Practical Perl Tools

Give as Good as You Get, My Tiny Dancer

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

During our last time together, we had a chance to explore some of the features of the Web library for Perl, the seminal HTTP client distribution (more commonly called LWP). We saw how to fetch HTTP content from Web servers, POST data to them, and so on. I thought it might be interesting to look at the other side of the coin and explore another way to construct Perl applications that serve data to HTTP clients like those from my April column. I say "another" way because all the rabid fans of this column (I'm waving to both of you!) will recall our forays into the CGI::Application framework back in July and September of 2009. CGI::Application is still alive and kicking, but since then there have been a number of new frameworks released that some considered to be the new hotness. In this column we'll look at one of those frameworks, and, if polling numbers stay high, we'll look at a "competing" framework in the next issue.

NEWSFLASH: Before we get to that framework, a quick newsflash related to last issue's column. After the column was submitted (but perhaps before it was printed), a new 6.0 version of the LWP distribution was released. It was largely a revision related to which modules were bundled, but there was one change that is likely to make a difference to readers of this column. I'm indebted to David Golden who quoted one of the new parts of the LWP::UserAgent documentation in his blog:

> If hostname verification is requested, and neither SSL_ca_file nor SSL_ca_path is set, then SSL_ca_file is implied to be the one provided by Mozilla::CA. If the Mozilla::CA module isn't available SSL requests will fail. Either install this module, set up an alternative SSL_ca_file or disable hostname verification.

Short translation: you probably want to install the Mozilla::CA module if you plan to make https requests using LWP. You could set PERL_LWP_SSL_VERIFY_HOSTNAME to 0 to disable hostname verification, but that would be considerably less secure. Just a quick heads-up that hopefully will save you a bit of a surprise when you upgrade LWP. Okay, onward to the main attraction.

Dancer

One of the things I appreciate about CGI::Application is its (initially) simple model of the world. In CGI::Application, each page is associated with something it calls a run mode. Each run mode could consist of a subroutine whose job it was to produce the output for that run mode. Simple Web applications are indeed that simple, although as things get more complex in a CGI::Application application, so does the code and its control and data flow. For this column, let's look at another framework

that aims for simplicity. Dancer started out as a straight port of a deservedly much praised Ruby Web application framework. No, not that one. Dancer is a port of Sinatra, a Ruby framework considerably simpler than Rails.

How simple? Although this example from the documentation tends to come up in any discussion of Dancer, you'll forgive me if I feel compelled to repeat it here as well:

```
use Dancer;

get '/hello/:name' => sub {
    return "Why, hello there " . params->{name};
};

dance;
```

Let's take this example apart in excruciating detail, because Dancer's simplicity is directly related to its concision. The first line is easy: we're just loading the module. The next lines define what Dancer calls a "route." Routes are specifications for incoming requests and how to handle them. They consist of the kind of request (in this case a GET), the path being requested (i.e., the part of the URL after the server name), and what to do with the request (i.e., what code to run when it comes in). The last two parts of the route definition are the more complicated aspects of the route lines above, so let's go into further detail.

In the example above, the specification says to look for incoming requests that match "/hello/:name". The first part, the part between the slashes, looks reasonable, but what's that ":name" part? Anything that begins with a colon is meant to indicate a placeholder accessed by that name (the doc calls it a named-pattern match). In this example, it means our code is looking for a request of the form:

```
http://server/hello/{something}
```

When it finds it, it places the component of the path represented by {something} into a parameter called "name". The code that gets run as part of this route looks up the "name" parameter in the params hash reference Dancer makes available to all routes and returns it as part of the subroutine's return value.

This is just one kind of pattern you can specify for a route. Dancer also makes full regular expression matches, wildcards, and conditional matches available in its specification. It also lets you write:

```
prefix '/inventory';
```

before each route handler, and that handler will add that prefix before the specified pattern. So "/shoes/:size" matches as if you specified "/inventory/shoes/:size" when that prefix is set. Dancer also lets you define a default route if desired.

If a pattern matches, it runs the code associated with the route. That code is responsible for providing a response to the incoming request. The example above is the simplest kind of response (essentially just a scalar value); we're going to get more sophisticated in just a few moments. Dancer provides hooks should you want to operate on a request before it gets processed or after the response has been generated. The documentation gives an example of code you could run in a hook to handle a request in a different way, depending on the logged-in status of the user making the request.

The final line in the example code ("dance;") may be the most poetic, but that just spins up the framework after all of the definitions are in place and starts accepting requests. If the Bob Fosse–like ending to your scripts doesn't really work for you (or if your boss is going to read your code and has a distinctly prosaic heart), you can use the more humdrum command "start;" instead.

So how does this thing work? First you need Dancer and its prerequisites installed on your system. Ordinarily, I wouldn't mention such an obvious detail, but I'm taken with the package's intro page, which points out that you can either install it on a UNIX system using the standard CPAN.pm/CPANPLUS invocations or use the very cool cpanminus script like so:

```
# wget -O - http://cpanmin.us | sudo perl - Dancer
```

(If you leave off the sudo invocation, cpanminus will install the whole kit and caboodle into a perl5 directory in your home directory.) I'm sure cpanminus will make another appearance in this column in the future.

As a quick aside, on an OS X machine using a version of Perl from MacPorts, I needed to set `PERL_CPANM_OPT='--local-lib=/Users/dnb/perl5'` in my environment and add `--no-check-certificate` to the wget command line to allow the non-sudo version of that command line to work. Once Dancer was installed, I could add `use local::lib;` at the beginning of my code to use the stuff installed in ~/perl5.

With Dancer on your system, congratulations, you now have a Web application complete with its own Web server, simply by running the script:

```
perl yourscript.pl
```

This will spin up a server listening on port 3000 on the local machine:

```
$ perl dancer1.pl
>> Dancer 1.3020 server 88541 listening on http://0.0.0.0:3000
== Entering the development dance floor ...
```

(and in another window)

```
$ curl http://localhost:3000/hello/Rik
Why, hello there Rik
```

This mini Web server is meant to be just for development. This would not be the way you'd actually want to deploy your Web app in production. The Dancer::Deployment documentation goes into the more robust methods for that (CGI/fast-cgi, Plack, behind a proxy, etc.).

Web Apps That Produce Output Are Generally More Interesting

In our exploration of CGI::Application, we took our time getting to the notion that one would want to use some sort of templating mechanism when writing Web applications. These mechanisms allow you to write static HTML into which dynamic content is inserted at runtime. This lets you create output that hews to a standard look and feel (e.g., same headers/footers/CSS styles) and is much easier than writing a bunch of print() statements yourself. For this intro, let's not pussyfoot around and instead dive right into using what Dancer has to offer in this regard.

Dancer lets you easily pick from at least two templating engines: its rather basic built-in engine and the pervasive engine for most Perl Web frameworks, Template

Toolkit. The Dancer engine (Dancer::Template::Simple) only does simple replacements. If you write:

```
<% variable %>
```

it will replace that string with the value of `variable`. If you decide to go with the other engine, there's a whole book on Template Toolkit and tons of online documentation if you'd like to explore the full range of template power at your disposal.

To use either engine, you place the template in a subdirectory of your application directory (more on directory structure later) called "views," with a name that ends in .tt. This template gets referenced in the subroutine you defined in your route specification, like so (as seen in another example from the documentation):

```
get '/hello/:name' => sub {
    my $name = params->{name};
    template 'hello.tt', { name => $name };
};
```

In this sample, we generate output by processing the hello.tt template. We pass in a parameter hash that replaces the variable "name" in that template with the value we retrieved as part of the path when the request comes in. If views/hello.tt consisted of:

```
<p>Why, hello there <% name %>!</p>
```

we'd get the same output from our `curl` command as before except that it would have paragraph tags around it.

Dancer provides, for lack of a better word, a meta-version of the view templating functionality called "layouts." Layouts let you create a single view that "wraps" or interpolates other views in order to produce the look-and-feel consistency we mentioned at the start of this section. You might create a layout that specifies the header and footer information for every page on your site. Such a layout would look like this:

```
(header stuff here)
<% content %>
(footer stuff here)
```

The content variable above is magic. When any template command gets processed, the layout view is returned with the results of the processed view inserted right at that spot. This allows you to avoid having all of that header and footer stuff copied into every view for your site. One way you could imagine using layouts would be to have a separate view template for every section of your Web site, each of which is inserted into the layout for the site before being sent to the browser.

Directory Assistance

In the previous section I mentioned that, by default, Dancer expects to see its templates in a views directory rooted off of the main application directory. There are a number of other defaults and expectations. Perhaps the easiest way to get them all out on the table, even though we won't have space to explore them all, is to see what happens when we use the helper script called "dancer" that ships with the distribution. Like many other Web frameworks, Dancer provides this script so that you can type one command and have an entire hierarchy of stub files for all of the different

files you may need to construct an application in one swell foop. Here's Dancer's take on that process:

```
$ dancer -a usenixapp
+ usenixapp
+ usenixapp/bin
+ usenixapp/bin/app.pl
+ usenixapp/config.yml
+ usenixapp/environments
+ usenixapp/environments/development.yml
+ usenixapp/environments/production.yml
+ usenixapp/views
+ usenixapp/views/index.tt
+ usenixapp/views/layouts
+ usenixapp/views/layouts/main.tt
+ usenixapp/lib
  usenixapp/lib/
+ usenixapp/lib/usenixapp.pm
+ usenixapp/public
+ usenixapp/public/css
+ usenixapp/public/css/style.css
+ usenixapp/public/css/error.css
+ usenixapp/public/images
+ usenixapp/public/500.html
+ usenixapp/public/404.html
+ usenixapp/public/dispatch.fcgi
+ usenixapp/public/dispatch.cgi
+ usenixapp/public/javascripts
+ usenixapp/public/javascripts/jquery.js
+ usenixapp/Makefile.PL
+ usenixapp/t
+ usenixapp/t/002_index_route.t
+ usenixapp/t/001_base.t
```

Yowsa that's a lot of files. Let's see if we can break this apart so it becomes a little less scary.

```
+ usenixapp/bin/app.pl
+ usenixapp/lib/usenixapp.pm
```

is essentially where you'd find the script we've been writing so far. The former imports the latter like a module and starts Dancer.

```
+ usenixapp/views
+ usenixapp/views/index.tt
+ usenixapp/views/layouts
+ usenixapp/views/layouts/main.tt
```

Here is where the views we've already talked about are stored.

```
+ usenixapp/public/*
```

Dancer places all of the static files into a public directory. This include CSS, HTML error pages, jQuery library, and so on.

True to the Perl tradition, Dancer wants you to write tests for your code:

```
+ usenixapp/t/*
```

Yay, Perl (and Dancer).

This just leaves a few mysterious stragglers:

```
+ usenixapp/config.yml
+ usenixapp/environments
+ usenixapp/environments/development.yml
+ usenixapp/environments/production.yml
```

Dancer makes it very easy to construct and read YAML-based configuration files for your application. In these files, you can specify things such as the templating engine choice we discussed before, what level of debugging to use, how logging will take place, how sessions are stored, and so on. Dancer lets you keep a general config file plus separate config files that can be used depending on how you are using your application (is it in development? production?). Dancer can keep configuration information in the code itself, but, as you can see above, it strongly suggests that you keep that sort of information in dedicated files for this purpose.

I Could Dance All Night

Unfortunately, we're almost out of space, so we're going to have to get off the dance floor and just list some of the functionality in Dancer we won't be able to explore. The documentation is good, so hopefully my just mentioning these capabilities will encourage you to read further to see how these things are accomplished in Dancer. Dancer, like the other frameworks, provides a number of ways to keep "sessions" (i.e., some permanent state between requests, such as for the user who is logged into an application). It has useful logging support that allows the application to log debug messages. It can (with the help of a separate plug-in) make AJAX programming easier (routes can return JSON and XML easily). It can cache routes for better performance if you'd like. There's lots of spiffy stuff in this one distribution, so I would encourage you to check it out for at least your smaller Web application needs.

Take care, and I'll see you next time.