DAVID N. BLANK-EDELMAN

practical Perl tools: back in timeline

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 *Perl for System Administration*. He has spent the past 22+ years as a system/network administrator in large multiplatform 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.

dnb@ccs.neu.dnb@ccs.neu.edu

YOU KNOW, I WAS JUST MINDING MY own business, reading my email and stuff, when the following message from the SAGE mailing list came on my screen (slightly excerpted but reprinted with permission):

> From: millerj@metro.dst.or.us
> Date: January 9, 2008 2:10:14 PM EST
> Subject: Re: [SAGE] crontabs vs /etc/cron.[daily,hourly,*] vs /etc/cron.d/
>
> On a more specific aspect of this (without regard to best practice), does anyone know of a tool that converts crontabs into Gantt charts? I've always wanted to visualize how the crontab jobs (on a set of machines) line up in time. Each entry would need to be supplemented with an estimate of the duration of the job (3 minutes vs 3 hours).
>
> JM

I just love sysadmin-related visualization ideas. This also seemed like a fun project with some good discrete parts well-suited to a column. Let's build a very basic version of this project together. For the purpose of this discussion I'm going to make the assumption that you already know what a crontab file is, what it contains, and what it does for a living. If not, please consult your manual pages about them and cron (try typing something like man 5 crontab or just man crontab).

Chewing on the Crontab File

The first subtask that comes up with this project is the parsing and interpretation of a standard crontab file. The easy part will be to read in the file and have our program make sense of the individual fields in that file. Having a crontab sliced and diced into nice bite-sized (read: object) pieces doesn't help us all that much, because our end goal is to be able to plot what happens when cron interprets those pieces. Cron looks at that file and decides when a particular command should be run. We'll need some way to determine all of the times cron would have run a particular line during some set time period.

For example, let's say we take a very basic crontab file like this:

```
45 * * * * /priv/adm/cron/hourly
15 3 * * * /priv/adm/cron/daily
15 5 * * 0 /priv/adm/cron/weekly
15 6 1 * * /priv/adm/cron/monthly
```

Every 45 minutes, the /priv/adm/cron/hourly program is run, so we'll be plotting that event at 1:45, 2:45, 3:45, and so on. At 3:15 in the morning each day we run /priv/adm/cron/daily, and so on.

Figuring all of this out seems doable, but, truth be told, kind of a pain. Luckily we've been spared that effort because Piers Kent wrote and published the module Schedule::Cron::Events, which makes this subtask super easy. It calls upon another module to parse a crontab line (Set::Crontab by Abhijit Menon-Sen) and then provides a simple interface for generating the discrete events we'll need.

To use Schedule::Cron::Events, we'll need to pass it two pieces of information: the line from crontab we care about and some indication of when we'd like Schedule::Cron::Events to begin calculating the events created by that crontab line:

```
my $event = Schedule::Cron::Events( $cronline, Seconds => {some time} );
```

(where {some time} is provided using the standard convention of describing time as the number of seconds that have elapsed since the epoch).

Once you've created that object, each call to `$event->nextEvent()` returns back all of the fields you'd need to describe a date (year, month, day, hour, minutes, second).

Now that we understand how to deal with this subtask, let's move on to the others. We'll put everything together at the end.

Displaying the Timeline

Creating a pretty timeline is a nontrivial undertaking, so let's let someone else do the work here for us as well. There are decent Perl timeline representation (Data::Timeline) and display (Graph::Timeline) modules available, but there's one way to create timelines that are so spiffy that I'm actually going to forsake the pure-Perl solution. I think the Timeline (as they put it) "DHTML-based AJAXy widget for visualizing time-based events" project from the SIMILE project at MIT is very cool and a good fit for this project. More info on it can be found at http://simile.mit.edu/timeline/. To give you an idea of what Timeline's output looks like, see the excerpt from Monet's life shown in Figure 1.

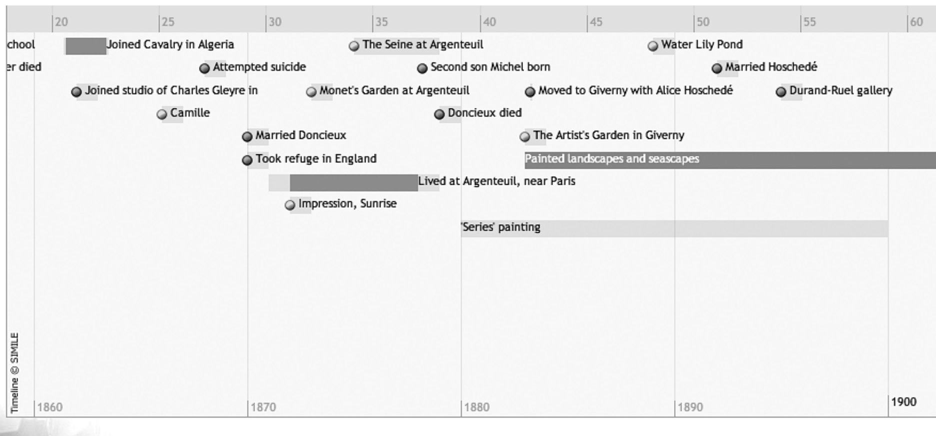

FIGURE 1: TIMELINE MONET EXAMPLE SCREENSHOT

To make use of this widget we need to create two files: an HTML file that sucks in the widget from MIT, initializes it, and displays it and an XML file containing the events we want displayed. That last part will be our third

challenge, which we'll address in the next section. In the meantime, let me show you the HTML file in question. I should mention that my Javascript skills are larval at best; most of the following is cribbed from the tutorial found at the URL provided above. If this is all gobbledygook to you, feel free to just read the comments (marked as <!-- --> and //).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <!-- Reference the widget -->
    <script src="http://simile.mit.edu/timeline/api/timeline-api.js" type="text/javascript">
    </script>

    <script type="text/javascript">

    function onLoad() {
     // tl will hold the timeline we're going to create
     var tl;
     // get ready to specify where we'll get the data
     var eventSource = new Timeline.DefaultEventSource();

     // Create a timeline with two horizontal bars, one displaying
     // the hours, the other the days that contain the hours.
     // Note: both bands are set to display things relative
     // to my timezone (-5 GMT).
     var bandInfos = [
         Timeline.createBandInfo({
            eventSource:    eventSource,
            timeZone:       -5, // my timezone in Boston
            width:          "70%",
            intervalUnit:   Timeline.DateTime.HOUR,
            intervalPixels: 100 }),
         Timeline.createBandInfo({
            timeZone:       -5,
            width:          "30%",
            intervalUnit:   Timeline.DateTime.DAY,
            intervalPixels: 100 }),
         ];

     // keep the two bands in sync, highlight the connection
     bandInfos[1].syncWith = 0;
     bandInfos[1].highlight = true;

     // ok, create a timeline and load its data from output.xml
     tl = Timeline.create(document.getElementById("cron-timeline"), bandInfos);
     Timeline.loadXML("output.xml", function(xml, url) { eventSource.loadXML(xml, url); });
    }

    // boilerplate code as specified in the tutorial
    var resizeTimerID = null;
    function onResize() {
     if (resizeTimerID == null) {
            resizeTimerID = window.setTimeout(function() {
            resizeTimerID = null;
            tl.layout();
       }, 500);
     }
    }
    </script>
    <title>My Test Cron Timeline</title>
  </head>
```

```
  <!-- run our custom code upon page load/resize -->
  <body onload="onLoad();" onresize="onResize();">

      <!-- actually display the timeline here in the document -->
      <div id="cron-timeline"
             style="height: 150px;
             border: 1px solid #aaa">
      </div>

  </body>
</html>
```

To avoid repeating the explanation for each part of this file as it is described in the Timeline tutorial, let me just refer you to that Web page instead.

The one last non-Perl thing I need to show you to complete this subtask is an example of the event data we'll need (in a file called output.xml). This will give you an idea of which data the widget is expecting us to provide. Here's an example that assumes we're showing the cron events for January 2008:

```
<data>
 <event start="Jan 01 2008 00:45:00 EST" title="/priv/adm/cron/hourly"></event>
 <event start="Jan 01 2008 01:45:00 EST" title="/priv/adm/cron/hourly"></event>
 <event start="Jan 01 2008 02:45:00 EST" title="/priv/adm/cron/hourly"></event>
 <event start="Jan 01 2008 03:45:00 EST" title="/priv/adm/cron/hourly"></event>
 ...
 <event start="Jan 01 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
 <event start="Jan 02 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
 <event start="Jan 03 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
 <event start="Jan 04 2008 03:15:00 EST" title="/priv/adm/cron/daily"></event>
 ...
 <event start="Jan 06 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
 <event start="Jan 13 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
 <event start="Jan 20 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
 <event start="Jan 27 2008 05:15:00 EST" title="/priv/adm/cron/weekly"></event>
 <event start="Jan 01 2008 06:15:00 EST" title="/priv/adm/cron/monthly"></event>
</data>
```

Hmm, writing an XML data file: how do we do that? Read on.

XML Output with No Effort

So far we've vanquished the tricky parts of the project having to do with determining which data we need and what will consume this data. The last part is to make sure we format the data in a form that will work. In this case we're looking to create an XML file with specific tags and contents. There are a whole bunch of Perl ways to generate XML files, ranging from simple print statements to fairly complicated event-driven frameworks. The one that probably best serves our rather meager needs for this project is the use of the module XML::Writer. It makes it easy to produce XML that has properly matched tags, each with the correct attributes. This mostly requires code something like this:

```
# set up a place to put the output
my $output = new IO::File(">output.xml");

# create a new XML::Writer object with some pretty-printing turned on
my $writer
   = new XML::Writer( OUTPUT => $output, DATA_MODE => 1, DATA_INDENT => 2 );
```

```
# create a <sometag> start tag with the given attributes
$writer->startTag('sometag', Attribute1 => value, Attribute2 => value );

# just FYI: we could leave out the tag name here and it will try to
# figure out which one to close for us
$writer->endTag('sometag');

$writer->end();
$output->close();
```

Putting It All Together

Congrats: we've now seen all of major pieces and we're ready to show the "final" code. I'll only explicate the pieces of the code that are new to the discussion.

PART ONE: LOAD THE MODULES

```
use strict;
use Schedule::Cron::Events;
use File::Slurp qw( slurp );        # we'll read the crontab file with this
use Time::Local;                    # needed for date format conversion
use POSIX;                          # needed for date formatting
use XML::Writer;
use IO::File;
```

PART TWO: SET US UP CHRONOLOGICALLY

We're going to have to tell Schedule::Cron::Events where to begin its event iteration. Basically, we have to pick a start date. It seems as though it might be useful to display a timeline showing the events for the current month, so let's calculate the seconds from the epoch at the beginning of the first day of the current month:

```
my $currentmonth = ( localtime( time() ) )[4];
my $currentyear  = ( localtime( time() ) )[5];
my $monthstart   = timelocal( 0, 0, 0, 1, $currentmonth, $currentyear );
```

PART THREE: READ THE CRONTAB FILE INTO MEMORY

```
my @cronlines = slurp('crontab');
chomp(@cronlines);
```

PART FOUR: CREATE AND START THE XML OUTPUT FILE

```
my $output = new IO::File(">output.xml");
my $writer
   = new XML::Writer( OUTPUT       => $output, DATA_MODE   => 1,
                      DATA_INDENT => 2 );

$writer->startTag('data');
```

PART FIVE: LA MACHINE (THE ACTUAL WORK)

We've now hit the place in the code where the actual iterating over the contents of the crontab file takes place. As we iterate, we need to enumerate all of the events produced by each line we find. Because Schedule::Cron::Events is happy to provide `nextEvent()`s ad infinitum, we'll have to pick an arbi-

trary time to stop. As mentioned before, showing a month seems like a good timespan, so our code stops asking for a `nextEvent()` as soon as that call returns something not in the current month.

Let's look at this iteration:

```
foreach my $cronline (@cronlines) {
    next if $cronline =~ /^#/;
    my $event
      = new Schedule::Cron::Events( $cronline, Seconds => $monthstart );
```

For each line in the crontab that is not a comment, we hand that line off to Schedule::Cron::Events with a start time of the beginning of the current month.

Then we iterate for as long as we're still in the current month:

```
while (1) {
    @nextevent = $event->nextEvent;

    # stop if we're no longer in the current month
    last if $nextevent[4] != $currentmonth;
```

For each event, we're going to want to generate an `<event>` element with the start attribute showing the time of that event and the title attribute listing the command cron would run at that time. We'll be calling the `strftime()` function from the POSIX module to get the date formatted the way the Timeline widget likes it:

```
 $writer->startTag('event',
      'start' => POSIX::strftime('%b %d %Y %T %Z',@nextevent),
      'title'  => $event->commandLine(),
  );
  $writer->endTag('event');
```

We could add an `end` attribute to this element if we knew how long each event would last. Unfortunately, there is no easy way to know or estimate the length of time a particular cron job takes (as suggested in the email that started this column). However, you could imagine writing more code to analyze past crontab logs to try to guess that information. Yes, this is one of those dreaded "This exercise is left to the reader" moments.

That's basically it. We now just need to close the Perl loops, close the outer tag in the XML file, stop XML::Writer's processing, close the file itself, and we're done:

```
    }
}

$writer->endTag('data');
$writer->end();
$output->close();
```

So, how's this look? Figure 2 shows a screenshot from the widget when loaded into a browser using our newly created data file.

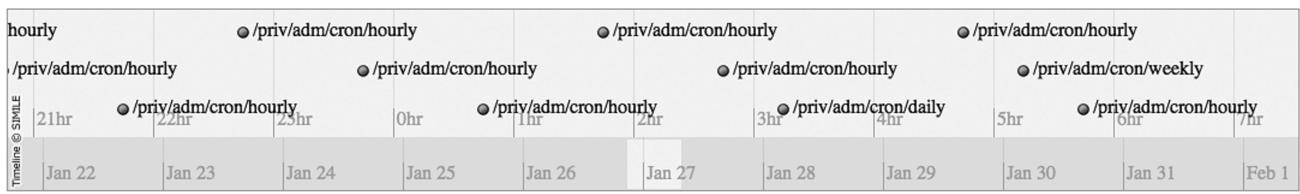

FIGURE 2: TIMELINE FROM A SIMPLE CRONTAB

Trust me, it's cooler in person, because you can scroll back and forth in the month.

I realize that this code doesn't fulfill the original correspondent's wishes because, number 1, it's not a GANTT chart (which would require analyzing the different cron jobs and seeing how they connect) and, number 2, it doesn't show multiple machines overlaid.

Defect number 1 turns out to be pretty hard to remedy. As Richard Chycoski pointed out in a follow-up to this message, dependency tracking in this context gets you into the fairly complex "batch processing" world, something we can't address in this column. Luckily, defect number 2 is pretty easy to fix; it just requires opening more than one crontab file and doing the same work on each file. That's actually a reasonable exercise for the reader with which to leave you without feeling guilty, so have at it.

Even with these defects the diagram seemed pretty spiffy to me. I wanted to see what would happen if I fed the script real-world data from another site. I contacted John, the writer of my opening email message, and he was kind enough to send me a set of crontabs including one that he described as follows: "These jobs are in use at Metro, producing space utilization reports for our NetApp, driving the cold backup sequence for Oracle databases, and other system tasks." Running my code against this crontab file (and changing the HTML file that displays it so it has a larger display area) yields the results in Figure 3, which John describes as "Sweet!"

FIGURE 3: TIMELINE FROM A REAL-WORLD CRONTAB

Hopefully this fun little example has given you some tools both for working with crontabs and for creating timelines. I'm certain there are some more interesting offshoots of this idea just waiting for you to find them. Take care, and I'll see you next time.