DAVID N. BLANK-EDELMAN

practical Perl tools: scratch the Webapp itch with CGI::Application, part 1

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.

dnb@ccs.neu.edu

EVER NEED TO WRITE A SIMPLE WEB application but didn't know the current easy way to go about it (in the Perl world)? Me too. I recently had to write something that would query a small database I prepopulated with user data, present the info we had to that user for confirmation, and then initiate a data migration process on their behalf.

I knew that I wanted to present the information in bite-sized chunks to the users so it would be easy for them to walk through the process. That meant the application would have to present several Web pages in a row as the users completed each part of a multi-page sequence. To show the users such a set of connected pages meant my application would (ideally) track sessions in order to retain a user's information from one HTTP POST to another. Writing all of that low-level plumbing is certainly possible in Perl, especially with the help of some modules, but by this point it was clear I should be hunting for somebody's "been there, done that" Web application framework to make my life easier.

I haven't paid as much attention as perhaps I should have to this corner of the Perl world, so I started looking at some of the usual suspects. The first stop was Catalyst (http://www.catalystframework.org/), one of those Model-View-Controller thingees that Ruby on Rails has made so popular. But the more I looked at Catalyst, the more it seemed to be overkill for the job. I didn't need to write something that was particularly database-driven. My app would only make one small query to its database at the beginning; it didn't really need an object-relational mapper (ORM) to help make database querying/manipulation easier. Catalyst looked great but it had way too much firepower (and a bit of commensurate learning curve) for this particular task. I had a similar reaction to Mojo and Mojolicious (http://mojolicious.org/).

Then I bumped into CGI::Application (http://www.cgi-app.org/), which is (as of this writing) working its way toward becoming a package called Titanium. CGI::Application was (to use a phrase often misattributed to Einstein) as simple as possible but no simpler. It offered a mental model that was immediately easy to grok without having to pay attention to three-letter acronyms like MVC and ORM. It had a bunch of plugins to handle the tricky parts of the plumbing (such as session handling, lazyloading of database handles, and data validation). CGI::Application was the perfect fit for the meager needs of my small Webapp.

In this two-part column I'm going to take you through the basics of CGI::Application in the hope that it may prove to be a good fit for your needs too. As part of this I'll be using a few of the plugins that are considered best practices these days (and hence are bundled with Titanium). We'll be sticking to largely just the ground floor of Webapp programming here; I'd recommend going to http://www.cgi-app.org/ for the fancier stuff. One last disclaimer: CGI::Application is object-oriented in nature, but you don't have to be object-oriented to make use of it. The OOP stuff in the column won't get much fancier than method calls. Feel free to treat anything you don't understand as an incantation that can be used without full knowledge of how the OOP works.

Defining Run Modes

With all of that out of the way, let's talk about the main idea that underlies CGI::Application. If you get this, you'll have little to no trouble using the framework. CGI::Application applications (sigh, let's just call them cgiapps for short) are composed of a number of run modes. The easiest way to think of them is that every Web page in your cgiapp has its own run mode. Have a page of instructions to display? That's a run mode (maybe we'll call it "display_instructions"). Have another page that collects the user's personal info? That's another run mode (perhaps "get_personal"). And so on.

Each run mode has code associated with it, in the form of at least one subroutine. That subroutine gets called when the cgiapp enters that run mode, and it is responsible for producing the contents of the run mode's Web page. In case you are curious, I say "at least one subroutine" just because the subroutine that gets called for a run mode might have other support routines you've written to help it out. For example, the run mode subroutine `get_personal()` might call `query_personal_database()` to get values that will be displayed by this run mode.

The set of run mode subroutines for an application gets collected in an "application module," which is just a regular ol' Perl module (i.e., usually named with a .pm suffix, ends with "1;", etc.). The module should define a subclass of CGI::Application. As sophisticated as that sounds, it just means you will start the file with:

```
package ColumnDemoApp;    # or whatever you want to call your application
use base 'CGI::Application';
```

Toward the end of this column, I'll show you how this application module actually gets used. Before we get there, let's figure out exactly what it contains. After the two OOP mumbo-jumbo lines above, we'll find the definitions of the subroutines that will be used for each run mode and the code that tells CGI::Application which run mode is associated with each subroutine. Once upon a time, this association was provided using a special `setup()` subroutine. The current best practice is instead to use a helper plugin called CGI::Application::Plugin::AutoRunMode:

```
use CGI::Application::Plugin::AutoRunMode;
```

C::A::P::AutoRunMode (sorry, from this point on in the column I'm going to start abbreviating the CGI::Application and CGI::Application::Plugin names to save my aging fingers) provides a convenient shortcut syntax that allows you to associate run modes with subroutines right at the point where the subroutines are defined. For example:

```
sub display_instructions   :   StartRunMode { <code here> };
sub get_personal           :   RunMode { <code here> };
sub engage_warp_drive      :   RunMode { <code here> };
```

At this point we will have defined three run modes (display_instructions, get_personal and engage_warp_drive) and the code that will be executed for each. The first is designated as the "start mode," which just means it is the first run mode a cgiapp enters before any other run mode is explicitly entered. We'll talk in the very next section about how one moves from run mode to run mode.

What Must Leave a Run Mode Subroutine

As I mentioned before, each run mode subroutine is responsible for providing the content for the Web page. It needs to return this information as a scalar like any other scalar value returned from a subroutine, i.e.:

```
return $page;
```

Note that I say `return` and not `print` the output. CGI::Application will handle getting the contents of that returned value to the Web server. Explicitly printing the page output (or any other output) to STDOUT is a big no-no. That being said, your program is responsible for making sure the contents of the page is a valid HTML document complete with <html> and <body> tags, i.e., the usual. There are at least a couple of ways to make creating this output easier, and we'll look at one of them in just a moment.

In general you can put anything you want into this valid HTML, but there is one requirement for all the Web pages in your application that lead to other Web pages. Each Web page must define an HTML form of some sort that defines a mode parameter. The mode parameter contains the name of the *next* run mode the application will move into once the form is submitted. If you think about any multi-page Web application you've used recently, it had some sort of "next" or "submit" button to take you to the next page. You'll need to include something similar in your HTML code that sets the mode parameter. By default the mode parameter is `rm` (for run mode).

To make this more concrete, here's a sample HTML form definition we could have as part of the HTML returned by display_instructions() to switch the user to the get_personal run mode:

```
<form method="post" action="http://server/columndemo.cgi">
  <input type="hidden" name="rm" value="get_personal" />
  <input type="submit" name="Continue" value="Continue" />
</form>
```

This is the answer to the question, "How do you go from one run mode to another?" To do so, the current run mode provides a form with an `rm` (or equivalent—you get to change the default if you need to) form parameter set. When that form gets `POST`ed, CGI::Application reads the mode parameter and enters the indicated run mode.

CGI::Application also has a couple of plugins to allow you to change run modes without having to wait for a form to be `POST`ed: C::A::P::Forward and C::A::P::Redirect. The Forward plugin is useful if your application realizes it should be in a different run mode or displaying a different Web page. For example, in the application I wrote, I created special error run modes to handle fatal and non-fatal errors separately. If something fails (e.g., a database lookup), it forwards out of the current mode into the right error run mode. I also forward in those cases where the user's input indicates I can skip past one of the Web pages in a sequence because the information on that page no longer applies. This change of mode happens transparently to the user; they never know the application had decided it belongs in a different run mode.

Sometimes, however, you want the user (or, more precisely, the user's browser) to know it belongs someplace else. That's when the Redirect plugin comes into play. It gets used to hand an HTTP redirect back to the user's browser. This could come in handy if, for instance, the user's session has timed out and you need to punt them back to the initial login page before they can continue. Returning the right headers to the client to make this happen isn't all that hard; this plugin just makes it really easy.

What Enters a Run Mode Subroutine

So far we've only talked about what a run mode should emit. But it gets some support from CGI::Application for its work. Each run mode is fed the current instance object from the application module's class. If that sentence didn't parse for you, don't sweat it, because I can safely rephrase it as "every run mode subroutine gets passed an object containing a bunch of stuff about the active application at that point." For example, you can write:

```
sub get_personal    :  RunMode {
   my $self = shift;

   my $q = $self->query();
   ...
}
```

and $q will have a CGI.pm object (CGI::Application is built on CGI.pm) hot and ready to go for you. This means you could then write:

```
my $formparam = $q->param('lastname');
```

to retrieve the "lastname" parameter that was filled in on the form that got you to this run mode. The CGI.pm HTML construction methods are also ready for your use, so you can write code like:

```
sub display_instructions    :  StartRunMode {
   my $self = shift;

   my $q = $self->query();

   my $page = $q->start_html(-title => 'Test Page');
   $page .= $A_BUNCH_OF_INSTRUCTION_TEXT;
   $page .= $q->start_form();
   $page .= $q->hidden(-name => 'rm', -value => 'get_personal');
   $page .= $q->submit();
   $page .= $q->end_form();
   $page .= $q->end_html();

return $page;
}
```

Earlier in this article I mentioned that there were ways to make the construction of a valid HTML page easier. Using CGI.pm methods like this is one of them.

There are a number of other really useful method calls available from this object beyond query(), especially if you start adding plugins to the mix. We've already mentioned C::A::P::Forward and C::A::P::Redirect, which provide (you guessed it) $self->forward() and $self->redirect(). Other plugins make it easy to pass around DBI database handles, Log::Dispatch dispatcher objects, and so on. We'll get to that stuff in part two of this column. I'll also show you the second method for easy page construction in the second part.

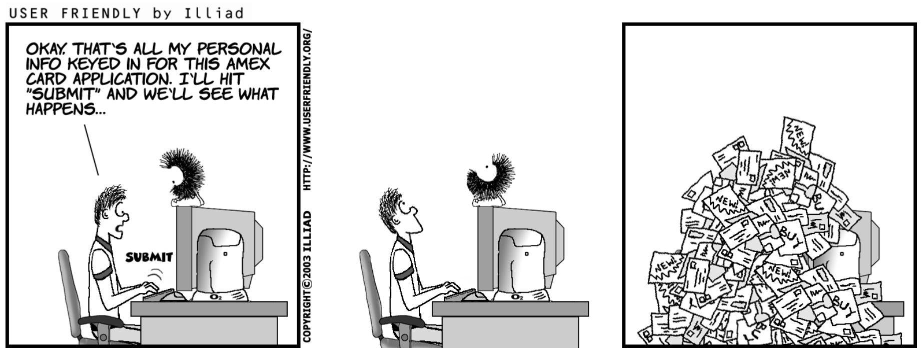
The Instance Script

You've probably guessed that the mention of the second part means we're approaching the end of this one. Before we part ways, it is pretty important that I show you how all of your hard work in writing run mode subroutines actually gets used. Here's the last piece of the puzzle that is necessary for actually constructing a running cgiapp. The script that gets called by users (i.e., that they point their browser at) is called an instance script. It has this name because its whole job is to load your application module, create an instance of the object it defines, and then run that object. In code, this looks like a file with a name like "columndemo.cgi" containing just these four lines:

```
#!/usr/bin/perl

use ColumnDemoApp;
my $Webapp = ColumnDemoApp->new();
$Webapp->run();
```

If we place this file on a Web server that knows how to deal with Perl-based CGI scripts (and has the CGI::Application modules installed), we should be able to go to http://server/columndemo.cgi in a browser and receive the output from our display_instructions run mode code. In the second installment of this column, we'll see some more advanced capabilities of CGI::Application and flesh out a simple Web application using them. In the meantime, take care, and I'll see you next time.


USER FRIENDLY by Illiad
OKAY. THAT'S ALL MY PERSONAL INFO KEYED IN FOR THIS AMEX CARD APPLICATION. I'LL HIT "SUBMIT" AND WE'LL SEE WHAT HAPPENS...
SUBMIT
COPYRIGHT©2003 ILLIAD HTTP://WWW.USERFRIENDLY.ORG/
NEW!