holder does something to verify you and to obtain your consent to delegate this access to your app. It hands back an authorization token that says, "Yes, I've confirmed that this person is fine and allowed to access X." Your app can then send this token to an authorization server, which hands you back another token (an access token) that will act as a key to unlock access to the protected resource in question when presented at the same time as a request for that resource.

In the case of the code we are going to write, the script will send a request to Google's auth service requesting read-only access to an account's private calendars. Google will reply with a URL,

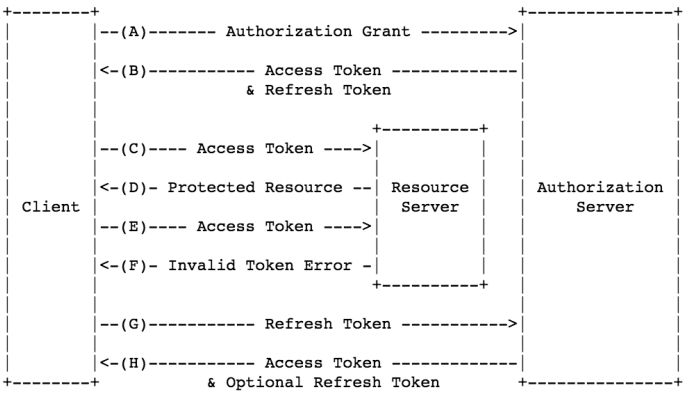

Figure 2: Refreshing an expired access token, from https://tools.ietf.org/html/rfc6749

which we'll go to "by hand." That URL will take us through the standard Google login procedure (if not logged in) and then pop up a screen that will say, "{App Name} would like to view your calendars [Deny] [Allow]." If you click the Allow button, a subsequent screen will be presented that says, "Please copy this code, switch to your application and paste it there: {longish_string_of_random_characters}." That string is the authorization token. We'll type it into a prompt presented by the script, the script will exchange it for an access token (which it will hold on to for us), and finally it will begin to send requests to the Google Calendar API with that token added to the headers of all of the requests. We'll then parse the returned data looking for specific calendar entries.

That's how we're going to dance. But I would be a little remiss if I didn't mention one complication that we won't touch out of either brevity or laziness (you decide). To improve the security of the situation, access tokens all have relatively short expiration times after which they can no longer be used. The authorization token we originally received is only good for one access token exchange, so what do we do after the access token expires? OAuth2 brings yet another token into the picture (yes, that's three so far, but who's counting?) called a refresh token. Figure 2 depicts the dance taken to refresh a token.

This crazy little ASCII diagram shows that we can also request a refresh token as part of the dance. The refresh token is supposed to be squirreled away someplace safe, only be brought out and sent over the wire when it is time to get a new access token issued as expiration approaches/arrives. Our script isn't going to run long enough to hit expiration timeouts, so we're not going to bother with this token, nor are we going to pay attention to storing tokens, something you would do for more complex/longer-lived applications.

Code Time

With that background in mind, let's look at two code samples. To get warmed up, let's look at how you would get a list of the calendars I have configured in Google calendar (the list of calendars I own and have subscribed to on the left side of the Google Calendar Web app).

Now, it would be entirely possible to write all of the OAuth2 code using bare-bones Perl modules used for making HTTP/S requests like LWP::UserAgent or HTTP::Tiny. We've seen a number of these in the past. But in this case, I'm perfectly happy to let someone else work out some of the fiddly bits because we are bound to hit plenty of fiddly when we actually get to use the Calendar API.

To make my life a bit easier, I am using LWP::Authen::OAuth2. Think of it as LWP::UserAgent with some magic OAuth2 pixie dust mixed in (plus it has some Google-centric code baked in). At the very least, I'd encourage you to look at its Overview doc, which does a good job of writing up some of the background/issues you are sure to want to know about before diving into this stuff.

So we start like this:

```
use LWP::Authen::OAuth2;
use Browser::Open qw( open_browser );
use IO::Prompt::Tiny qw/prompt/;
use JSON;

my $cal_uri = 'https://www.googleapis.com/calendar/v3';

my $oauth2 = LWP::Authen::OAuth2->new(
   client_id =>
'getyourowncode.apps.googleusercontent.com',
   client_secret     => 'need your own secret',
   service_provider  => "Google",
   scope             => 'https://www.googleapis.com/auth/
                          calendar.readonly',

   redirect_uri => "urn:ietf:wg:oauth:2.0:oob",

   # Optional hook, but recommended.
   #save_tokens => \&save_tokens,
   #save_tokens_args => [ $dbh ],

   # This is for when you have tokens from last time.
   #token_string => $token_string.
);
```

After setting a variable for later use that shows the base URL for the Google Calendar API, we create a new oauth2 object. It requires a client_id and client_secret from Google. To get one of these, you will want to go to https://console.developers.google.com, create a new project, enable just the Google Calendar API for it, then under "Credentials" request a new OAuth2 client_id. When

it asked for an application type, I selected "Other," and that's worked swimmingly for me for this script.

There are two other key parameters here. The first is "scope," which is the level of access we are requesting to the resource. With Google Calendar, the choices are few (read-only or read-write access). The other parameter is the redirect_uri. This is the URI that will be handling the authorization token once it is issued. That little weird-looking line just says, "Display the token in the browser and ask the user to do something with it" (vs. going to a real URI). I left the token-related parameters from the documentation commented out just as a reminder that better token handling would normally be inserted at this point.

oauth2 object in hand, we then do this:

```
my $url = $oauth2->authorization_url();

open_browser($url);

my $auth_code = prompt("Please enter auth code
                provided by Google:");

$oauth2->request_tokens( code => $auth_code );
```

The module figures out the right magic URL necessary for giving consent and obtaining the authorization token for us. Because I'm lazy, I pulled in a module that attempts to open a browser for you (works great on OS X, but your mileage may vary) with that URL. As I mentioned before, the end result is we are sitting at a Web page that says, "Please copy this code, switch to your application and paste it there: ...". The next lines have our script prompting for that code and exchanging it for an access token.

Now let's do the actual work with the Google Calendar API. The documentation at [5] is decent, but Google provides something even better, an awesome Web-based API explorer that shows all of the possible required and optional parameters and helps you try out various combinations before you actually code. This API explorer is linked off their Get Started page [6].

To actually get a list of calendars, we would do something like this:

```
my $response =
$oauth2->get("$cal_uri/users/me/calendarList");
die 'Could not retrieve cal list' unless
                  $response->is_success;

my $callist = decode_json $response->decoded_content;

foreach $cal ( @{ $callist->{items} } ) {
   print $cal->{summary} . ":\n" . $cal->{id} . "\n";
}
```

We make the proper GET request, and if it succeeds, we are handed back some JSON. We parse that JSON into Perl data structures and print a selected set of fields back from that info.

If that all makes sense, let's take a look at the code for performing our real task. Our goal is to find all of the events that have some sort of notification set on them. Google's calendar lets you set a default for the entire calendar for notifications; let's make the assumption that the default for the calendar is sane (otherwise, we'd get alerted for a very large quantity of events because people don't often change the default when they create an event). So now we have to locate the individual events that have a non-default notification set for them.

First, we'll need to make sure we are looking at the right calendar. To read the events from a calendar, you have to request them from the calendar by referencing that calendar's unique ID. That's the reason why the previous sample prints out both a calendar's summary (i.e., name) and its ID. The ID gets passed along in the request URL (along with some other parameters, more on that in a moment), so we'll need to make sure it is URL-safe.

Here's what it looks like to set an ID and make the original request for data (note, all of the OAuth2-related code is exactly the same as in the previous sample, so I'm only going to show the Calendar API-related code here):

```
my $calid = uri_escape 'somecalendarid@gmail.com';

$response =
$oauth2->get("$cal_uri/calendars/$calid/events?
          maxResults=100");

die 'Could not retrieve list of entries'
  unless $response->is_success;
```

You can see that the URL has changed and that we've added on a parameter to the URL itself. If you find you are using a number of parameters in the query, I recommend constructing the URL using the query_form() function from the URI module instead of doing it by hand as above.

So now we can print the results. Here we look for a special field in an event entry that indicates it is using a custom notification. If that exists, we print out the name of the event plus either the exact day and time it starts or just the day (if it is an all-day event).

```
foreach my $entry ( @{ $entries->{items} } ) {
     print "$entry->{summary} "
      . ( $entry->{start}->{dateTime} ||
          $entry->{start}->{date} ) . "\n"
     if exists $entry->{reminders}->{overrides};
}
```

So, we're done, right? Sorry, not so fast. If you have lots of entries in your calendar, you will have to deal with pagination. As we saw in the last column, the results come back *N* results at a time. By default, that number is 250, although you can raise it to 2500 according to the doc. I prefer to walk through the data in reasonable-sized pieces (100 at a time, that's what the maxResults parameter is doing there). With each result set (except for the last), Google hands back a nextPageToken that you can send in a subsequent request's pageToken parameter (note the different name!). Here's code that repeats the previous step for every subsequent page of data if there is any:

```
while ( $entries->{nextPageToken} ) {
   $response =
     $oauth2->get( "$cal_uri/calendars/$calid/events?"
        . "maxResults=100&"
        . "pageToken="
        . $entries->{nextPageToken} );
   die 'Could not retrieve addtl list of entries'
     unless $response->is_success;

   $entries = decode_json $response->decoded_content;

   foreach my $entry ( @{ $entries->{items} } ) {
      print "$entry->{summary} "
        . ( $entry->{start}->{dateTime} ||
             $entry->{start}->{date} ) . "\n"
        if exists $entry->{reminders}->{overrides};
   }
}
```

The key thing is that you send the exact same query again that yielded the paged result, the only difference being the addition of the pageToken parameter.

Now we are done. I hope this has given you a brief peek into how OAuth2 can be used to gain access to a real-live service. Take care, and I'll see you next time.

Resources

[1] Eran Hammer explains OAuth: http://hueniverse.com/2007/09/05/explaining-oauth/.

[2] Eran Hammer disclaiming OAuth2: http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/.

[3] Eran Hammer, NSFW, politics of OAuth2 process: https://vimeo.com/52882780.

[4] Google docs on using OAuth2: https://developers.google.com/identity/protocols/OAuth2.

[5] Google docs for using the Calendar API: https://developers.google.com/google-apps/calendar/.

[6] Calendar API explorer: https://developers.google.com/google-apps/calendar/get_started.