Practical Perl Tools

I Just Called to Say $_

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

If you've noticed a spate of "here's how to use Perl to talk to X Web service" topics in this column lately, it probably isn't a coincidence. I have to confess, I'm a sucker for a Web service that gives you super powers just by using the simple API they provide. For example, in our last column we looked at how to easily translate text in and out of a large number of the world's major languages using Google Translate's API. For this column, we're going to do all sorts of fun things with phones from Perl. If you've ever wanted a way to send and receive SMS and voice messages, retrieve input from a caller, and stuff like that, have I got a column for you.

Like last time, in this column, we're going to be using an API from a commercial vendor. I am not a shill for that vendor. They are not paying me to promote their product. Like last time, I'm actually paying them to use the service. There are other vendors offering similar services. I'm choosing this one because their API is easy to use and the cost for small volumes of use is sufficiently low that it doesn't cost very much to play around (and, in fact, they offer a free account should you want to pay nothing during your playtime). Their API has the added benefit of using things that we've seen in past columns such as a REST and XML. You won't need to reference past columns, but if you're an avid reader of this column (hi mom!), a number of things we'll be looking at should be comfortably familiar.

So who is the lucky vendor this time that gets to take my money? In this column we're going to work with the Twilio API. As we've done in the past, I'm going to hold off on talking about the Twilio-specific Perl modules available for a bit just so we can get a good handle on the basics of what is going on before we let someone else's code do the driving. In this case looking at the underlying stuff is doubly important because Twilio's API and the Perl modules that interact with it assume you understand TwiML, their little mini-XML dialect for command and control. And that's just where we are going to start.

Twinkle, Twinkle, Little TwiML

We're starting a Perl-themed column with an XML dialect because in order to use their service, you'll be slinging TwiML around lots. In the past I've praised XML because it can be superbly readable as long as you don't take pains to thwart this quality (I'm looking at you Microsoft Office). TwiML is no exception; it basically consists of a set of "verb" tags that instruct the service to do something. For example, if we wanted to ask it to send an SMS message, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
   <Sms from="+14105551234" to="+14105556789">
           The king stay the king.</Sms>
</Response>
```

That's a direct quote from their API docs at https://www.twilio.com/docs/api (which I just had to quote because of the embedded reference). If we wanted to call our special Twilio number (more on this shortly) and have it speak to us, we could write:

```
<?xml version="1.0" encoding="UTF-8"?>
<Response>
  <Say>Twinkle, Twinkle, little TwiML</Say>
</Response>
```

The first line is the standard XML declaration. The commands we write will live in a response tag (you'll see why it is called this when we get to talking about REST stuff). Twilio will perform the commands, and if the context is a phone call as in the second example, it will hang up at that point.

There are other verbs available with names that all make sense such as Dial to dial a call, Play to play a sound file on to the call from a specified URL, Record to record sound from the call, and so on. The only one that may not be obvious at first blush is Gather. Gather is used to receive input from a caller (i.e., "Press 1 to speak to Larry Wall…"). We'll see an example of that later in this column.

The REST of the Story

To actually use this stuff, we need to see how to set up conversations with the Twilio's server. This is where the REST stuff we mentioned in the beginning comes in. Let's dive right into some useful examples to see how this works. The TwiML part won't show up until our second example, so we'll set that aside for a brief moment as we look at some code for sending an SMS message:

```
use HTTP::Request::Common qw(POST);
use LWP::UserAgent; # can't use LWP::Simple to POST
use strict;

my $tw_serverURL = 'https://api.twilio.com';
my $tw_APIver    = '2010-04-01';

my $tw_acctsid   = '{YOUR ACCT SID HERE}';
my $tw_authtoken = '{YOUR AUTH TOKEN HERE}';

my $tw_number   = '{YOUR TWILIO NUMBER}';
my $test_number = '{A VALIDATED NUMBER}';

# create the user agent and give it credentials to use
my $ua = LWP::UserAgent->new;
$ua->credentials( 'api.twilio.com:443', 'Twilio API',
            $tw_acctsid, $tw_authtoken );
```

```
# create a request
my $req = POST "$tw_serverURL/$tw_APIver/Accounts/
                $tw_acctsid/SMS/Messages",
   [
   'From' => "$tw_number",
   'To'   => "$test_number",
   'Body' => 'Just a spiffy message!',
   ];

  # ...and send it
my $response = $ua->request($req);

if ( $response->is_success ) {
   print $response->decoded_content;
}
else {
   die $response->status_line;
}
```

Let's walk through this fairly generic LWP::UserAgent code together. After loading the modules we'll need, we define some variables that include the URL we're going to contact (their server plus API version), the API SID and token (user name and password we get when we sign up), and the numbers we'll be using. When you sign up for Twilio, you are given the opportunity to choose a number from which your text and voice messages will be sent and received. Once you pay for the service, you are also able to purchase additional numbers. At demo signup time you are also prompted to validate a number (i.e., prove you own it)—their system calls you and asks you to provide a passcode the Web site shows you—to act as a test number. You will need to use this test number as the source/destination number to call or be called by your Twilio number until you become a paying customer.

With these things defined, we can create a UserAgent object (the thing that is going to pretend to be a browser) and give it the credentials it will need to access that REST API URL. We construct the request by specifying the URL and the parameters we'll want to pass in when we do the POST. We send it off, and then print the response we get back. Here's a sample response that I've pretty printed so it is easier to read:

```
<?xml version="1.0"?>
  <TwilioResponse>
     <SMSMessage>
        <Sid>SMb70e55827ff00117fd88060902ffddddd</Sid>
        <DateCreated>Mon, 26 Nov 2012 02:56:44 +0000</DateCreated>
        <DateUpdated>Mon, 26 Nov 2012 02:56:44 +0000</DateUpdated>
        <DateSent/>
        <AccountSid>{MY ACCOUNT SID}</AccountSid>
        <To>{THE VALIDATED NUMBER}</To>
        <From>{MY TWILIP NUMBER}</From>
        <Body>Just a spiffy message!</Body>
        <Status>queued</Status>
        <Direction>outbound-api</Direction>
        <ApiVersion>2010-04-01</ApiVersion>
        <Price/>
```

```
            <Uri>/2010-04-01/Accounts/{MY_ACCOUNT_SID}/SMS/Messages/
            SMb70e55827ff00117fd88060902ffddddd</Uri>
        </SMSMessage>
    </TwilioResponse>
```

It is basically an echo of the message we sent, but I want to draw your attention to one of the elements:

```
<Status>queued</Status>
```

When you send a message, it gets queued to be sent. Unlike some services, you do not stay connected to the server until the message is actually sent. This means you don't get a definitive response code back from the request that indicates success or failure on the sending. How you get the response back brings us to the REST stuff...

In my request, I didn't include the optional StatusCallback parameter. If I were to include that in my request, e.g.,

```
 my $req = POST "$tw_serverURL/$tw_APIver/Accounts/$tw_acctsid/SMS/
Messages",
       [
       'From'          => "$tw_number",
       'To'           => "$test_number",
       'Body'          => 'Just a spiffy message!',
       'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
       ];
```

Twilio would fire off a POST request to that URL once the message has gone through (or not). This callback style of programming shows up throughout the API, so if you plan to do much with it you'll need a Web server where you can place code that will receive messages from their server. Here's a very simple example we could use as messagestat.pl to receive a message from them:

```
use CGI;
use Data::Dumper;

my $q = CGI->new;
my $params = $q->Vars;

open my $OUTPUT, '>>', 'twilio.out' or die "Can't write output: $!";
print $OUTPUT Dumper \$params;
close $OUTPUT;
```

This receives the POST from Twilio's servers (and anyone who can contact that URL) and writes the parameters of the request to a file. If we look at the contents of the file, we see it says:

```
$VAR1 = \{
             'AccountSid' => '{MY_ACCOUNT_SID}',
             'SmsStatus' => 'sent',
             'Body' => 'Just a spiffy message!',
             'SmsSid' => 'SMe1620214513ccee199851ad9f13ffff',
             'To' => '{THE VALIDATED NUMBER}',
             'From' => '{MY TWILIP NUMBER}',
             'ApiVersion' => '2010-04-01'
           };
```

Here we can see that the SmsStatus was 'sent', so the message went out. If for some reason it couldn't be sent successfully, that would have been reflected in the status posted to our CGI script.

Early in this section I mentioned "conversations with their server," but the last example didn't offer anything particularly scintillating in this regard. Let's do something more sophisticated, this time with a voice call instead of an SMS message. Let's do a two-question telephone poll from Perl using Twilio. This will allow us to bring the TwiML we learned earlier back into the picture.

The first step is to tell Twilio's servers to initiate a voice call. I'm going to leave out all of the initialization code from the example below to save space because it is exactly the same as the previous example. Here's the one part of the code that changes:

```
my $req = POST "$tw_serverURL/$tw_APIver/Accounts/$tw_acctsid/Calls",
   [
   'From'          => "$tw_number",
   'To'            => "$test_number",
   'Url'           => 'http://your.Web.server.com/voicepoll.pl',
   'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
   ];
```

The first change is we're requesting a different kind of REST object; we're asking for Calls instead of SMS/Messages. The second change is we've told Twilio that once it initiates a call, it should contact the voicepoll.pl script for further instructions to follow once the call has connected. And this is where TwiML becomes important.

The URL pointed to by the Url parameter is expected to provide Twilio's server with a TwiML document it should process. Here's our voicepoll.pl script that will provide this document:

```
use CGI qw(:standard);
use strict;

my $q      = CGI->new;
my $params = $q->Vars;

print $q->header('text/xml');

if ( not $params->{'Digits'} ) {
   print <<POLL;
<?xml version="1.0" encoding="UTF-8"?>
  <Response>
   <Gather numDigits="1" action="/voicepoll.pl">
      <Say>Welcome to the login poll</Say>
      <Say>Press 1 if you are happy and you know it</Say>
      <Say>Press 2 if you really want to show it</Say>
   </Gather>
   <Say>No input, toodles!</Say>
  </Response>
POLL
}
```

```
else {
   if ( $params->{'Digits'} ne "3" ) {

      open my $RESPONSE, '>>', 'twresp.out'
         or die "Can't write to twrestp.out";
      print $RESPONSE "Received $params->{'Digits'}\n";
      close $RESPONSE;

      print <<POLL;
<?xml version="1.0" encoding="UTF-8"?>
     <Response>
       <Gather numDigits="1" action="/voicepoll.pl">
         <Say>Next Question</Say>
         <Say>Press 1 if you are happy and you know it</Say>
         <Say>Press 2 if you really want to show it</Say>
         <Say>Press 3 if you are suffering from ennui</Say>
       </Gather>
         <Say>No input, toodles!</Say>
     </Response>
POLL
}
else {
      open my $RESPONSE, '>>', 'twresp.out'
         or die "Can't write to twrestp.out";
      print $RESPONSE "END POLL\n";
      close $RESPONSE;

      print <<BYEBYE;
<?xml version="1.0" encoding="UTF-8"?>
         <Response>
             <Say>End of Poll, thanks!</Say>
         </Response>
BYEBYE
   }
}
```

Here are two small caveats before we look at what the script and its embedded TwiML is doing. First, this script is going to spit out TwiML in the most straightforward but uncouth way possible. It's just a bunch of print statements using HEREDOC syntax (<<). If you were doing this for real, you'd want to use some sort of XML generator or one of the custom Twilio modules we'll get to in a moment. Second, all TwiML fetched from the servers is coming from this one script with a bunch of dumb control logic. In real life it might make more sense to have different responses to different queries from their servers handled by different scripts ("Press 1 for Sales" then points to the sales.pl script and so on).

Let's walk through what is going on one step at a time. We used a modified version of our SMS script to ask Twilio to initiate a call and then have it fetch the URL for the CGI script above. This CGI script checks to see whether it has received a parameter called 'Digits' for reasons you'll see in just a second. If that parameter isn't defined yet (true because this will be the first time it has been accessed by Twilio for this call), it prints the following TwiML back to their server:

```
<?xml version="1.0" encoding="UTF-8"?>
 <Response>
  <Gather numDigits="1" action="/voicepoll.pl">
     <Say>Welcome to the login poll</Say>
     <Say>Press 1 if you are happy and you know it</Say>
     <Say>Press 2 if you really want to show it</Say>
  </Gather>
  <Say>No input, toodles!</Say>
 </Response>
```

This TwiML uses "Gather," a verb we haven't seen before. Gather will attempt to read keypad input from the call (i.e., the caller pressed the phone's number buttons). In the TwiML, there are two attributes being passed in for Gather: "numDigits" for the number of digits we hope to get back (one) and "action" for the URL that will be called if Gather is successful.

As I mentioned above, the TwiML makes another call to the voicepoll.pl script in <Gather>, but it could easily have been told to fetch some other CGI script for its next batch of TwiML. Embedded in this Gather call are a number of <Say> elements used to speak the menu for the person on the line. Because they are listed as sub-elements of Gather, this means that the Gather is active while they are speaking. The caller doesn't have to wait to press a button. She or he can interrupt the <Say> directives, and the Gather will complete and immediately pass the results to the URL specified in the action attribute (bypassing anything else in this TwiML file). The action URL is used as the source of the next TwiML directive, and the control flow continues using whatever TwiML it provides. Should the <Gather> fail, e.g., time out if it doesn't get input, the directives outside of the <Gather> are run. In this case, a final <Say> command will bid the caller adieu before hanging up.

Let's assume the <Gather> was able to retrieve a choice from the caller and see what happens next. The CGI script specified in its action attribute gets called with the results from the Gather command being passed as a parameter called 'Digits'. This is why the script looks to see whether it has received that parameter. If it hasn't, it knows it is the first time it is being called. If it gets the number 3 in that parameter, it will immediately print the "BYEBYE" TwiML code. Any other number tells the script to print the Next Question TwiML. All along the way, we collect the results we received to a response file that accumulates lines like:

```
Received 2
Received 5
Received 1
END POLL
```

This is just to show off the input we received. If we really cared about it we'd want to store it in a more considered way, like a database. At the very least, we'd want a way to store things so multiple polls going at once record their results properly. And speaking of results, in the script that originated the call, we kept the

```
'StatusCallback' => 'http://your.Web.server.com/messagestat.pl',
```

line. Just like with an SMS message, this URL gets called after the operation has been completed (successfully or unsuccessfully). In the case of a voice message, we get a cool set of parameters posted to the messagestat.pl URL:

```
'AccountSid' => '{MY_ACCOUNT_SID}',
        'ToZip' => '02283',
        'FromState' => 'MA',
        'Called' => '{THE VALIDATED NUMBER}',
        'FromCountry' => 'US',
        'CallerCountry' => 'US',
        'CalledZip' => '02283',
        'Direction' => 'outbound-api',
        'FromCity' => 'CAMBRIDGE',
        'CalledCountry' => 'US',
        'Duration' => '1',
        'CallerState' => 'MA',
        'CallSid' => 'CA4fdd2c584cd58230f9e7413c452fffff',
        'CalledState' => 'MA',
        'From' => '{MY TWILIO NUMBER}',
        'CallerZip' => '02139',
        'FromZip' => '02139',
        'CallStatus' => 'completed',
        'ToCity' => 'BOSTON',
        'ToState' => 'MA',
        'To' => '{THE VALIDATED NUMBER}',
        'CallDuration' => '17',
        'ToCountry' => 'US',
        'CallerCity' => 'CAMBRIDGE',
        'ApiVersion' => '2010-04-01',
        'Caller' => '{MY TWILIO NUMBER}',
        'CalledCity' => 'BOSTON'
```

Yup, a little bit of geolocation is thrown in for free.

There's one last category of operations I want to mention (but not demonstrate for space reasons). So far we haven't seen any code for the case where someone calls in to your Twilio number or sends an SMS message to it. When you set a Twilio number up, you associate two URLs with it: one for voice, the other for SMS. When a call or an SMS message comes in to that number, Twilio attempts to post information about the incoming call/message to the appropriate URL (as parameters, same as we've seen before) and expects to be handed back some TwiML telling it what to do. That CGI script can do whatever you need with the incoming information (e.g., log the parameters) and direct Twilio to do something (like start a phone poll or take an order for a pizza).

WWW::Twilio::API and WWW::Twilio::TwiML

I'd like to end with a quick look at the two special purpose Perl modules for interacting with Twilio. The first lets you make API calls without having to trouble your pretty little head with all of the LWP::UserAgent details. Instead of our first code example, we could write:

```
use WWW::Twilio::API;

  my $twilio = WWW::Twilio::API->new(
     AccountSid => '{MY ACCOUNT SID}',
     AuthToken  => '{MY ACCOUNT TOKEN}',
  );
```

```
 my $response = $twilio->POST(
    'SMS/Messages',
    'From' => '{MY TWILIO NUMBER}',
    'To'   => '{THE VALIDATED NUMBER}',
    'Body' => 'Just a spiffy message!'
);
```

WWW::Twilio::API lets you use all of the other API calls we've seen before. For example, in the WWW::Twilio::API doc, we see an example of making a call:

```
$twilio->POST(
    'Calls',
    To   => '5558675309',
    From => '4158675309',
    Url  => 'http://www.myapp.com/myhandler'
);
```

The other special purpose Twilio module courtesy of the same author is WWW::Twilio::TwiML. It is designed to make authoring TwiML easier, but I'll say up front that I'm not entirely clear it is much easier to use than any of the other XML authoring modules that are available. I think it holds the most promise for people who enjoy writing chained method expressions (i.e., code with lots of thing->thing->thing statements). For example, if we wanted to output the first set of TwiML we printed in voicepoll.pl above, we would write:

```
use WWW::Twilio::TwiML;

my $twiml = WWW::Twilio::TwiML->new;

$twiml
    ->Response
      ->Gather( { action => '/voicepool.pl' } )
    ->Say('Welcome to the login poll')
    ->parent->Say('Press 1 if you are happy and you know it')
    ->parent->Say('Press 2 if you really want to show it')
    ->parent->parent->Say('No input, toodles!');

print $twiml->to_string;
```

The chained statement can be read something like this: "Create a <Response> element. In this element create a <Gather> element. In the <Gather> element, create a <Say> element. Now, instead of creating the next <Say> within the current <Say> element, put it in the parent (the <Gather>). Do that again for the next <Say>. Then, go to that element's grandparent (the <Response> element) and place a final <Say> element in it." If your brain has no problems mapping the chained steps to the process of building our little XML tree structure, great, this might be the module for you. If not, seek another solution.

And with that, we now have a good start on how to use Twilio's API from Perl to do all sort of fun phone-related stuff. Take care and I'll see you next time.