Practical Perl Tools

Rainy Days and Undocumented APIs Always Get Me Down

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 24+ 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

It seems like everyone talks about the weather, but few people code about it. It's not that I'm particularly enamored by weather (I go along with the Phantom Tollbooth quote, "I'm the Whether Man, not the Weather Man, for after all it's more important to know whether there will be weather than what the weather will be."), but I think it makes a lovely trampoline from which to explore a few of the more prevalent kinds of Web services/APIs you may encounter. For these demonstrations I'm going to stick to free or close to free services (at least for a personal level of queries). There are some commercial weather data providers who are exceptionally miserly with their data or force you to sign a EULA the size of my arm; they will be conspicuously absent from this column.

One quick note for my international readers: The goal of this column is to demonstrate how to bring Perl to bear to work with these kinds of APIs, not the specific services or APIs themselves. If any of these services fail to cover your particular geographical area, it is possible you can find one that does and use the same techniques to query it. It's not that I don't care deeply and passionately about the weather where you live; it is just that it is easier for me to show code that I can validate by looking out my window.

Weather Provided as XML

One thing that most of these services have in common is that they like to return data in a structured XML format. I thought I would start our exploration by looking at a service that returns a really simple XML document. Before I show this example to you, I have to admit we're going to be a bit naughty. The following example will query a service that doesn't really have a documented API for this purpose and is almost certainly not supported in this context. And even though there is a Perl module available to use this service (though we're going to do it by hand in this column), I can't recommend you use it for anything besides educational/demonstration purposes.

So who provides this API-less service that we're going to use in such a transgressive manner? Google. I realize this is a bit of a surprise given how important APIs are to them, but this is not a separate official service to them. Google provides weather data as part of their ability to customize your Google home page (iGoogle, sigh) with a weather gadget. There is also a small amount of information on how their weather data is represented in a document about customizing their toolbar. Hopefully, this information gives you a sense of just how much in the wilderness we'll be when we attempt to use this service.

That being said, the actual work is really easy. If you make an HTTP GET request to a URL of this form:

```
http://www.google.com/ig/api?weather={some place}
```

it will return an XML document like this (I used Boston, MA, as the place and reformatted the reply for easier reading):

```
<?xml version="1.0"?>
<xml_api_reply version="1">
   <weather module_id="0" tab_id="0" mobile_row="0" mobile_zipped="1"
   row="0" section="0">
      <forecast_information>
         <city data="Boston, MA"/>
         <postal_code data="Boston MA"/>
         <latitude_e6 data=""/>
         <longitude_e6 data=""/>
         <forecast_date data="2012-03-29"/>
         <current_date_time data="2012-03-29 16:54:00 +0000"/>
         <unit_system data="US"/>
      </forecast_information>
      <current_conditions>
         <condition data="Overcast"/>
         <temp_f data="42"/>
         <temp_c data="6"/>
         <humidity data="Humidity: 76%"/>
         <icon data="/ig/images/weather/cloudy.gif"/>
         <wind_condition data="Wind: N at 9 mph"/>
      </current_conditions>
      <forecast_conditions>
         <day_of_week data="Thu"/>
         <low data="34"/>
         <high data="48"/>
         <icon data="/ig/images/weather/rain.gif"/>
         <condition data="Showers"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Fri"/>
         <low data="37"/>
         <high data="50"/>
         <icon data="/ig/images/weather/sunny.gif"/>
         <condition data="Clear"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Sat"/>
         <low data="30"/>
         <high data="45"/>
         <icon data="/ig/images/weather/chance_of_rain.gif"/>
         <condition data="Chance of Rain"/>
      </forecast_conditions>
      <forecast_conditions>
         <day_of_week data="Sun"/>
         <low data="34"/>
```

```
        <high data="55"/>
        <icon data="/ig/images/weather/chance_of_rain.gif"/>
        <condition data="Chance of Rain"/>
      </forecast_conditions>
   </weather>
</xml_api_reply>
```

Now, let's grab the data using Perl and parse it. Since this is really simple XML and our use of this data is straightforward, we can turn to the tremendously helpful XML::Simple module to parse the data. Although XML::Simple can parse the data, it can't fetch it from Google. For that we'll use LWP::Simple, which provides a get() function that will retrieve data for a given URL. Here's the code:

```
use strict;
use LWP::Simple;
use XML::Simple;

my $xml = XMLin( get('http://www.google.com/ig/api?weather=Boston+MA'),
   ValueAttr => ['data'] );

print "Current conditions: "
   . $xml->{weather}->{current_conditions}->{condition} . " "
   . $xml->{weather}->{current_conditions}->{temp_f} . " F\n";

foreach my $day ( @{ $xml->{weather}->{forecast_conditions} } ) {
   print $day->{day_of_week} . ': '
      . $day->{condition} . ' '
      . $day->{high} . '/'
      . $day->{low} . "\n";
}
```

XML::Simple's XMLin function gets called to parse the data retrieved by LWP::Simple. We use the defaults for it with one exception to make our life easier. If you take a look at the sample XML document above, you'll see lines such as:

```
<condition data="Overcast"/>
<temp_f data="42"/>
<temp_c data="6"/>
<humidity data="Humidity: 76%"/>
```

where the elements don't actually hold the data; the attributes of those elements do. By default, XML::Simple will place those attributes into their own separate hashes with the name of the attribute as the key. This means we would ordinarily get a data structure that looks like this excerpt:

```
'current_conditions' => HASH(0x7f8032f32b80)
  'condition' => HASH(0x7f8032f332b8)
    'data' => 'Overcast'
  'humidity' => HASH(0x7f8032f2da90)
    'data' => 'Humidity: 73%'
  'icon' => HASH(0x7f8032f2db20)
    'data' => '/ig/images/weather/cloudy.gif'
  'temp_c' => HASH(0x7f8032f333d8)
    'data' => 6
  'temp_f' => HASH(0x7f8032f33348)
    'data' => 42
```

```
'wind_condition' => HASH(0x7f8032f2dbb0)
  'data' => 'Wind: N at 9 mph'
```

It would be much more pleasant if we could just eliminate the need for a separate sub-hash to hold the values, and instead get something like:

```
'current_conditions' => HASH(0x7fc001733058)
   'condition' => 'Overcast'
   'humidity' => 'Humidity: 73%'
   'icon' => '/ig/images/weather/cloudy.gif'
   'temp_c' => 6
   'temp_f' => 42
   'wind_condition' => 'Wind: N at 9 mph'
```

and indeed, that's what the ValueAttr option to XMLin() does for us in one swell foop. Now you get some sense of why I tend to be pretty effusive in my praise of XML::Simple.

Weather Provided as an RSS Feed

The second kind of weather service I'd like to explore with you is one I introduced in a column back in 2006. There are services that provide weather data to you as RSS feeds. RSS is better known as a blog-related standard, so you may not have encountered it much except as internal plumbing found largely behind the scenes. Wikipedia's got the following lovely description:

> RSS (originally RDF Site Summary, often dubbed Really Simple Syndication) is a family of Web feed formats used to publish frequently updated works—such as blog entries, news headlines, audio, and video—in a standardized format. An RSS document (which is called a "feed," "Web feed," or "channel") includes full or summarized text, plus metadata such as publishing dates and authorship.
>
> RSS feeds benefit publishers by letting them syndicate content automatically. A standardized XML file format allows the information to be published once and viewed by many different programs. They benefit readers who want to subscribe to timely updates from favorite websites or to aggregate feeds from many sites into one place.

The RSS format spec has gone through a number of revisions, but all of them are represented using XML. We could use a generic XML parser to deal with it (as you'll see in the next section in this column), but in this case it is a little easier to use a dedicated RSS module called XML:RSS::Parser to parse the data. There is also an XML::RSS module available that I would normally use, but it doesn't seem to (at least in my experience) play nicely with the slightly customized RSS feed we're going to consume in this section.

Yahoo! is probably the most popular provider that makes RSS feeds for weather available, so we'll use them for the code example for this section. Querying Yahoo! for weather for a US location is as easy as requesting the RSS feed for that location's zip code using a URL:

```
http://xml.weather.yahoo.com/forecastrss?p={zipcode here}
```

Even though that query format works, it is deprecated; instead, Yahoo! now wants you to do this instead:

```
http://weather.yahooapis.com/forecastrss?w={WOEID}
```

In the second format above, you provide a WOEID in the URL using the w parameter. Yahoo! created the WOEID, or "Where On Earth ID," to be a unique identifier for any place on the planet. It's a cooler system than you might expect (so cool Twitter decided to adopt it). More details on it can be found at this Yahoo! URL: http://developer.yahoo.com/geo/geoplanet/guide/concepts.html.

So where do you get the WOEID for a particular place? Yahoo! suggests the easiest way to do so is to search for that place at http://weather.yahoo.com. The resulting URL for that place's weather page will end in the WOEID for that place. For example, if I search for Boston, MA, the URL for the page that is returned has this URL:

```
http://weather.yahoo.com/united-states/massachusetts/boston-2367105/
```

If the idea that you have to type in each place you would want to query into a search box in a browser seems a bit, ehemm, manual to you (and I certainly hope it does to regular readers of this column), Yahoo! provides a place-to-WOEID query service available. It's simple to use, but I think it is out of the scope of this column. For more details, please see http://developer.yahoo.com/geo/geoplanet/.

So let's get back to the task at hand and see how to use XML::RSS::Parser to deal with data from Yahoo!'s RSS-based weather service. XML::RSS::Parser doesn't actually fetch the data from either of the two Yahoo! RSS feed URLs above, so we'll again use LWP::Simple's get() function. Putting all of these pieces together, we get sample code that looks like this:

```
use strict;
use LWP::Simple;
use XML::RSS::Parser;

my $parser = XML::RSS::Parser->new;

# Yahoo! uses a custom namespace for their data
$parser->register_ns_prefix( 'yweather',
   'http://xml.weather.yahoo.com/ns/rss/1.0' );

# 2367105 is the WOEID for Boston, MA
my $feed = $parser->parse_string(
   get('http://weather.yahooapis.com/forecastrss?w=2367105') );

print "Current Conditions: "
   . $feed->query('//yweather:condition/@yweather:text') . " "
   . $feed->query('//yweather:condition/@yweather:temp') . " F\n";

my (@forecasts) = $feed->query('//yweather:forecast');
foreach my $day (@forecasts) {
   print $day->query('@yweather:day') . ': '
      . $day->query('@yweather:text') . ' '
      . $day->query('@yweather:high') . '/'
      . $day->query('@yweather:low') . "\n";
}
```

Most of the code above is pretty straightforward, with one exception. The lines that include code like this might be a bit curious:

```
$feed->query('//yweather:condition/@yweather:text')
```

One of XML:RSS::Parser's strengths (which it gets from the Class::XPath module it uses) is that it provides an XPath-like/lite query language. XPath provides a terse but elegant syntax for finding elements in an XML document. I like it a great deal, so much so that I suspect you can look for a future column on just XPath. In the meantime, let me explain that the code above returns the yweather:text attribute from an XML element with the name "yweather:condition" found anywhere in the document ('//' means anywhere starting from the root element).

The line later in our code that says:

```
my (@forecasts) = $feed->query('//yweather:forecast');
```

is performing a similar query, this time requesting all of the elements called yweather:forecast. We iterate over each of the elements returned by that query, and for each element we request the attributes we want to display:

```
foreach my $day (@forecasts) {
   print $day->query('@yweather:day') . ': '
      . $day->query('@yweather:text') . " "
      . $day->query('@yweather:high') . "/"
      . $day->query('@yweather:low') . "\n";
}
```

Weather Provided as JSON

For our final example, I though it would be good to change up the format we're processing even though XML is by far the most prevalent format being used to provide weather data. But XML itself is starting to get stiff competition from another data interchange format when it comes to Web services these days. The competitor is JSON, made popular because AJAXy things are tending to use it more and more. To understand a bit about JSON, I want to quote verbatim from the json.org Web site:

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition—December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
>
> JSON is built on two structures:
>
> - A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
> - An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.
>
> These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

If you've dealt at all with YAML, you will have very little problem coping with JSON (YAML proponents claim it is a superset of JSON). We explored working

with JSON back in the June 2008 *;login:* column, so you may want to take a quick look at that column if you'd like more information on how to work with it from Perl.

For this demonstration, we're going to use the Web service from Weather Underground (wunderground.com) that does charge money for their service if used over a certain amount. The free plan offers you 500 calls per day, 10 calls per minute—more than sufficient for the needs of this column. If you want to go higher, the next tier (5000 calls per day, 100 per minute) is only $20US per month, nothing outrageous.

One quick aside before we actually get into coding against their API: our previous data provider, Yahoo! actually has a "secret" (i.e., not directly documented in their Weather section, as far as I can tell) JSON API available. I think one undocumented API per column is more than enough so I'm going to just mention it exists (use "forecastjson" in the URL instead of "forecastrss") and move on.

To use the Weather Underground API, you need to sign up for an API key. With that key, you can construct your query URL. In the examples below, I've replaced my personal API key with YOURAPIKEY.

Unlike the previous services we've seen, Weather Underground lets you request different "features" from the service by adding keywords to the part of the URL you would normally associate with the path to the resource. For example, if I wanted to retrieve just the current conditions for a place, I would use a URL that began:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions ...
```

If I wanted to duplicate what we've received from the other services by requesting both the current conditions and the forecast, the URL would begin with:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast ...
```

After the features part of the URL, you provide the location and an indication of the format you'd like back. Here's the complete URL we'd use to get back the current conditions and forecast for Boston as a JSON document:

```
http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast/q/MA/Boston.json
```

Based on our previous examples, you can probably guess what our sample code will look like. The main difference is we'll be feeding the results of our get() to the JSON module's from_json function. This converts the JSON documented into a Perl data structure, along the lines of this (I've heavily excerpted below because the data you get back is pretty voluminous):

```
0  HASH(0x7fd2bad12020)
   'current_observation' => HASH(0x7fd2bad11f18)
      'relative_humidity' => '79%'
      'solarradiation' => 109
      'station_id' => 'KMAWINTH1'
      'temp_c' => 6.1
      'temp_f' => 42.9
      'temperature_string' => '42.9 F (6.1 C)'
      'visibility_km' => 16.1
      'visibility_mi' => 10.0
      'weather' => 'Overcast'
      'wind_degrees' => 47
      'wind_dir' => 'NE'
```

```
    'wind_gust_kph' => 16.1
    'wind_gust_mph' => 10.0
  'forecast' => HASH(0x7fd2baad6758)
    'simpleforecast' => HASH(0x7fd2bad00850)
      'forecastday' => ARRAY(0x7fd2badfcf08)
        0  HASH(0x7fd2badfcfc8)
          'avehumidity' => 67
          'avewind' => HASH(0x7fd2badfdd48)
            'degrees' => 34
            'dir' => 'NE'
            'kph' => 10
            'mph' => 6
          'conditions' => 'Rain Showers'
```

The trickiest part is simply finding the right parts of the data structure to display. Here's our last piece of sample code:

```
use strict;
use LWP::Simple;
use JSON;

my $weather = from_json(

   get('http://api.wunderground.com/api/YOURAPIKEY/conditions/forecast/q/
   MA/Boston.json'
   )
);

print 'Current conditions: '
   . $weather->{current_observation}->{weather} . " "
   . $weather->{current_observation}->{temp_f} . " F\n";

foreach my $day ( @{
     $weather->{forecast}->{simpleforecast}->{forecastday} } ) {
        print $day->{date}->{weekday_short} . ": "
            . $day->{conditions} . ' '
            . $day->{high}->{fahrenheit} . '/'
            . $day->{low}->{fahrenheit} . "\n";
}
```

To end this column, let me show you the current output of the previous code so you can feel a bit better about the weather near you:

```
Current conditions: Overcast 42.9 F
Thu: Rain Showers 46/36
Fri: Clear 48/36
Sat: Chance of Rain 43/30
Sun: Chance of Rain 52/32
```

Take care, and I'll see you next time.