Practical Perl Tools

"Mala trom pee chock makacheesa."

DAVID 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

In this column, we hope to address the question "Is Perl the language of love?" And if that attempt falls short (as I suspect it will), is there a way to speak the language of love using Perl? Now, you and I may disagree which language is indeed the language of love. Is it Huttese, as in the title above (okay, really Quechua) or some other language? We're going to need some way to go back and forth between languages fluidly to answer this question.

One way to do this would be to use machine translation. Perhaps the most well known way to access this kind of translation is through a service Google provides called Google Translate. We're going to look at how to work with this service from Perl and also explore some of the little side lessons we can pick up along the way. There are three quick things I need to bring to your attention before we jump into how this all works:

1. Google Translate is not free to use; it used to be, back in the day, but now you have to pay a small amount to even play with it. Their pricing is listed on the Web site (https://developers.google.com/translate/v2/pricing). As of this writing, I'll be shelling out $20 US for the first one million characters of text being translated in this column. There is a separate charge of $20 US for using their language detection feature (again, per one million characters). To use the code found in this article or to write your own, you'll need to obtain your own Google Translate API key (https://code.google.com/apis/console/?api=translate) and also set it up so Google can bill you for the usage.
2. In addition to paying for Google Translate use, there are a whole bunch of other requirements you must adhere to in terms of how you need to identify its use in your application, branding, blah, blah, blah. Please read the "Attribution Requirements" and "HTML Markup Requirements" sections of their documentation (https://developers.google.com/translate/) carefully. You may wish to play the proper sections of John Williams' soundtrack in the background for proper effect.
3. There is a Perl module whose whole job is to hide the implementation details of using this service. I will indeed show you how to use it toward the end of this column, so if you are impatient, you can skip to the end. We're going to learn a bunch of cool things before we get there, but, hey, I'll understand if you are a busy bounty hunter and don't have the time to read all the way through.

Let's Take a REST

Many, many Web services these days provide some sort of REST API. REST stands for "Representational State Transfer." The Wikipedia article on REST at http://en.wikipedia.org/wiki/Representational_State_Transfer is decent (or was on the day I read it). Let me quote briefly from it:

> REST-style architectures consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource
>
> ....
>
> Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: presented with a network of Web pages (a virtual state-machine), the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for his use.

(That last sentence comes from Roy Fielding's dissertation, which actually defined the REST architecture and changed Web services forever as a result.)

Discussing the REST idea in detail would get us into a whole other kettle of fish that might leave both of us smelling bad. For example, I think some people might quibble with Google calling their Google Translate API a REST API. They acknowledge this in their doc when they say "the Google Translate API is somewhat different from traditional REST. Instead of providing access to resources, the API provides access to a service."

They essentially bend the REST idea to say that using this service consists of constructing URLs with the right service request details in them, fetching the URL, and getting data back with the results of that service request. To quote their doc directly:

> "[T]he API provides a single URI that acts as the service endpoint.
>
> You access the Google Translate API service endpoint using the GET REST HTTP verb, as described in API operations. You pass in the details of all service requests as query parameters."

In this column, we've done this sort of thing lots of times. Let's use similar code to take a baby step:

```
use LWP::Simple;

my $tkey = '{YOUR Google Translate API KEY HERE}';

my $results = get("https://www.googleapis.com/language/translate/v2/
languages?key=$tkey");

print "$results\n";
```

This prints out what looks like a large data structure along the lines of:

```
{
 "data": {
  "languages": [
   {
    "language": "af"
   },
   {
    "language": "ar"
   },
   ...
    {
    "language": "de"
   },
   {
    "language": "el"
   },
   {
    "language": "en"
   },
   ...
   {
    "language": "vi"
   },
   {
    "language": "yi"
   },
   {
    "language": "zh"
   },
   {
    "language": "zh-TW"
   }
   ]
  }
}
```

And with that, we've made our first Google Translate API call. In this case we've asked for the list of languages supported by the service.

When you look at the output, you may be thinking, "Hey, that output looks like a data structure but is pretty legible; what format is it in?" Glad you asked. The Google Translate API returns data in JSON (JavaScript Object Notation) format. JSON has become one of the lingua francas of Web services on the Net. We've seen JSON in this column before because it is a kissing cousin (where kissing cousin might be better described as "a subset") of the YAML data serialization format that shows up all around the Perl world. Given JSON's ubiquity, knowing how to work with it is a desirable skill that is easy to pick up.

As an example, we could modify our previous code to read like this:

```
use LWP::Simple;
use Data::Dumper;
use JSON;

my $tkey = '{YOUR Google Translate API KEY HERE}';

my $results = get(
   "https://www.googleapis.com/language/translate/v2/
languages?key=$tkey");

my $decoded = decode_json $results;

foreach my $language ( @{ $decoded->{data}->{languages} } ) {
   print values $language, "\n";
}
```

and it would print out the list of supported languages. Let's take apart the new parts of this code so it is clear. We've loaded the JSON module that turns JSON data into a Perl data structure. In this case, it has returned a hash reference to a data structure that looks like this (courtesy of the Perl debugger):

```
-> HASH(0x7f88f4499e48)
     'data' => HASH(0x7f88f44eef68)
       'languages' => ARRAY(0x7f88f4760ad8)
         0  HASH(0x7f88f462efb0)
           'language' => 'af'
         1  HASH(0x7f88f462ef98)
           'language' => 'ar'
  ...
```

There's an outer hash with the single key of `'data'` that contains a reference to an inner hash whose only key is `'languages'` and whose value is a reference to a Perl array. Each element of that array contains a reference to a hash with `'language'` as its key and the name of the language as the value.

This line:

```
$decoded->{data}->{languages}
```

returns the reference to the languages array. We dereference it (using the @{something} notation) to get at the values in the array, iterate over them, and print the list of values contained in the hash stored in each value.

If your first reaction to this code is "yucko," I don't blame you. It sure seems like a lot of work to get a single list of languages. But if you look carefully at the JSON excerpt that is coming back from the service as printed above, you'll see that the JSON decode() call is faithfully translating the JSON structure into the correct Perl data structure. In the real world you would encapsulate this call into a routine in your code that would construct a more pleasant data structure for the rest of the code to share. C'est la vie.

Translate Something Already!

Let's actually get to the translation process. To do so we have to add a few more twists to our previous code:

1. We have to make sure we're playing by the rules of the (Web) road. Anything we send to Google Translate in a URI must be properly escaped. Luckily, we have a few Perl modules available to us that make this easy.
2. Depending on what sort of code you are writing, you may have to deal with going back and forth between different character encodings (UTF-8, Latin-1, etc.). Character encoding is a serious rabbit hole in itself, so we're not going to talk much about it for fear of being sucked into an entirely separate column. I did sneak one character encoding decision in the previous code. The routine we used from the JSON module, decode_json(), expects to receive a UTF-8 encoded string and interpret it as UTF-8 text. You'll see a similar, albeit more overt, decision in the code we're about to construct.
3. As much as I appreciate the simplicity of LWP::Simple, we are shortly going to be at the point where we are going to have to assert more control over just how the request is constructed and sent out. For example, if the text that you are translating is over a certain length (Google's docs says 5000 characters, but I've seen some indication that 2000 characters might be a safer limit), the request must be sent as a POST instead of the usual GET type. And when you do this, Google Translate requires you also send along a header that says "X-HTTP-Method-Override: GET" (i.e., treat this POST request as if it were a GET). I'm not going to show code that takes this limit into account because it is built into one of the modules we'll see later, but I just wanted to let you know about it should you start writing something on your own and find LWP::Simple can't get this fancy. We'll use its big brother, LWP::Agent, in the examples below, but there are a number of possible modules we could use.

A quick aside that originates from #3 above: in the process of researching this article, I came upon a module I had never seen before called REST::Client. REST::Client describes itself as "A simple client for interacting with RESTful http/https resources." It basically provides a little bit of syntactic sugar that makes the fairly simple task of talking to a REST server even simpler. That's cool, but even cooler is the module based on it called REST::Client::Simple. REST::Client::Simple lets you describe the API you are communicating with (e.g., resources you can access, parameters that can be passed) and then write code that speaks in terms of that API. For example, excerpted from the documentation:

```
 use Net::CloudProvider;

my $nc = Net::CloudProvider(user => 'foobar', api_key => 'secret');
my $response = $nc->create_node({
    id                              => 'funnybox',
    hostname                        => 'node.funnybox.com',
    os                              => 'debian',
    cpus                            => 2,
    memory                          => 256,
    disk_size                       => 5,
});
```

This code started off with a largish definition (not printed here) that said there exists a create_node command performed by making a POST call with certain possible parameters for that resource. Note that the code above doesn't show anything about how the actual POST request is constructed or sent: that's all done in the background by REST::Client::Simple. This level of sophistication is beyond what

we need for our translation example code, but I thought you might find this module handy some day.

Okay, so let's actually translate some stuff. To do so, we'll need to construct a URI with the following parameters:

- key: we saw this before—it is our API key
- q: the string we want to translate
- target: the target language for the translation
- source: the source language, if we don't want Google Translate to try and guess it

Here's a small program that takes an English string as an argument and prints Google Translate's best guess at its French equivalent:

```
use LWP::UserAgent;
use URI::Escape;
use JSON;

my $tkey = '{YOUR Google Translate API KEY HERE}';

my $sl = 'en';
my $tl = 'fr';

my $query = URI::Escape::uri_escape_utf8( $ARGV[0] );

my $ua = LWP::UserAgent->new();

my $results
   = $ua->get( 'https://www.googleapis.com/language/translate/v2/'
      . "?key=$tkey"
      . "&q=$query"
      . "&source=$sl"
      . "&target=$tl" );

die "Translation failed: " . $results->status_line
   unless $results->is_success;

my $decoded = decode_json $results->decoded_content;

binmode(STDOUT, ":utf8");

foreach my $translation ( @{ $decoded->{data}->{translations} } ) {
   print values $translation, "\n";
}
```

This code isn't very different from the previous examples. The code uses a slightly different URI that requests translations, and it adds a few small things to make sure that we stay URI and UTF-8 legal (URI::Escape::uri_escape_utf8 to encode the URI before sending, binmode(STDOUT, ":utf8") to make sure our output to STDOUT is kept UTF-8).

The process is still pretty simple. Let's see how to make it even simpler using a custom module for the job.

WWW::Google::Translate and Beyond

There's a lovely module called WWW::Google::Translate that makes writing code for this API even easier than what we saw above. The module takes into account the encoding, JSON, text size, and other details so you don't have to. It can do spiffy things such as cache the results so future requests for the same text don't force you to use up more of your API calls.

Here's the first example from the docs:

```
use WWW::Google::Translate;

my $wgt = WWW::Google::Translate->new(
    {   key             => '<Your API key here>',
        default_source  => 'en',   # optional
        default_target  => 'ja',   # optional
    }
);
my  $r = $wgt->translate( { q => 'My hovercraft is full of eels' } );
for  my $trans_rh (@{ $r->{data}->{translations} }) {
    print $trans_rh->{translatedText}, "\n";
}
```

Given our previous code, you can probably guess we just need to create a new WWW::Google::Translate object and then ask the module to call a translate() method to get the job done. One quick note about this example: if you run this code using later versions of Perl, you may get a non-fatal error that looks something like this:

```
Wide character in print at Desktop/untitled.pl line 16.
```

This is warning you that you were trying to print Unicode data without preparing STDOUT to receive them. The fix for this is to add the line from above to the program before the print takes place:

```
binmode(STDOUT, ":utf8");
```

Okay, one last addition before we draw this column to a close. If you want to write code that performs translations but isn't directly tied to Google Translate, you may wish to check out the Lingua::Translate set of modules. Lingua::Translate is the closest equivalent to the DBI framework for databases.

With Lingua::Translate, you write the same code independent of the back-end service. Lingua::Translate::Google describes itself as "mostly a wrapper for the WWW::Google::Translate module," so you basically can use the power of WWW::Google::Translate while keeping your code back-end neutral. If later on you decide to stop using Google Translate and switch to something else, very little will have to change in the rest of your program.

Take care and I'll see you next time.