DAVID BLANK-EDELMAN

practical Perl tools: Car 10.0.0.54, where are you?

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of *Perl for System Administration* (O'Reilly, 2000). He has spent the last 20 years as a system/network administrator in large multiplatform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the chair of the LISA '05 conference.

dnb@ccs.neu.edu

WHEN SOMEONE ASKS ME ABOUT Web services, I hear this loud buzzing sound, because those words are all the latest rage. Give it a year or two, and either the ballast will get changed so they don't buzz as much, or people will have moved on to something else. In the meantime, let's take a look at something in that ballpark.

If this were a more theory-oriented column we might talk about the fundamentals of Web services. We'd probably look at how Web services are sometimes just an extension of the standard client-server module in that they often entail one server consuming the output from another server as part of performing a task for a user. We'd note that XML is a key component of many Web services because it provides a lingua franca/Esperanto in which these server-to-server conversations can take place. Having a well-structured language for this purpose makes it much easier to write the software for either end of the transaction. Mention of XML would no doubt lead to talk of more complex protocols built on XML such as XML-RPC and SOAP. WSDL (the Web Services Description Language), a way of describing the possible conversations for a Web service, would also be a natural segue. For good measure, we might even get into REST (REpresentational State Transfer) as another way of thinking about Web services.

But that's all good material for a column with a different bent. This time we're going to focus on something a little more fun in the general vicinity of Web services. If you are interested in the fundamentals (and you probably should be), there's a decent *Programming Web Services with Perl* book by Randy J. Ray and Pavel Kulchenko. However, today's topic is one of my favorite Web service application realms: geocoding.

Geocoding from Postal Addresses

Let's start with one of the standard tasks: Given a postal address of some sort, is it possible to locate that address on the planet such that we could plot it on a map? This is one example of a process known as geocoding. Doing geocoding well (where well means "could use it for commercial applications") is actually a fairly hard problem for a number of reasons, including all the data being suspect. Postal addresses can be ambiguous, the geographical data are sometimes incomplete or incorrect, and both humans and nature are always changing the surface features of the planet. This is all said to help form a disclaimer that holds true for everything in this column. Try the examples here, but don't depend too heavily on them. If you need professional geocoding done, hire a professional.

> Disclaimer 1: I have no commercial or other relationship to the various Web service providers mentioned in this article beyond occasionally paying for the cheaper ones so that I can play with them.
>
> Disclaimer 2: Often when people in the United States talk about geocoding, they really mean "North America geocoding" and are much less concerned with finding points outside of the U.S. Setting aside the standard U.S. ethnocentrism, we see that this is also a function of the availability of data. The U.S. government makes a passable data set available for free; most other countries don't have an equivalent. If you are interested in non-U.S. geocoding, the people at www.nacgeo.com have a relatively inexpensive commercial offering that may suit your needs.

If we leave out the expensive for-pay geocoding services, there are still a few geocoding methods available to us. The first one Perl people tend to turn to is the geocoder.us Web site/service provider, because they provide not only a free set of Web services but also the Geo::Coder::US module on CPAN should you desire to set up your own server. geocoder.us offers several different flavors of Web service, including XML-RPC, SOAP, REST, and "plaintext" REST. We're going to pick XML-RPC to start with, because the code to use it is very simple:

```
use XMLRPC::Lite;
my $reply = XMLRPC::Lite
    -> proxy ( 'http://rpc.geocoder.us/service/xmlrpc' )
    -> geocode( '2560 9th Street, Berkeley, CA')

    -> result;

foreach my $answer (@{$reply}){
    print "lat: "   . $answer->{'lat'}
        . " long: " . $answer->{'long'} . "\n";
}
```

First we load the XMLRPC::Lite module, which is bundled in the SOAP::Lite distribution. The proxy() method (which, despite its name, doesn't have anything to do with a Web proxy or any other kind of proxy) is used to specify where the query will be directed. We make our remote call out to that server using the geocode() method and ask XMLRPC::Lite to return the result.

The code for printing the result may look a little more complex than necessary. geocode() returns a list of hashes, one hash per result of the query. Some queries can yield multiple answers (e.g., if we asked for "300 Park, New York, NY" there might be a 300 Park Street, a 300 Park Drive, and a 300 Park Lane). There's only one 9th Street in Berkeley, so it would have been easier (but less robust) to write the following:

```
print "lat: "  . $reply->[0]->{'lat'}  .
      "long: " . $reply->[0]->{'long'} . "\n";
```

If you decide for some reason that you don't like the results you receive from geocoder.us, there are a number of other cheap geocoding services available; these include ontok.com (but be warned that later versions of SOAP::Lite do not play nicely with its SOAP interface) and Yahoo!'s REST-based geocoding API (for fewer than 5000 queries a day). Let's look at the latter. To use this service, we need to apply for a free application ID at http://api.search.yahoo.com/webservices/register_application. With that ID we can then use the API described at http://developer.yahoo.com/maps/rest/V1/geocode.html. Here's some sample code to do that:

```
use LWP::Simple;
use URI::Escape;
use XML::Simple;

# usage: scriptname <location to geocode>
my $appid  = "{your appid here}";
my $requrl = "http://api.local.yahoo.com/MapsService/V1/geocode";

my $request = $requrl .
    "?appid=$appid&output=xml&location=" . uri_escape( $ARGV[0] );

my $response = XMLin( get($request), forcearray => ['Result'] );

foreach my $answer ( @{$response->{'Result'}} ){
    print "Lat: $answer->{Latitude} " .
          "Long: $answer->{Longitude} \n";
}
```

One of the pleasant properties of REST interfaces is that they are really easy to query. If you know how to retrieve a Web page in Perl using a GET or PUT, you can use a REST interface. In the preceding example, we construct the URL by taking the base Yahoo! REST request URL and adding a few parameters, that is, the required appID, our preferred output format, and a URL-encoded version of the location to query. This gets handed to LWP::Simple's get() routine, the output of which we immediately parse using XML::Simple.

XML::Simple would ordinarily hand us back a hash that contained a single hash if the geocode server returned a single response. If the server returned several answers—remember the ambiguous address case in our last example—it would provide a hash that contained a list of hashes, one for each answer. When it came time to display the results, we could have written code to distinguish between the single answer data structure and the multianswer data structure, using ref(), and act accordingly, but that's too much work. Instead, we take the easy way out and ask XML::Simple (via forcearray =>['Result']) to always hand us back a hash with a list of hashes. The code for results output then gets to do an easy foreach walk over that list.

Note that if this code seems a little too complex for you, there's even a simpler way to do it courtesy of the Geo::Coder::Yahoo module. This module has exactly two calls in it, one to create the search object and another to call the geocoding API. The latter call returns a list of hashes, with no XML parsing required. Use whichever one suits your fancy.

Now that we've seen a couple of ways to turn an address into its corresponding latitude and longitude, what can we do with that information? The obvious answer to this question is to plot the information on a map. There are a number of good Web services for doing this, including Google Maps (http://www.google.com/apis/maps/), Yahoo! Maps (http://developer.yahoo.com/maps/), and TerraServer (http://terraservice.net/webservices.aspx). For fun, you can generate KML or KMZ (compressed KML) files for Google Earth (http://earth.google.com/kml/) and fly between your data points.

The process of plotting geocode data into one of these maps usually involves fiddling with HTML and that icky Javascript stuff. In Perl we luck out for Google Map creation, because Nate Mueller has written an

HTML::GoogleMaps module that makes the process really easy. Here's a sample CGI script that displays a map with labeled marker pointing at the USENIX mothership:

```
use HTML::GoogleMaps;

my $coords = [-122.291713, 37.859524]; # 2560 9th Street, Berkeley, CA
my $map = HTML::GoogleMaps->new( key => '{your api key here}' );
$map->center( $coords );           # center it on the address
$map->zoom( 2 );                   # zoom it to street level

# add a marker at the address using the given html as a label
# (and don't change the size of that label)
$map->add_marker(
    point    => $coords,
    noformat => 1,
    html     => "<a href='http://www.usenix.org'>USENIX</a> office" );

# add some map controls (zoom, etc)
$map->controls( "large_map_control", "map_type_control" );

# create the parts of the map
my ($head, $body, $js) = $map->render;

# output the HTML (plus CGI-required Content-Type header) for that map
print "Content-Type: text/html\n\n";
print << "EOH";
<html>
<head>
<title>;login test</title>
$head
</head>
EOH

print "<body> $body $js</body> </html>\n";
```

Note: The above code uses HTML::GoogleMaps to generate output for the Google Maps v1 API. A few days after this article was submitted for publication, Google released version 2 of their Maps API. They are pushing developers to upgrade before v1 is decommissioned. Luckily, the author of HTML::GoogleMaps is hard at work and should have a v2-compliant update to his module by the time you read this.

There's much more that can be done with Google Maps and the other services. Be sure to check out the respective documentation for these services and products.

Geocoding from IP Addresses

Let's circle back to the original question that started this column, namely, "Given a postal address of some sort, is it possible to locate that address on the planet?" It seems eminently doable that one could take a postal address and look it up on some list to find its coordinates. That seems like something you can picture rows and rows of clerks in little green visors doing in a big, nondescript office somewhere in the Midwest.

It's a lot more magical if I tell you, "Give me the name of your computer on the Internet and I can make a guess as to where that computer is located." There's something about crossing over the virtual/physical divide that makes this task seem all the more impressive. There are a number of reasons (besides impressing people at parties) for wanting to geocode from an IP address, and we'll get to those in a minute as well.

The first step of the process is to turn the DNS fully qualified domain name into an IP address. That's straightforward with the Net::DNS module:

```
    use Net::DNS;
    my $resolv = Net::DNS::Resolver->new;

    my $query = $resolv->search( $ARGV[0] );

    die "No response for that query" if  !defined $query;

    # only print addresses found in A resource records
    foreach my $resrec ( $query->answer ){
        print $resrec->address . "\n" if ($resrec->type eq "A");
    }
```

Chances are you'll only be geocoding a name that has one IP address associated with it, but the code listed here tries to give you back all of the addresses returned in response to your query. Note that if you plan to do this sort of lookup many times (e.g., when parsing a log file), you'll want to maintain a cache of your results as you go along so you can avoid beating up the name servers needlessly. If you plan to process massive amounts of data, you'll probably want to look into some of the asynchronous DNS libraries such as adns (http://www.chiark.greenend.org.uk/~ian/adns/) to handle parallel queries well.

Now that we have an IP address in hand, it is time to bring Web services back into the picture. There are a few fairly cheap (for the amount of data I push through them) providers. The following examples use the service provided by maxmind.com, because that is the one I've played with the most. We're going to concentrate on Web services, but it should be noted that MaxMind and several other providers offer both a Web services interface to their data and a database subscription that allows you to download the data to your server for faster lookups.

For MaxMind's Web service, we just need to construct a simple HTTP GET (or PUT, if that is your fancy) similar to what we did for the Yahoo! API in a previous example. The main difference between that example and this one is the format returned. Here we get Comma/Character Separated Values (CSV) instead of something in XML format:

```
use LWP::Simple;
use Text::CSV_XS; # This is the faster version of Text::CSV

# usage: scriptname <IP address to geocode>

my $maxmkey  = "{maxmind key here}";

my $requrl = "http://maxmind.com:8010/f";
my $request = $requrl . "?l=$maxmkey&i=$ARGV[0]";

my $csvp = Text::CSV_XS-> new(); # (or Text::CSV->new())

    $csvp->parse(get($request));

my ($country, $region, $city, $postal, $lat, $lon,
    $metro_code, $area_code, $isp, $org, $err) = $csvp->fields();
```

You've already seen what we can do with the results of a latitude and longitude geocoding; let's briefly look at how the other fields could be pressed into service.

If we ran some code against our Web server log, we could use $country to create a nice Web page showing the flags from the countries that have visited the site that day. There are a number of places to get the flag data. For example, ip2location.com, one of MaxMind's competitors, offers a whole set of tiny flag gifs available for download for free from http://www.ip2location.com/products.asp. If you prefer larger flags from a more interesting source, the Geo::CountryFlags module will download them on the fly for you from the Central Intelligence Agency's World Factbook. That's a simple process:

```
use Geo::CountryFlags;
# returns the path to the flag file it downloaded or undef if not found

my $path = Geo::CountryFlags->new->get_flag('{country code}');
```

On a more techie note, we could use the information on what country the request comes from to direct someone to their closest mirror Web site, provide a reasonable default in a Web form asking for address information, or even provide the site in a different language. There are some helpful Perl modules that use a local database (from one of the subscription services mentioned earlier) to handle this process for you. For example, if you use Apache::Geo::Mirror, then the documentation points out that you can put this in your Apache configuration:

```
PerlModule Apache::Geo::Mirror
<Location /CPAN>
  PerlSetVar GeoIPDBFile "/usr/local/share/geoip/GeoIP.dat"
  PerlSetVar GeoIPFlag Standard
  PerlSetVar GeoIPMirror "/usr/local/share/data/mirror.txt"
  PerlSetVar GeoIPDefault us
  PerlHandler Apache::Geo::Mirror->auto_redirect
</Location>
```

and your Web server will automatically redirect a client to the right mirror site based on the country associated with its IP address.

To wind this column down with a last Web services flourish, let's end with one more fun use of this sort of data. If we geocode an IP address associated with a U.S. address and get back a zip code, it is easy to provide the weather forecast for that zip code. I know of at least four U.S. weather services that are free for noncommercial use:

- NOAA's National Weather Service has a SOAP-based service; details are at http://www.weather.gov/xml/.
- Weather.com provides an XML-based service; details are at http://www.weather.com/services/xmloap.html (though it comes with a whole boatload of requirements you have to satisfy if you want to use it on your Web site).
- Yahoo! provides weather information via RSS; see http://developer.yahoo.com/weather/. You'll need to parse the RSS format using something like XML::RSS (or even XML::Simple).
- http://www.rssweather.com also provides weather info via RSS.

To end this column, let's put several of these parts together. The following is a CGI script that attempts to determine your zip code from your IP address and then queries Yahoo! for your current weather conditions and forecast:

```
use LWP::Simple;
use Text::CSV_XS;
use XML::RSS;

my $maxmkey  = "{maxmind key here}";
my $requrl  = "http://maxmind.com:8010/f";
my $request = $requrl . "?l=$maxmkey&i=$ENV{'REMOTE_ADDR'}";
my $csvp = Text::CSV_XS->new();

$csvp->parse( get($request) );
my ($country, $region, $city, $postal, $lat, $lon,
    $metro_code, $area_code, $isp, $org, $err) = $csvp->fields();

print “Content-Type: text/html\n\n”;
print << “EOH”;
<html><head><title>;login test</title></head>
<body>
EOH
print “<p>Hi there “ . $ENV{‘REMOTE_ADDR’} . “!</p>\n”;

if ($postal) {
    my $rss = new XML::RSS;
    $rss->parse(
      get(“http://xml.weather.yahoo.com/forecastrss?p=$postal”) );
    print “<h1>” . $rss->{items}[0]->{‘title’} . “</h1>\n”;
    print $rss->{items}[0]->{‘description’}, “\n”;
}
print “</body></html>\n”;
```

Pretty cool, eh?

And with that, I’m afraid we have to bring this issue’s column to a close. Take care, and I’ll see you next time.

[Correction: In my last column I pointed people at the PUGS project (www.pugscode.org). At the time I wrote the column it hadn’t come to my attention that the developer who started the project had changed her name to Audrey. My apologies to Ms. Tang for the mistake.]