CGI.pm Basics

By icarus

Table of Contents

The Lazy Programmer

If you've been following along in our Perl 101 series of tutorials, you probably already know how Perl scripts can dynamically generate HTML pages in response to user input. You've learnt how to create a basic guestbook and form mailer, and you've understood the nitty−gritties of string and file manipulation.

Now that you've done all that hard work, I'm going to show you a few shortcuts that should help reduce the amount of time you spend on your Perl scripts. These shortcuts come in the form of a powerful Perl "module" called CGI.pm, which offers a number of interesting features and functions for the lazy Perl programmer.

Over the next few pages, I'm going to show you a few CGI.pm basics. So order a pizza, pop a can of Coke, unplug the telephone, and sit back − this is good stuff!

The Wonder That Is CGI.pm

First, we need to get the definitions down. In case you're wondering what a module is, don't − for the purpose of this article, just assume that it's a thingamajig that allows you to add new capabilities to your Perl program, or a series of pre−rolled functions which can be plugged in to your Perl program.

There are a number of such modules out there − CPAN, the Comprehensive Perl Archive Network, at http://www.cpan.org/ , has a complete list − and most are available free of charge, and are simple to import into your Perl program. If you're running a fairly recent version of Perl − say, 5.004 or higher − you probably already have CGI.pm installed as part of your distribution; if not, drop by CPAN and get yourself a copy.

You're probably wondering just what CGI.pm brings to the Perl party. Let me enlighten you.

In the Perl 101 series, we showed you to how to roll your own functions to parse the query strings generated when a form is submitted. With CGI.pm, all that is history − the module comes with powerful parsing capabilities that assist in the icky task of parsing query strings and separating them into individual name−value pairs.

Next, it also comes with some neat functions that simplify the task of writing HTML code. No more coding your way through <TABLE>s and <FORM>s − CGI.pm can generate them for you with simple, easy−to−understand−and−remember commands.

Finally, CGI.pm is written in the best traditions of object−oriented programming, fondly known as OOP. All the activities I've described above take place through a CGI object, which has its own methods and properties. In case you don't know what OOP is, you're probably not impressed. Don't worry about it − it's a good thing, and I'll be covering the basics of OOP in Perl in a separate article very soon.

Starting At The Top

There are two ways in which CGI.pm can be used − the manual refers to them as "function−oriented" mode and "object−oriented" mode. For the moment, I'll stick to function−oriented mode, and will demonstrate object−oriented mode as we proceed.

CGI.pm offers numerous functions that simplify the creation of HTML pages. For example, all HTML documents typically begin with

```
<html> <head> </head> <body>
```

and end with

```
</body> </html>
```

If you were using CGI.pm to generate your HTML page, you could generate all that code in one fell swoop like this:

```
#!/usr/bin/perl use CGI qw(:standard); print header(); print
start_html(); print end_html();
```

And here's what the output looks like when you browse to the page:

```
<!DOCTYPE HTML PUBLIC "−//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY></BODY></HTML>
```

Why stop there?

```
#!/usr/bin/perl use CGI qw(:standard); print header(); print
start_html(); print "Really?! You mean I don't need to
remember the difference between SUP and SUB anymore?"; print
br(), hr(); print end_html();
```

And here's what it looks like:

```
<!DOCTYPE HTML PUBLIC "−//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY>Really?! You mean I don't need to remember the
difference between SUP and SUB anymore?<BR><HR></BODY></HTML>
```

Thus, CGI.pm comes with numerous methods that allow you to generate the basic components of an HTML page. As the example above demonstrates, the start_html() and end_html() methods generate the opening and closing tags for an HTML document, while the p(), br() and hr() methods generate the <p>, <br> and <hr> tags respectively.

You can add a title to the document through the optional −title parameter to the start_html() function − the following code

```
print start_html(-title=>'The Light Dawns');
```

would generate a page with the title

```
<TITLE>The Light Dawns</TITLE>
```

You're probably wondering what the

:standard

means. The CGI.pm module is classified into different "families" of methods, thereby allowing the user to only import those methods into his or her Perl script which are absolutely necessary. The important families are:

:all − all available methods

:html2 − all methods which generate HTML code, as per the HTML2 specification

:html3 − all methods which generate HTML code, as per the HTML3 specification

:form − all methods related to form elements

:cgi − methods relating to CGI script processing (including parsing of query strings)

:standard − an umbrella family covering :html2, :form and :cgi

Pretty Pictures And Twirly Tables

Obviously, CGI.pm also allows you to add formatting to your document − the following example will demonstrate bold, italic and underlined text, together with the various block−level headings.

```
#!/usr/bin/perl use CGI qw(:all); print header(); print
start_html(−title=>'Test Drive'); print center(h1("Test
Drive")); print h4("Hmm. Let's see what else we can do with
this thing."); print p(); print "I can " . b("emphasize") . ",
" . i("italicize") . " and " . sup("superscript") . " text −
as a matter of fact, I just did!"; print end_html();
```

How about fonts, anchors and images?

```
#!/usr/bin/perl use CGI qw(:all); # header print header();
print start_html(); # set up some fonts # note how tag
attributes are placed in a hash, with the text outside the
hash print font({−face=>'Arial', −size=>'+1'}, 'Pick a
font...any font!'); print p(); # create an <IMG> tag print
img({−src=>'button.gif',−border=>'0'}); print p(); # or an
anchor − it's very similar to the font() method above print
a({−href=>'http://www.somewhere.com/somefile.html'}, 'And then
click me...'); print p(); # or you could get really advanced −
this is a clickable image print
a({−href=>'http://www.somewhere.com/somefile.html'},
img({−src=>'button.gif', −border=>'0'})); print end_html();
```

And here's the output:

```
<!DOCTYPE HTML PUBLIC "−//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY><FONT SIZE="+1" FACE="Arial">Pick a font...any
font!</FONT><P><IMG BORDER="0" SRC="button.gif"><P><A
HREF="http://www.somewhere.com/somefile.html">And then click
me...</A><P><A
HREF="http://www.somewhere.com/somefile.html"><IMG BORDER="0"
SRC="button.gif"></A></BODY></HTML>
```

Note the manner in which tags with attributes are represented in the code above − does it look familiar? If it does, it's because the various attributes and their values are placed in an associative array or hash as key−value pairs.

The dashes preceding each key are optional − the above example could be re−written like this:

```
#!/usr/bin/perl use CGI qw(:all); # header print header();
print start_html(); # set up some fonts # note how tag
attributes are placed in a hash, with the text outside the
hash print font({face=>'Arial', size=>'+1'}, 'Pick a
font...any font!'); print p(); # create an <IMG> tag print
img({src=>'button.gif',border=>'0'}); print p(); # or an
anchor − it's very similar to the font() method above print
a({href=>'http://www.somewhere.com/somefile.html'}, 'And then
click me...'); print p(); # or you could get really advanced −
this is a clickable image print
a({href=>'http://www.somewhere.com/somefile.html'},
img({src=>'button.gif', border=>'0'})); print end_html();
```

and still function correctly.

Finally, you can also easily create tables, complete with rows and cells, through the aptly named table(), Tr() and td() methods − take a look:

```
#!/usr/bin/perl use CGI qw(:all); # header print header();
print start_html(); print # table table({−border=>'1',
−cellpadding=>'5', −cellspacing=>'5'}, # table row
Tr({−align=>'left', −valign=>'top'}, [ # table cells
td(['R1,C1','R1,C2','R1,C3']), td(['R2,C1','R2,C2','R2,C3']) ]
) ); print end_html();
```

And here's what it looks like:

```
<!DOCTYPE HTML PUBLIC "−//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE>
</HEAD><BODY><TABLE CELLSPACING="5" BORDER="1"
CELLPADDING="5"><TR ALIGN="left" VALIGN="top"><TD>R1,C1</TD>
<TD>R1,C2</TD> <TD>R1,C3</TD></TR> <TR ALIGN="left"
VALIGN="top"><TD>R2,C1</TD> <TD>R2,C2</TD>
<TD>R2,C3</TD></TR></TABLE></BODY></HTML>
```

Within The Parameters

Once the form is submitted, the task becomes to split the various elements of the query string into individual name value pairs and make them available to Perl as variables. And CGI.pm comes well−equipped to do this with the powerful param() function.

If used without any parameters, the param() function will simply display a list of all available name−value pairs. Alternatively, if you need to obtain the value of a particular form field, you can simply provide the field name to param() as a parameter.

```
#!/usr/bin/perl # this is script.cgi − accepts form input and
displays welcome message use CGI qw(:all); print header(),
start_html(), "Hello, " . param('fname'), end_html();
```

What does this mean? It means that you can use param() to combine the query page and the result page into a single script, like this:

```
#!/usr/bin/perl # this is script.cgi − accepts form input and
displays welcome message use CGI qw(:all); # if param()
returns a result, it implies that the form has been submitted
if (param()) { print header(), start_html(), "Hello, " .
param('fname'), end_html(); } # else display the initial page
else { # header print header(); print
start_html(−title=>'Personal Information'); print
center(h1('So, Who Are You Anyway?')); print
start_form(−method=>'post', −action=>'script.cgi'); # text
field print "First name: " . textfield(−name=>'fname'); print
br(); print "Last name: " . textfield(−name=>'lname'); print
p(); print "Desired user name: " .
textfield(−name=>'username'); print br(); # password field
print "Desired password: " .
password_field(−name=>'password'); print p(); # drop−down
listbox print "Age: " . popup_menu(−name=>'age',
−values=>['Under 18', '18−30', '30−45', 'Over 45'],
−labels=>{'Under 18'=>'Young', '18−30'=>'Older',
'30−45'=>'Middle−aged', 'Over 45'=>'Out to pasture'}); print
p(); # radio buttons print "Sex: " . radio_group(−name=>'sex',
−values=>['Male', 'Female'], −default=>'Male'); print p(); #
checkbox print checkbox(−name=>'married', −value=>'yes',
−label=>'Married?'); print p(); # file upload button print
"Enter path to picture file: " .
filefield(−name=>'picture_file'); # hidden form data print
hidden(−name=>'timestamp', −value=>'200011011256'); print p();
# submit and reset buttons print submit("Register"),
reset("Start Over"); print end_form(); print end_html(); }
```

Note the manner in which param() can be used to check whether or not the form has already been submitted; if a call to param() returns nothing, it implies that the form has not yet been submitted.

The Object Of This Exercise Is...

For those of you familiar with object−oriented programming in Perl, you can also use CGI.pm in object−oriented mode. The primary difference here is that you first create a new CGI object, and then invoke methods on this object to create the form.

I've re−written the previous example using CGI.pm's object−oriented mode to demonstrate the differences.

```
#!/usr/bin/perl # this is script.cgi − accepts form input and
displays welcome message # note the difference in syntax when
using CGI.pm as an object use CGI; # create a new CGI object
$regform = new CGI; # if param() returns a result, it implies
that the form has been submitted if ($regform−>param()) {
print $regform−>header(), $regform−>start_html(), "Hello, " .
$regform−>param('fname'), $regform−>end_html(); } # else
display the initial page else { # header print
$regform−>header(); print
$regform−>start_html(−title=>'Personal Information'); print
$regform−>center($regform−>h1('So, Who Are You Anyway?'));
print $regform−>start_form(−method=>'post',
−action=>'script.cgi'); # text field print "First name: " .
$regform−>textfield(−name=>'fname'); print $regform−>br();
print "Last name: " . $regform−>textfield(−name=>'lname');
print $regform−>p(); print "Desired user name: " .
$regform−>textfield(−name=>'username'); print $regform−>br();
# password field print "Desired password: " .
$regform−>password_field(−name=>'password') ; print
$regform−>p(); # drop−down listbox print "Age: " .
$regform−>popup_menu(−name=>'age', −values=>['Under 18',
'18−30', '30−45', 'Over 45'], −labels=>{'Under 18'=>'Young',
'18−30'=>'Older','30−45'=>'Middle−aged', 'Over 45'=>'Out to
pasture'}); print $regform−>p(); # radio buttons print "Sex: "
. $regform−>radio_group(−name=>'sex', −values=>['Male',
'Female'], −default=>'Male'); print $regform−>p(); # checkbox
print $regform−>checkbox(−name=>'married', −value=>'yes',
−label=>'Married?'); print $regform−>p(); # file upload button
print "Enter path to picture file: " .
$regform−>filefield(−name=>'picture_file'); # hidden form data
print $regform−>hidden(−name=>'timestamp',
−value=>'200011011256'); print $regform−>p(); # submit and
reset buttons print $regform−>submit("Register"),
$regform−>reset("Start Over"); print $regform−>end_form();
print $regform−>end_html(); }
```

Obviously, you should use whichever one you're more comfortable with − although an object does offer certain traditional advantages.

You Have Mail!

Finally, I'll wrap this up by creating a simple form mailer using CGI.pm. This is similar to the one created in Perl 101, except that this version will be written entirely through CGI.pm constructs. If you compare the two, you'll see that this version is more compact and easier on the eye.

```
#!/usr/bin/perl # feedback.cgi − sets up feedback form mailer
use CGI; $feedback = new CGI; # if form has been submitted if
($feedback−>param()) { # open mail pipe open
(MAIL,"|/usr/sbin/sendmail −t"); print MAIL "To:
<webmaster\@yoursite.com>\n"; print MAIL "From: Feedback Form
Mailer\n"; print MAIL "Subject: Feedback on your site\n\n";
print MAIL "Here is the result of your feedback form.\n\n"; #
insert values from form print MAIL "Name: " .
$feedback−>param('who') . "\n"; print MAIL "Email address: " .
$feedback−>param('email') . "\n"; print MAIL "Age: " .
$feedback−>param('age') . "\n"; # close mail pipe close MAIL;
# print thank−you message print $feedback−>header(),
$feedback−>start_html(),
$feedback−>center($feedback−>h1('Thank you for your
feedback!')), $feedback−>end_html(); } else # if form has not
been submitted { # print form print $feedback−>header(),
$feedback−>start_html(−title=>'Feedback Form'),
$feedback−>h1('Feedback Form'),
$feedback−>start_form(−action=>'feedback.cgi',
−method=>'post'); print "Name: ",
$feedback−>textfield(−name=>'who', −size=>'30'),
$feedback−>p(); print "Email address: ",
$feedback−>textfield(−name=>'email', −size=>'30'),
$feedback−>p(); print "Age: ",
$feedback−>textfield(−name=>'age', −size=>'2'),
$feedback−>p(); print $feedback−>submit('Send mail'); print
$feedback−>end_form(), $feedback−>end_html(); } # EOF
```

And that's about it. See you soon!