DAVID N. BLANK-EDELMAN

practical Perl tools: scratch the webapp itch with CGI::Application, part 2

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

LAST TIME, WE HAD THE PLEASURE OF exploring the basics of a Web application framework called CGI::Application (CGI::App). I appreciate this particular package because it hits a certain sweet spot in terms of its complexity/learning curve and power. In this column we'll finish up our exploration by looking at a few of the more powerful extensions to the framework that can really give it some oomph.

Quick Review

Let's do a really quick review of how CGI::App works, so that we can build on what we've learned so far. CGI::App is predicated on the notion of "run modes." When you are first starting out, it is easiest to map "run mode" to "Web page" in your head. You write a piece of code (i.e., a subroutine) that will be responsible for producing the HTML for that page and returning it as a scalar value.

To switch from one run mode to another using the classic CGI::App method, the first run mode produces an HTML page containing a form or a URL to follow. This form or link provides the name of the destination run mode in a hidden field (for POSTs) or as a parameter (for GETs). CGI::App looks at the incoming request and determines which run mode to call based on that info. If this sounds too cumbersome or too Web 1.0-ish, you can improve on this method by using the module CGI::Application::Plugin::Dispatch. We won't see an example here, but C::A::P::Dispatch lets you encode the run mode in the path used to call the script using clean, REST-y URLs.

Constructing a Web application using CGI::App consists of:

1. Writing a bunch of run mode subroutines to generate Web pages.
2. Placing them in a file to be loaded as a Perl module (with a little OOP pixie dust sprinkled on top).
3. Writing a separate three-line instance script that loads this module and sets things in motion (this is actually the script that gets called by the Web server directly).

If the CGI::App basics seem easy to grasp, then the mental model CGI::App presents is working for you. Other Perl frameworks like Catalyst seem, at least at first glance, to be more complex (although Dave Rolsky's excellent blog post at http://blog.urth.org/2009/07/what-is-catalyst-really.html

shows this is mostly a PR problem), so perhaps something with this level of simplicity will appeal to you. If it does, then read on, because we're going to show a few features/plugin modules that can pump up the volume while mostly leaving this simplicity intact.

Working with Templates

The first and perhaps the most significant feature is the support for templating available in vanilla CGI::App. If you've ever had to construct a Web site or webapp that involved several pages, you've probably already done at least a basic form of templating, because you needed all of the pages to have some common elements such as headers and footers. Perhaps your code looked something like:

```
$webpage = $header;
$webpage =. generate_the_actual_content();
$webpage =. $footer;
```

Perl has had various template modules of varying complexity available since the Mesozoic era. Most are predicated on the notion that you'll write content that includes some special tokens or markup which is later replaced with real data by the templating engine. For example:

```
<html>
<head><title><TMPL_VAR NAME=PAGETITLE></title></head>
<body>
   <p>Dear <TMPL_VAR NAME=FULLNAME>:</p>
   <p>Get it <a href="<TMPL_VAR NAME=BOOKURL>">here!</a></p>
</body>
</html>
```

The right Perl code, when called with that document, will spit out the document with all of the <TMPL_VAR NAME={something}> mentions replaced with the contents of the named variable. The good news is that CGI::App has this code built in. Out of the box it supports HTML::Template, which uses templates like the one above. It can also use other templating engines (e.g., Template Toolkit), but for this column we'll stick with the built-in support.

To use HTML::Template from within a CGI::App application, you first tell it where to find the templates it will use. This is most commonly done in one of the global initialization subroutines for an application like cgiapp_init:

```
sub cgiapp_init{
   my $self = shift;
   $self->tmpl_path('/path/to/your/templates');
}
```

Once the webapp knows where to find templates, it can load them for processing using the load_tmpl() method. Once loaded, we can replace the <TMPL_VAR> tokens with their values using a set of param() method calls and then produce the final result using output(). Here's an example run mode showing all of these steps:

```
sub welcome_doc : Runmode {
   my $self = shift;

   # die_on_bad_params set to 0 tells HTML::Template to avoid getting huffy
   # if we attempt to replace a parameter that doesn't exist in the template.
   # This will come in handy in the next section of this column.
   my $template = $self->load_tmpl( 'begin.tmpl', die_on_bad_params => 0 );
```

```
$template->param(PAGETITLE => 'Welcome Page');
$template->param(FULLNAME  => 'Mr. Mxyzptlk');
$template->param(BOOKURL=>
  'http://oreilly.com/catalog/9780596006396/');
return $template->output();
```

This is easier and more flexible than the method we saw in the first column of using CGI.pm methods to construct pages. As a related aside, I should point out that there is nothing (the name notwithstanding) that requires you to generate HTML using HTML::Template templates. In the last sizable webapp I built, I used the built-in HTML::Template support to have the webapp generate custom shell scripts that could be run separately from the webapp. At some point you start to realize that all the world's a template.

CGI::App Plugins

Now let's move into the territory of CGI::App enhancements provided by various plugin modules. I'd like to highlight three of them and then finish up with a small example that demonstrates everything we've talked about in these two articles. CGI::Application::Plugin::ValidateRM is the first plugin I'd like to mention.

CGI::APPLICATION::PLUGIN::VALIDATERM

Any decent Web application will validate its input. If your form asks for a phone number, the users need to be told they've done something wrong if they type in a set of letters (PEnnsylvania 6-5000 notwithstanding). You can block input errors via JavaScript before they are submitted, but for Perl programmers without JavaScript experience, it's probably easier to validate the input server-side with a module like Data::FormValidator. C::A::P::ValidateRM lets you hook Data::FormValidator into CGI::App in a relatively easy-to-use fashion. First we'll look at how Data::FormValidator is called and then at how C::A::P::ValidateRM lets us use it in a CGI::App context.

Data::FormValidator expects to receive an "input profile." The input profile specifies which fields are optional/required, any constraints on the field, filters to run on the input before checking, and what to do about errors. Here's an example input profile:

```
{  required=> [qw(inputfirstname inputlastname)],
   filters => ['trim'],
   constraint_methods => {
      inputfirstname => qr/^[A-Z]+[A-Za-z-'.]{1,25}$/,
      inputlastname => qr/^[A-Z]+[A-Za-z-'.]{1,25}$/,
   },
   msgs => {
      any_errors => 'err__',
      prefix => 'err_',
   },
}
```

Before validation is attempted, all fields are filtered, so leading and trailing whitespace is trimmed using a built-in filter (Data::FormValidator has a number of others). This profile expects there to be two required fields. Those fields have to be composed of one to twenty-five alpha characters (plus a few punctuation characters), specified here as regexps. We are using a custom constraint here, but Data::FormValidator::Constraint has a number of preset

constraints such as “ip_address,” “email,” and “american_phone” you could use—you don’t need to roll your own.

And, finally, in the input profile, we specify how validation results (messages, or msgs for short) will be reported. In the example above, we ask that something called err__ be set if there are any validation errors at all and that the error messages for each failed validation be returned using the convention err_{fieldname} (i.e., err_inputfirstname and err_inputlastname). Looking at how these validation results are used is a good segue to how forms are validated within CGI::App.

With C::A::P::ValidateRM, we use a method called check_rm():

```
use CGI::Application::Plugin::ValidateRM
        (qw/check_rm check_rm_error_page/);

# in some run mode subroutine...
my $results = $self->check_rm(
                          'get_basic_info',
                            {required=> [qw(inputfirstname inputlastname)],
                               ... # same stuff as above
                            }
) || return $self->check_rm_error_page();
```

The check_rm() method takes a “return run mode” and an input profile. If the input profile turns up any validation errors, the return run mode specified in the first argument will be called to produce an error page. This error page is returned instead of any HTML the run mode would normally generate.

The parts of the validation handling are starting to come together, so let’s review and see what we are missing. When CGI::App enters a run mode, validation code checks that the input sent to that run mode (e.g., posted from a form) is valid. If it isn’t, CGI::App calls the return run mode (usually the run mode we just came from) and asks it to produce an HTML error page so that the user can try resubmitting. How does a run mode actually generate an error page? That’s the last part we have to cover.

First, each run mode is passed a hash reference as a second argument if it is being called as a return run mode. This hash reference contains the validation error messages mentioned earlier. To make use of them via the templating already discussed, we add a line to each run mode to insert the validation results into the document being generated:

```
sub welcome_doc : Runmode {
   my $self = shift;
   my $errs = shift;

      ... # do the stuff for this run mode including load_tmpl()
   $template->param($errs) if ref $errs;
}
```

The bold line above inserts the error information into our template. We’ll need a slightly more sophisticated template to incorporate these error messages:

```
<html>
<head><title><TMPL_VAR NAME=PAGETITLE></title></head>
<body>
   <!-- TMPL_IF NAME="err__" -->
    <p> Some data in your form was missing or invalid. </p>
   <!-- /TMPL_IF -->
```

```
<form method="POST">
  <input type="hidden" name="rm"value="next_runmode" />
  First Name: <input type="text" size=30  name="inputfirstname"/>
  <TMPL_VAR NAME="err_inputfirstname"> <br>
  Last Name:</label> <input type="text" size=30  name="inputlastname"/>
  <TMPL_VAR NAME="err_inputlastname">
  <input type="submit" name="submit" value="Submit" />
</form>

</body>
</html>
```

This sample introduces the TMPL_IF syntax from HTML::Template. If the named parameter is present it will output the contents of the tag. In our validation input profile we specified that we wanted err___ set if any errors were encountered. Here's where that pays off. The other addition to our standard template is the use of TMPL_VAR NAME="err_{fieldname}" tags. These will get replaced with field-specific error messages (by default, CSS-styled in bold red text). The end result of all this fuss is that your webapp will accept input via a form, validate that input using a pretty powerful specification (worse case: written in Perl itself), and automatically spit out an error form with the valid information still filled in and any validation errors flagged with pretty error messages. All of that can be done with much less code than you'd normally need if you were writing it yourself.

That's the (considerable) positive side of using C::A::P::ValidateRM. The hidden negative side of using this plugin that isn't explicitly mentioned in any of the documentation is the increased complexity validation like this can add to each of your run mode subroutines. Before, your code path might have been super simple—you enter each run mode, do your work, and move on to the next run mode—but now you might have a second entry path into each run mode to handle validation errors. If you've written the run mode code to expect to run only once per session/user, any situation where the webapp doubles back on itself can complicate lots of things. It certainly can make debugging the application a little harder. Caveat coder.

CGI::APPLICATION::PLUGIN::SESSION

The next plugin we are going to see requires considerably less background to cover. CGI::Application::Plugin::Session lets you use the handy CGI::Session module easily from within CGI::App. Because HTTP is a stateless protocol, a Web programmer has to do a bit of work in concert with the user's browser to present the illusion that the user is operating within a "session." CGI::Session handles all of the behind-the-scenes plumbing for that (cookies, session caches and expiry, etc.). Using all of this power becomes really easy thanks to CGI::Application::Plugin::Session:

```
use CGI::Application::Plugin::Session;

sub run_mode_A : Runmode {
  my $self = shift;

  $self->session->param('some_parameter' => 'some value to store');
}

sub run_mode_B : Runmode {
  my $self = shift;
  $some_value = $self->session->param('some_parameter');
}
```

In the example above, the two run modes share a piece of information called "some_parameter" by storing and retrieving it as a session parameter. Neither run mode has to know just how that magic takes place behind the scenes. You can tweak just how this is done, e.g., what database is used for persistent storage, via CGI::Session setup arguments called from the plugin.

CGI::APPLICATION::PLUGIN::DBH

Our last CGI::App plugin for this column is another door opener. CGI::Application::Plugin::DBH makes it easy to bring the power of Tim Bunce's fundamental database-independent interface (DBI) package into your webapp. If your webapp needs to work with data found in an SQL database, C::A::P::DBH provides an efficient way to do it. The key efficiency this plugin provides is "lazy loading"; the plugin is smart enough not to spin up a database connection unless the specific run mode in play needs it.

To use C::A::P::DBH, you describe the DBI connection in the CGI::App initialization routine:

```
use CGI::Application::Plugin::DBH (qw/dbh_config dbh/);

sub cgiapp_init {
   my $self = shift;

   $self->dbh_config({standard DBI connect() arguments});
}
```

If you'd prefer to keep the configuration information in your instance script, arguably a better place for it, there's an alternative syntax mentioned in the documentation.

With your database configuration specified, the other run mode subroutines can make use of a DBI database handle:

```
sub lookup_data {
   my $self = shift;

   my $data = $self->dbh->selectrow_arrayref(qq{SELECT name,uid
         FROM users});
   # ... do something with $data->[0] and $data->[1]
}
```

C::A::P::DBH lets you use named DBI database handles if your webapp has a need for connections to multiple databases.

Sample Code

So let's put this all together into a toy webapp that demonstrates everything we've talked about in both parts of this series. We're going to look at the code in an outside-in fashion. The first piece of code a user executes is the instance script. This is the script that is called when we go to http://www.server.com/LoginExample.cgi:

```
use strict;
use lib '/path/to/webapps/lib';

use LoginExample;

my $app = LoginExample->new();
$app->run();
```

This super-simple script just spins up the application module and its run mode subroutines. That's a fairly sizable file (LoginExample.pm). Here it is in its entirety:

```
use strict;

package LoginExample;
use base 'CGI::Application';
use CGI::Application::Plugin::AutoRunmode;
use CGI::Application::Plugin::Session;
use CGI::Application::Plugin::DBH          (qw/dbh_config dbh/);
use CGI::Application::Plugin::ValidateRM   (qw/check_rm
        check_rm_error_page/);
use Data::FormValidator::Constraints       (qw/FV_length_between email/);

# Configure the database connection and the location of the templates
sub cgiapp_init {
   my $self = shift;
   $self->dbh_config( "dbi:SQLite:dbname=loginex.sqlite", "", "" );
   $self->tmpl_path('templates');
}

# The initial run mode displays a form and handles another pass
# through the form should it not be filled out correctly. It also
# stashes the time the script was first run by the user in a session object
sub get_userinfo : StartRunmode {
   my $self = shift;
   my $errs = shift;

   $self->session->param( 'starttime' => time )
        unless $self->session->param('starttime');

   my $template = $self->load_tmpl( 'getuser.tmpl',
        die_on_bad_params => 0 );

   # if we're showing errors from validation
   $template->param($errs) if ref $errs;

   return $template->output();
}

# First test to make sure it has received valid output. If it has, query a
# database and the session object for other fields and display the info
# via a simple template
   my $self = shift;

   # Validate input from getuser_info's form
   my $results = $self->check_rm(
      'get_userinfo',
      {  required              => [qw/fullname email/],
         filters               => ['trim'],
         constraint_methods    => {
            fullname           => FV_length_between( 1, 50 ),
            email              => email(),
         },
         msgs => {
            any_errors => 'err__',
            prefix => 'err_',
         },
      }
   ) || return $self->check_rm_error_page();
```

```
    my $template
      = $self->load_tmpl( 'showuser.tmpl', die_on_bad_params => 0 );

    my $email = $self->query->param('email');

    my $idnumber = $self->dbh->selectrow_array(
      qq{SELECT idnumber FROM users WHERE email = \"$email\"});

    $template->param( FULLNAME      => $self->query->param('fullname') );
    $template->param( EMAIL         => $email );
    $template->param( IDNUMBER      => $idnumber );
    $template->param(
      STARTTIME => scalar localtime $self->session->param('starttime') );

    $self->session->delete();
    $self->session->flush();

    return $template->output();
}

1;
```

After the modules are loaded, we perform all of our initialization work by configuring the database handle and template directory. Next come the two run modes for this module: get_userinfo and show_userinfo. The first run mode uses a template to display a two-field form (we'll see the template in just a moment) to collect information from the user. In the process, it makes note of the time the run mode was first entered by squirreling away that value in a session object. This run mode also handles a redisplay of that form should the user not provide valid information.

The second run mode, show_userinfo, checks the input it has been passed from get_userinfo. If the input is not valid, it creates an error page by calling get_userinfo again. If the input is valid, it retrieves info from the form parameters (`$self->query->param()`), an SQLite database (via DBI), and the session object (`$self->session->param()`). This information is inserted into a template, the session object is disposed of, and CGI::App is handed output to display.

To see what is displayed for each run mode, let's look at the two templates. Here's getuser.tmpl:

```
<html>
<head><title>Login Example </title></head>
<body>

  <!-- TMPL_IF NAME="err__" -->
  <p> Some data in your form was missing or invalid. </p>
  <!-- /TMPL_IF -->

<form method="POST">
<input type="hidden" name="rm" value="show_userinfo" />
Full Name: <input type="text" size=30  name="fullname"/>
<TMPL_VAR NAME="err_fullname"> <br>
Email: <input type="text" size=30  name="email"/>
<TMPL_VAR NAME="err_email">  <br>
<input type="submit" name="submit" value="Submit" />
</form>
</body>
</html>
```

and here's showuser.tmpl:

```
<html>
<head><title>Login Example </title></head>
<body>
   Full Name: <TMPL_VAR NAME=FULLNAME><br>
   Email: <TMPL_VAR NAME=EMAIL><br>
   ID: <TMPL_VAR NAME=IDNUMBER><br>
   Started: <TMPL_VAR NAME=STARTTIME>
</body>
</html>
```

Leftovers

We're very out of room (it's amazing how fast you can use up 140 characters), so let me just say that there are a number of CGI::App plugins you'll definitely want to explore if you decide to start building Web applications with it. The best place to start is to look at the "bundled" version being produced (search CPAN for "Titanium") to have all of the current "best practices" CGI::App plugins at your fingertips. Beyond that, search CPAN for "CGI::Application::Plugin" for a plethora of choices. Have fun writing webapps (perhaps for the first time) with CGI::Application. Take care, and I'll see you next time.