Practical Perl Tools
Off the Charts

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 systems 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'. dnb@usenix.org

I recently had the pleasure of giving another invited talk to the LISA 2016 audience. Part of preparing that talk involved performing some basic forensics on a machine that I could no longer directly access. I wanted to explore how its file systems had changed over time. I like pretty pictures, so my first goal was to attempt to draw a diagram that represented this evolution. In this column, we'll take a look at the code I wrote to achieve this. Just a warning up front: there are a number of moving parts for the approach I took (not all of which are Perl), but I'll do my best to explain all of the plates that are being spun.

The Best Tool for Storing Data Is a...

When I started out, I wasn't exactly clear what sort of representation of the data I needed or even what was the best way to wade through the information I had on hand. I was given access to a set of directory listings (basically the output of recursive `ls -lR {something} > output` commands for the file system with different flags used). The beginning of one of these files looked like this:

```
/etc:
total 851
drwxr-xr-x   67 root     sys            5120      Sep 19 12:19 .
drwxr-xr-x   39 root     root           2048      Jan  8  2016 ..
drwxr-xr-x    2 adm      adm             512      May 15  2006 acct
lrwxrwxrwx    1 root     root         14 May 15    2006 aliases -> ./mail/aliases
drwxr-xr-x    2 root     bin         512 May  5    2009 apache
drwxr-xr-x    2 root     bin         512 May 15   2006 appserver
-rw-r--r--    1 root     bin          50 May 15   2006 auto_home
-rw-r--r--    1 root     bin         113 Mar 20   2008 auto_master
-rw-r--r--    1 root     other     47389 Mar 31   2009 bootparams
-rw-r--r--    1 root     other     47389 Mar 31   2009 bootparams.old
-rw-r--r--    1 root     other     47397 Mar 27   2009 bootparams.orig
lrwxrwxrwx    1 root     root         18 May 15    2006 chroot -> ../usr/sbin/chroot
-rw-r--r--    1 root     other          314      Jun 15 15:04 coreadm.conf
lrwxrwxrwx    1 root     root         16 May 15    006 cron -> ../usr/sbin/cron
drwxr-xr-x    2 root     sys             512      Jun 15 15:05 cron.d
```

My thinking was that if I could get all of this information into a database, it would allow me to play around with the data through ad hoc queries. The tricky part was parsing the files because, as you can see above, it is basically a hot mess. The actual directory name itself appears in a different format before the entries. Some files have explicit years in their dates, some do not. Some files aren't even files (in the classic sense), they are links to files or directories. Whee!

Output like this is notoriously hard to parse as a Web search on the question will quickly reveal. Much to my delight, it turns out that a module that attempts to handle this unpleasantness actually exists called File::Listing. Here's the code I wrote to use that module to slurp the contents of a directory listing into a SQLite database. SQLite was used because of its lightweight nature and ability to install as part of a single Perl module (DBI::SQLite) install. This was easier than installing/configuring a database, its libs, and a separate Perl module before I could make progress.

```
use strict;
use DBI qw(:sql_types);

my $file = shift;

open my $L, '<', $file or die "Can't open $file:$!\n";
my $dir = File::Listing::parse_dir( $L, undef, 'unix', 'warn' );
close $L;

my $dbh = DBI->connect( "dbi:SQLite:dbname=$file.db", "", "" );

# throw an error if something fails so we don't have to check
# the results of every statement and turn off committing the
# data to the file on every insert
$dbh->{RaiseError} = 1;
$dbh->{AutoCommit} = 0;

$dbh->do(
  "CREATE TABLE dir (filename text, filetype text, filesize
integer, filetime integer, filemode text)"
);

my $sth = $dbh->prepare(
  "INSERT INTO dir (filename, filetype, filesize, filetime,
filemode) VALUES (?,?,?,?,?)"
);

my $rowcount = 0;
foreach my $listing (@$dir) {
   $sth->execute(@$listing);
   if ( $rowcount++ % 1000 == 0 ) {
      $dbh->commit;
      print STDERR ".";
   }
}

$dbh->commit;
$dbh->disconnect;
```

We've talked about using DBI (the Perl DataBase Independent) framework before, so we won't go into depth about how that part of this code works. Instead, let me just give a brief summary of what is going on and mention a few salient points. The first thing this code does is read in the listing file as specified on the command line and parse it. The file gets parsed and stored in memory (the listings I had were only about 30 MB so I could get away with it).

We then do the DBI magic necessary for "connecting" to a SQLite database file (creating it if it does not exist—in this case we use the name of the listing file as the start of that database file name), create a table called "dir" into which we'll store the info, and then start to populate it. We iterate through the parsed file info we have in memory, inserting the info into the database. After every 1000 records, we actually commit those inserts to the file and print a dot to let us know the process is working. We didn't have to turn off autocommit and commit explicitly like this, but we get a wee bit of a performance boost if we do so. After the script runs we are left with a nice `filename.db` SQLite file we can query to our heart's content.

After much playing around with SQL queries and Web searches about SQLite SQL queries, I finally hit upon this SQL statement to do what I needed:

```
SELECT strftime('%Y-%m', filetime, 'unixepoch') yr_mon,count(*)
num_dates FROM dir GROUP BY yr_mon;
```

It produces results that looks like this:

```
yr_mon:num_dates
1973-05:1
1992-09:1
1993-04:2
1993-06:1
1993-07:4
1993-08:12
1993-09:4
1993-10:6
1993-11:1
1993-12:1
...
```

Here we have the number of files created in each of the listed months (e.g., in August of 1993, 12 files were created). Now all we have to do is represent this information in a chart.

Google Charts Ho!

I could have fed these results into any number of applications or services that draw graphs, but I thought it might be fun to learn how to use Google Charts from Perl. Plus, it had a wide variety of charts available, so I thought it would be good to hedge my bets.

The tricky thing with Google Charts is it is not meant to run client-side. We won't be running a program on local data and have it spit out a chart. Instead, the process is roughly: you load a Web page, that Web page loads some Google Chart libraries, creates the necessary JavaScript objects, makes a call to get the data for the chart (if it isn't embedded in the page), and then asks Google to return the desired chart, which is shown by your

browser as embedded in the Web page. If that sounds like a little bit of work, it definitely is (at least the first time you are trying it). We'll go slow.

One thing to note here is that in order for this to work, you will need to load this Web page (and its surprise guest that we'll get to in a moment) from a Web server. You can't just load it from the File->Open menu items in your browser. Your server doesn't have to be anything high-powered (I used Apache via MAMP PRO running on my laptop to serve the files, but that was just because it was already handy), but you do need one for Google Charts to function properly.

The first thing to do is to create the Web page mentioned above. It is going to have a small amount of HTML and a bunch of JavaScript. The docs at https://developers.google.com/chart are really quite good, so you can get very far via simple cut-and-pasting even if your JavaScript isn't so hot (phew). Here's the .html page I used (I'll break it down in a sec):

```
<head>
  <script type="text/javascript" src="https://www.gstatic.com/
charts/loader.js"></script>
  <script type="text/javascript" src="//ajax.googleapis.com/
ajax/libs/jquery/1.10.2/jquery.min.js"></script>
  <script type="text/javascript">
     google.charts.load('current', {'packages':['scatter']});
     google.charts.setOnLoadCallback(drawChart);

     function drawChart () {

           var jsonData = $.ajax({
                 url: "get_data.pl?filename=listing.db",
                 dataType: "json",
                 async: false
                 }).responseText;


         var data = new google.visualization.
DataTable(jsonData);

         var options = {
           width: 2000,
           height: 700,
           chart: {
             title: 'File Creation Dates',
                          subtitle: 'listing',
           },
           hAxis: {title: 'Date'},
           vAxis: {title: 'Number'}
         };

         var chart = new google.charts.Scatter(document.
getElementById('scatterchart_material'));
         chart.draw(data,   google.charts.Scatter.
convertOptions(options));
     }
   </script>
  </head>
  <body>


     <div id="scatterchart_material"></div>
  </body>
</html>
```

Much of the above is straight from the docs, so I'll just briefly mention what is going on. It can be a bit of a challenge to read because most of the listing consists of definitions that get triggered at the right moment. After we load the right libraries and set up something that will cue the function that does all of the work after everything is loaded, we define that function `drawChart()`. In `drawChart()` we specify how we are going to pull the data (more on that in a moment), various options on how the chart should look, what HTML element in the document will "hold" the resulting chart, followed by a call to actually kick off the drawing. When the page loads, it will call `drawChart()` and we are off to the races.

How Does the Data Get into the Chart?

Yeah, that's one of the fun questions. The key part was in our description of how the data should be loaded:

```
var jsonData = $.ajax({
                      url: "get_data.pl? filename=listing.db",
                      dataType: "json",
                      async: false
                     ).responseText;
var data = new google.visualization.DataTable(jsonData);
```

Google Charts lets you specify the data for a chart inline (i.e., you can put JSON right in the .html file), but that only works for smallish data sets. The chart I was hoping to build had 278 rows of data, so I wasn't keen on embedding that all in the same doc. Instead, we're going make an AJAX call to another URL (`get_data.pl`) and ask it to send us the data set. We can then take those results and put them in the proper object for graphing.

Time for some more Perl. We'll need a CGI script that will query our database using the SELECT statement we previously saw and format the results into the proper JSON output expected by the Google Charts API. When I heard the requirements "CGI" and "JSON output," a couple of the frameworks we've seen in this column before (Mojolicious and Dancer) leapt right to mind. My choice was cemented when I saw Joel Berger's excellent blog post "Some code ports to Mojolicious, just for fun" [1]. It was a post about porting another person's work on Google Charts from

Perl to Mojolicious::Lite. The code we're about to see is a direct descendant of Joel's example with a few fun twists.

Let's take this task piece by piece. The CGI portion of the script is only a few lines:

```
use Mojolicious::Lite;
use DBIx::Connector;

any '/' => sub {
   my $c = shift;

   my $filename = $c->param('filename');
   my $data = $c->get_data($filename);
   $c->render( json => $data );
};
```

This just says that when a request for the URL "/?filename=something" comes in we will parse out the parameter (the file name of the database we'll be using), the proper database query will be made and results returned in the right form, and this will be converted into JSON and sent to the requester.

More interesting are the two helper functions we will define. The first is responsible for getting us a safe database handle for the right SQLite database:

```
helper db => sub {
   my $filename = $_[1];
   state $db =
     DBIx::Connector->connect( "dbi:SQLite:dbname=$filename",
'', '' );
};
```

By the way, if you haven't seen DBIx::Connector before (I hadn't), it is worth looking up because it is quite spiffy.

Now for the more complex part of the script, a helper that does the actual query for data and then transforms the results so the JSON will be correct.

First step, perform the actual query:

```
helper get_data => sub {


   my $filename = $_[1];
   my $db = shift->db($filename);

   my $query =
"SELECT strftime('%Y-%m', filetime, 'unixepoch') yr_mon,count(*)
num_dates FROM dir GROUP BY yr_mon;";

   my $data = $db->selectall_arrayref($query);
```

Now for some annoying stuff. Google Charts expects to receive the data set in a very specific JSON format. This means we're going to have to transform the data coming out of the database in a very particular way such that we match the format expected when the Perl data structure to JSON conversion is made.

The JSON format Google Charts expects looks like this [2]:

```
{
 cols: [{id: 'A', label: 'NEW A', type: 'string'},
       {id: 'B', label: 'B-label', type: 'number'},
       {id: 'C', label: 'C-label', type: 'date'}
 ],
 rows: [{c:[{v: 'a'},
          {v: 1.0, f: 'One'},
          {v: new Date(2008, 1, 28, 0, 31, 26), f: '2/28/08
12:31 AM'}
       ]},
        {c:[{v: 'b'},
          {v: 2.0, f: 'Two'},
          {v: new Date(2008, 2, 30, 0, 31, 26), f: '3/30/08
12:31 AM'}
       ]},
        {c:[{v: 'c'},
          {v: 3.0, f: 'Three'},
          {v: new Date(2008, 3, 30, 0, 31, 26), f: '4/30/08
12:31 AM'}
       ]}
 ]
}
```

I'm going to describe this JSON blob in terms of Perl data structures because I think it will make it easier to understand the Perl code we are about to see. You can look at this like a hash with two keys, 'cols' and 'rows'. The cols part is basically a definition of the contents of the rows that will follow. If it helps, think of this as the column heading of a spreadsheet followed by a bunch of rows.

The cols portion is constructed from an array that holds three separate hashes, one for each column being defined. So the first column has a key of 'id' whose value is "A," a key of "label" whose value is "NEW A," and a key of "type" whose value is "string." This is how we specify the id of the first column, how it will be labeled, and what kind of values it will contain.

Here's how we build our version of that part of the data structure in Perl:

```
   my $response->{'cols'} = [
      { 'id'    => 'Date',
        'label' => 'date',
        'type'  => 'string' },
      { 'id'    => 'Count',
        'label' => 'count',
        'type'  => 'number' },
   ];
```

This code creates a similar array with two hashes in it, one for each column. We'll have a column for the date (e.g., "1993-08") and the number of files in that time period ("12").

Now let's tear apart one of the rows. A row consists of an array of cells containing values (that makes sense, yes? if just from your use of spreadsheets).

Here's an example row (the first one):

```
rows: [{c:[{v: 'a'},
        {v: 1.0, f: 'One'},
        {v: new Date(2008, 1, 28, 0, 31, 26), f: '2/28/08
12:31 AM'}
      ]},
```

It shows a row that consists of cells containing the values "a," "1.0," and a newly defined date. So, something like this:

```
|a|1.0|2/28/08 12:31 AM|
```

Each value is stored in a hash with the key 'v' (for value). There is also another (optional) key 'f' in the example above for "formatted value" (i.e., how the value should be displayed).

To review:

- We're going to create a key called 'rows' in the hash we created above when defining the columns.
- 'rows' will contain an array that holds the hashes defining the cells in each row.
- Each cell hash needs to have a single key of 'c' to mark it as cell data.
- 'c' will hold an array of hashes, each representing a cell value.
- Each hash holding a value will have a key called 'v' whose value is the value for that cell.

Phew! See why I call this annoying? Now let's take on creating this in Perl:

```
foreach my $row (@$data) {
   my $c->{'c'} = [ map { { 'v' => $_ } } @$row ];
   push( @{ $response->{'rows'} }, $c );
}
```

It may be a bit surprising to realize all of that rigmarole can be implemented in just three lines of code. It is probably not surprising that it is three lines of fairly gnarly/compact code. Let's unravel it so it all makes sense.

The DBI query we made above

```
$db->selectall_arrayref($query);'
```

returns a reference to an array containing the results of our query:

```
0  ARRAY(0x7fccb45eaa90)
   0  ARRAY(0x7fccb45ebca0)
      0  '1973-05'
      1  1
   1  ARRAY(0x7fccb45eb060)
      0  '1992-09'
      1  1
   2  ARRAY(0x7fccb4002c68)
      0  '1993-04'
      1  2
   3  ARRAY(0x7fccb45ec880)
      0  '1993-06'
      1  1
   4  ARRAY(0x7fccb45ec988)
      0  '1993-07'
      1  4
   5  ARRAY(0x7fccb45eb090)
      0  '1993-08'
      1  12
```

As you can see, each array is a row from the results of the query. Our code is going to iterate over these results, one row/array at a time:

```
foreach my $row (@$data) {
```

For each value in the results, we're going to return an anonymous hash with a key of 'v' whose value is the value in the result. Here's the part that creates the anonymous hash for a value:

```
{ 'v' => $_ }
```

That's for a single value in the results. Here's how we iterate over all of the values in the results array, returning anonymous hashes as we go:

```
map { { 'v' => $_ } } @$row
```

We collect all of the {v}=something hashes the map{} returns into an array

```
[ map { { 'v' => $_ } } @$row ]
```

and stuff that array into a hash under the key of 'c' (representing the cells of the row):

```
my $c->{'c'} = [ map { { 'v' => $_ } } @$row ];
```

At this point, we now have a hash for the cells of that row. Go us! We need to store that hash into the array holding all of the rows of cells, so we append it to that array:

```
push( @{ $response->{'rows'} }, $c );
```

If you find all of the punctuation in that line confusing, don't feel bad. Here's what's going on:

```
$response->{'rows'}
```

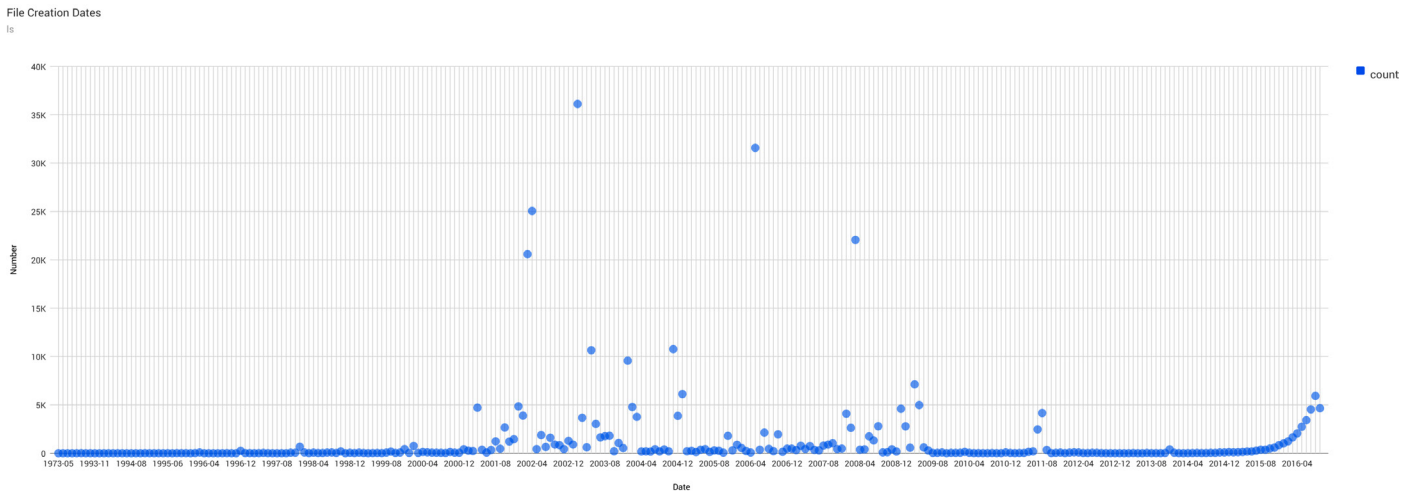

Figure 1: This chart appears in our browser when we load our Web page, and it all comes together.

$response is a reference to an anonymous hash, which has a key called 'rows'. So far so good?

The value for this key is a reference to an anonymous array (the one that is going to hold all of the row information). We need to de-reference it to get at the array itself, hence:

```
@{ $response->{'rows'} }
```

Once we've done that, we can add this set of cells as another row in that array:

```
push( @{ $response->{'rows'} }, $c );
```

And with that, we've done the work of retrieving the info from the database and transforming it into the right data structure.

If that felt a bit painful, I'll be the first to agree. It took me a while to build all that up piece by piece. To add insult to injury, well after I had completed the work I happened to stumble on the module Data::Google::Visualization::DataTable, which describes itself as "attempts to hide the gory details of preparing your data before sending it to a JSON serializer—more specifically, hiding some of the hoops that have to be jump[ed] through for making sure your data serializes to the right data types."

Sigh.

It hadn't come up during any of my other searches for Google Chart modules, so I (and now you) learned how to do it the hard way.

The last step for the CGI script is to translate the data structure into JSON and send it along to the requester; this happens in the last line of the script because Mojolicious::Lite makes it this simple:

```
any '/' => sub {
    my $c = shift;
    my $filename = $c->param('filename');
    my $data = $c->get_data($filename);
    $c->render( json => $data );
};
```

If we browse to the page we made, we get the lovely graph shown in Figure 1.

As I said, there are a number of moving parts. But once you get a sense of how they all work, you now get to bring to bear all of the power Google Charts has to offer you. Take care, and I'll see you next time.

References

[1] http://blogs.perl.org/users/joel_berger/2013/10/some-code-ports-to-mojolicious-just-for-fun.html.

[2] Google Visualization API Reference: https://developers.google.com/chart/interactive/docs/reference.