COLUMNS

Practical Perl Tools

Got My Mojolicious Working

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

Ah, last issue's column. Who can forget it? It had poise. It had verve. It had rhythm. And it certainly had a better introduction than this one. Lest you have forgotten that bygone issue, let met do a little recap. We looked at a very spiffy Web framework called Dancer. Actually, on the official Dancer Web site at perldancer.org they call it a "micro Web application framework," because it is so simple and concise. Based on Ruby's Sinatra framework, it lets you write a script that looks like this:

```
use Dancer;

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

dance;
```

...and then spin up a tiny Web server that will process requests simply by running the script.

The code looks for incoming GET requests with a path of /hello/{something} and then returns a jovial response of "Why, hello there {something}". The line that begins with "get..." is a route specification. It states that if a request comes in that matches that specification, the associated code should be run. This route-based approach, where the programmer essentially provides a dispatch table, seems to be pretty common in the Web framework world these days. Another pervasive feature for Web frameworks is the ability to generate Web pages through the use of a templating system. We took a brief look at Dancer's support for templates and a helper command-line script that created an entire sample application directory structure which included a place for those templates. At that point, I had to end our time together and leave you with a bit of a cliffhanger by mentioning but never really naming a "competing" Web framework that compared to Dancer. That's what we are here to explore today.

In this column, we're going to look at Mojolicious (http://mojolicio.us, cute, huh?). More precisely, we're going to mostly consider Mojolicious::Lite, the "micro Web framework" part of the larger package, because it is closer in nature to Dancer. Let's look at our first piece of Mojolicious::Lite code:

```
use Mojolicious::Lite;

get '/hello/:name' => sub {
    my $self = shift;
    $self->render(text => "Why, hello there " . $self->param('name'));
};

app->start;
```

Why yes, it does look remarkably like the Dancer example above with a wee bit of object-oriented programming syntax snuck in there. The close resemblance is intentional on my part, but the first bit of code in the Mojolicious::Lite introduction is nearly identical to what I wrote above. There's a bunch more going on here, but if you are comfortable with the Sinatra-like syntax in Dancer, you'll be fine with Mojolicious::Lite as well.

What Is Mojolicious and Where Can I Get One?

One thing I don't want to do with this column is imply that Mojolicious or even Mojolicious::Lite is just Dancer++. It would be possible to write a column that says "Dancer offers this, but Mojolicious::Lite offers that plus this other thing," but I think that would do a disservice to both frameworks. If I did that you wouldn't really get a sense of just how different they are, even just in their basic worldview. So let me step way back for a moment and discuss Mojolicious itself and how it relates to Mojolicious::Lite.

Once upon a time, a German Perl programmer by the name of Sebastian Riedel took over maintenance of a Web application framework called Maypole. Eventually he left that project to found another Web application framework, called Catalyst (Catalyst is probably the most popular of the Perl Web app frameworks). Riedel eventually left the Catalyst project and created yet another framework, called Mojolicious. And that's where our story begins. I have no idea why Riedel is such a serial framework creator, I just know he tends to start and leave good stuff behind in his wake that tends to move the field forward.

One of the first ways Mojolicious distinguishes itself from other frameworks is in its dependencies on other modules outside the Perl core.

It doesn't have any.

"But that's crazy talk!" you say. "What about LWP? How about the parsing of HTTP messages? URL processing? Templates? Cookies? Surely it doesn't implement its own full featured UNIX-optimized preforking async I/O HTTP 1.1 and WebSocket server with IPv6, TLS, Bonjour, epoll, kqueue, and hot deployment support?" That last question is a bit over the top (and if you really said something like that I would probably start edging away slowly), but, yes, Mojolicious does indeed ship with code that implements all of these things.

I'm not entirely sure all of this reimplementation is automatically a good thing. It means that the rest of the Perl world isn't constantly battle-testing Mojolicious's building blocks in the same way other modules do when they rely on common modules such as HTTP parsing libraries. A rejoinder to this concern comes in the Mojolicious FAQ in response to the question, "Why reinvent wheels?":

"Because we can make them rounder."

The same FAQ document also points out that Mojolicious will also optionally use some external Perl modules such as IO::Socket::SSL when it makes sense to "provide advanced functionality."

One plus of this approach is that the functionality found in a Web framework like Mojolicious becomes bounded, not by the restrictions of that framework's external dependencies, but, rather, by the imagination of the framework's author. In Mojolicious's case, it has let the author create a very nice prototyping Web application framework in the Sinatra vein with lots of hidden power under the hood. Mojolicious::Lite is a "micro Web framework built around Mojolicious," according to the documentation. (And in case you were curious, Mojolicious itself is built on top of something called Mojo, which the author calls "a flexible runtime environment for Perl Web frameworks"). Let's focus on some parts of Mojolicious::Lite.

Mojolicious::Lite's Templates

I don't want to rehash the last column's discussion of routes, because they get used in a very similar fashion in Mojolicious::Lite. Once you understand the basic idea, your understanding can be applied in both places without much adaptation. Instead, let's look at a few places where Mojolicious::Lite does things a bit differently, because they will reveal some of the hidden power I mentioned a moment ago.

Templating is one of the areas worth exploring. Mojolicious::Lite has its own template language (although you can use other template engines if you so desire) called Embedded Perl. Templates can either live in files that end with .ep (usually found in a templates directory in your application) or actually embedded in the script itself at the end in a __DATA__ handle. Since most of the introductory documentation demonstrates the latter, let's use that too. Here's a modified version of our last example:

```
use Mojolicious::Lite;

get '/hello/:name' => sub {
    my $self = shift;
    $self->render();
} => 'hello';

app->start;

__DATA__
@@ hello.txt.ep
<%= "Why, hello there $name" %>

@@ hello.html.ep
<html>
<head><title>"Hello"</title></head>
<body>
Why, hello there <strong><%= $name %></strong>
</body>
</html>
```

The first thing you'll notice is that I've added two templates, hello.txt.ep and hello.html.ep, to the __DATA__ section of the script. Since you don't see it used that often in practice let me mention that the idea of a __DATA__ section is actually a Perl, not a Mojolicious thing. In Perl, if you add __DATA__ at the end of the script, that script can read the lines following that marker as if they were coming from a real filehandle (e.g., with while (<__DATA__>) {something...}). Mojolicious::Lite lets you specify multiple "files" in that section by prefixing each section with @@ and the name of the file. It should be mentioned that using the __DATA__ section is just a shortcut. If we wanted to create two separate files and place them in the right templates directory for an application, they would function exactly the same way.

In the templates themselves, you see the use of the <%= Perl_expression %> tag. The contents of that tag are replaced by the result of the enclosed expression after Perl has evaluated it. Here are the other possibilities from the Mojo::Template documentation:

```
<% Inline Perl %>
<%= Perl expression, replaced with result %>
<%== Perl expression, replaced with XML escaped result %>
<%# Comment, useful for debugging %>
% Perl line
%= Perl expression line, replaced with result
%== Perl expression line, replaced with XML escaped result
%# Comment line, useful for debugging
```

Using either the "<" or "%" convention, you can basically embed whatever Perl code you would like into the template (hence the name). The code above just substitutes in the value of $name, but it could just as easily have been a much more complex Perl expression.

While we're talking about the $name variable, I should explain how that variable gets set, since there are a few things going on behind the scenes. In the route specification above, we provided the named placeholder using the syntax ":name". When an incoming request matches that route, the part of the request that matched the placeholder automatically gets extracted and stored in something called "the stash" under that name. The stash is a temporary holding spot for stuff like template values. When the template gets rendered, it looks in the stash for these values.

Data can also be put in the stash by hand; for example, we can also write code such as:

```
$self->stash('editor' => 'rik');
```

and $editor would be available to render in a template.

There are two additional changes between the two previous code samples that reveal more interesting default "smartiness" in Mojolicious::Lite. The first is the addition of the following to the routing specification:

```
} => 'hello';
```

This gave the route a name. Mojolicious::Lite will use that name when deciding what template to render. That's why we didn't have to specify a template in this line:

```
$self->render();
```

When I removed the arguments to the render() call, I brought two Mojolicious::Lite defaults into play. The first is the use of the route name to select the appropriate template basename (the name without the format suffix). The second is an automatic detection of format (i.e., .html or .txt). If I use the following URL in the browser:

```
http://127.0.0.1:3000/hello/dnb.txt
```

the hello.txt.ep template is rendered for me. Similarly, if I use:

```
http://127.0.0.1:3000/hello/dnb.html
```

the hello.html.ep template is chosen.

Mojolicious::Lite's Filigrees

This idea of "guess what I might need to do and make it easy" pervades the whole package. Here's an example of using sessions:

```
use Mojolicious::Lite;

get '/login/:username' => sub {
   my $self = shift;
   $self->session( username => $self->param('username') );
   $self->render();
} => 'login';

get '/hello' => sub {
   my $self = shift;
   my $username = $self->session('username') || '(no one)';
   $self->stash( username => $username );
   $self->render();
} => 'hello';

get '/logout' => sub {
   my $self = shift;
   my $username = $self->session('username') || '(no one)';
   $self->stash( username => $username );
   $self->session( expires => 1 );
   $self->render();
} => 'logout';

app->secret("shhh, I'm hunting wabbits");
app->start;

__DATA__
@@ login.html.ep
Ok, <%= $username %> is logged in.

@@ hello.html.ep
Welcome back  <%= $username %> !

@@ logout.html.ep
<%= $username %> has been logged out.
```

The only really new concept in this example can be found in the use of the session() method calls. Mojolicious::Lite provides built-in session management to help deal with the perpetual question of how to maintain state (e.g., someone's logged-in status) across a set of stateless HTTP requests. Mojolicious::Lite does all the work for you, including generating, exchanging, and expiring HMAC-MD5 signed session cookies. The app->secret() call above simply sets the secret used to sign the cookies. It is a good idea to set your own secret like this so the default secret (the name of the application) is not used.

Here's another piece of code right from the documentation:

```
use Mojolicious::Lite;

any '/upload' => sub {
    my $self = shift;
    if (my $example = $self->req->upload('example')) {
        my $size = $example->size;
        my $name = $example->filename;
        $self->render(text => "Thanks for uploading $size byte file $name.");
    }
};

app->start;
__DATA__

@@ upload.html.ep
<!doctype html><html>
    <head><title>Upload</title></head>
    <body>
        <%= form_for upload =>
                (method => 'post', enctype => 'multipart/form-data') => begin %>
            <%= file_field 'example' %>
            <%= submit_button 'Upload' %>
        <% end %>
    </body>
</html>
```

This code shows how easy it is to perform a file upload using Mojolicious::Lite. In the template, you can see some built-in tag helpers such as "form_for", "file_field", and "submit_button" that make creating a form easier by generating the right HTML. What you can't see from just this code is that Mojolicious::Lite will (1) prevent the user from uploading a file that is larger than some limit you set, and (2) write the incoming data to a temporary file (for files over 250k) during the upload rather than trying to store all the data in memory.

Here are three more cool features that don't really have specific code associated with them:

1. To create tests for your application (you are creating tests, right?), you can create a "t" directory and place your usual Perl tests there. Mojolicious provides a number of Web application testing helpers such as get_ok() (to fetch a Web page and compare the result) and status_is() (to test the response code). It also provides easy ways to parse an existing HTML document (more on this later) that can be used for a more precise test code.
2. If you want to capture a detailed log of your prototype application, it is as simple as creating a "log" directory. Mojolicious::Lite will start to write a log file into that directory with lines such as:

```
Mon May 30 16:51:48 2011 info Mojo::Server::Daemon:297 [20304]: Server
listening (http://*:3000)
Mon May 30 16:52:10 2011 debug Mojolicious::Plugin::RequestTimer:22 [20304]:
GET /login/dnb (Mozilla/5.0 (Macintosh; U;
        Intel Mac OS X 10_6_7; en-us) AppleWebKit/533.21.1 (KHTML, like Gecko)
         Version/5.0.5 Safari/533.21.1).
Mon May 30 16:52:10 2011 debug Mojolicious::Routes:376 [20304]: Dispatching
callback.
Mon May 30 16:52:10 2011 debug Mojolicious::Plugin::EplRenderer:57 [20304]:
Rendering template "login.html.ep" from DATA section.
Mon May 30 16:52:10 2011 debug Mojolicious::Plugin::RequestTimer:44 [20304]:
200 OK (0.004464s, 224.014/s).
```

3. And one more feature that seems obvious once you hear it is possible:

When you start up a prototype application in daemon mode (the one that spins up a test Web server), it can be started with a `--reload` flag, like so:

```
$ perl testapp.pl daemon --reload
```

When started this way, Mojolicious::Lite's test Web server will monitor your testapp.pl file for modification and reload itself with the new contents of that file if it spots any changes. With this approach, making a change, restarting the Web server, making another change, restarting the server, and so on becomes unnecessary and the interaction is much more pleasant. Using the Web server in this mode has some limitations (see the documentation), but most of the time it works great.

What Else?

Once again we are at a place where we've opened the door to peek at a subject but, for space reasons, don't have a chance to fully explore the magic land beyond that door. With Mojolicious this is especially true, because we've only scratched the basic prototyping layer it provides (Mojolicious::Lite). Mojolicious itself is a full-on Web framework that lets you build a full-bore MVC Webapp. You can split up the logic for your application into appropriately sized compartments (each in its own class/package/module). More advanced route handling can be specified to direct the program flow to specific handlers based on conditions (e.g., which browser is being used, invalid input), to ignore parts of the request URL, and so on. There are too many other lovely features to this package (including a cool HTML parser called Mojo::Dom which can use CSS-like selectors to return information) to cover in just one article. Oh, and I haven't mentioned all of the plugins available which make using back-ends like Redis, MongoDB, and Memcached from Mojolicious pretty painless. Set aside some time to look at the documentation at http://mojolicio.us. I think you'll be pleased if you do.

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