PERL

a language by Larry Wall

Practical Extraction and Report Language
or
Pathologically Eclectic Rubbish Lister

Tom Christiansen
<tchrist@perl.com>

There's more than one way to do it!

Overview

- We’re here to learn how to write interactive applications of varying complexities in perl.
- This course presupposes a basic knowledge of perl, essentially equivalent to the beginning course I teach.
- Fortunately, almost everything we’ll do here is merely learning how to use libraries and modules written by others, so you won’t actually have to re-invent it all yourself.
- Everything here is written using the current release of perl, often known as perl5. Version 5 of perl was released in 1994. If you’re still using version 4, you’re now 5 years behind, as that was release in late 1990.

Further Info

The World Wide Web

- Today it seems that everyone and their cat has a home page on the Web. It's relatively straightforward to put together the HTML to do this. The Beginner's Guides to HTML can be found at
  `http://www.ncsa.uiuc.edu/demoweb/html-primer.html`
- When it comes time to write an interactive web page (perhaps a forms entry page) using a CGI script, many complex matters arise, including parameter passing, debugging, security, and maintaining state.
- Fortunately, not only is perl an excellent target language for such scripts, extensive CGI-related libraries and modules in perl have already been developed.

Net Resources

- A *majordomo* mailing list for CGI programming in perl has been established. Send a subscribe request to get on the list to:

  `CGI-perl-request@webstorm.com`

- Besides the obvious newsgroup *comp.lang.perl*, you should also check out the newsgroup *comp.infosystems.www.authoring.cgi* and gets its FAQ.

CGI Programs

- A Beginner's Guides to CGI can be found at `http://www.ncsa.uiuc.edu/` in the `/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html` file. See also `http://www.bio.cam.ac.uk/web/form.html`
- Just as server-side includes are often restricted to certain directories, your server may only permit CGI scripts to execute from a particular directory, such as `...../httpd/cgi-bin/` or with a certain extension, like `.cgi`.
- You'll need to consult your server's configuration documentation or check with your local adminstrator to see how your web server has been configured.

More HTTPD Config Issues

- **When someone executes your CGI program, it will *not* run as you, but rather under whatever UID your HTTP daemon has chosen, often "nobody" (uid == -2). This is a useful but not entirely sufficient security step.**
- **The script may not have access to files outside the spool area set up by the administrator, who many have configured the server to run CGI programs under a *chroot* (2)ed environment.**
- **Perl's low-level access to your processes real and effective user and group IDs, environment variables, current working and chrooted directories, plus its tainting checks and and the new `safeperl` module help address all these concerns.**

Browser Incompatibilities

- **Many old browsers have difficulties with "fancy new features" like `VALUE="string"` in SELECT widgets. Some even have problems with properly formated HTML.**
- **The non-Unix PCs and mass-market online services seem particular to bad behaviour here.**
- **Many new browsers silently do the "right" thing even on improperly formed HTML.**
- **Optimally, you should test your pages and cgi output against a variety of browsers, incluing *lynx*, *mosaic*, and *netscape* as a bare minimum. Strenuously avoid being tempted into relying on *netscape*-specific features.**

Verifying Your Browser and Web Pages

- **Neil Bowers <neilb@khoral.com> has a very nice program in *weblint*. You may get it from `ftp.khoral.com` in `/pub/perl/www/weblint-1.011.tar.gz`**

- **Henry Churchyard <churchh@uts.cc.utexas.edu> has written a verification suite in `htmlchek`, which you can find in the perl/aux/ directory. Run as**

```
perl htmlchek.pl somefile.html
```

- **An extensive test suite and browser bug list can be found at**

  `http://www.research.digital.com/nsl/formtest/home.html`

CGI Without Forms

- **There's nothing that says that you have to execute a CGI program from a form. You can call it directly, as I do in my *picknewsgroups* interface, included in the appendices. (You should be careful of programs being called like this when they're intended to only be called via a form.)**

- **If no arguments are given, it produces a certain kind of output that includes calls back to intelf:**

```
<UL>
<LI><A HREF="http://mox.perl.com/cgi-bin/picknewsgroups">
    All of Usenet</a>
<LI><A HREF="http://mox.perl.com/cgi-bin/picknewsgroups?rec.games">
    rec.games</a> hierarchy
<LI><A HREF="http://mox.perl.com/cgi-bin/picknewsgroups?comp.lang">
    comp.lang</a> hierarchy
</UL>
```

More Web Programming Without Forms

- Sometimes you need to produce a complex user-interface but the complexity is in producing it, not executing what it calls. For example, I have something of a threaded newsreading interface just on particular newsgroups, author, and organizations available under:
`http://mox.perl.com/deckmaster/usenet/`

- The layout is relatively simple looking, but to figure it out requires chatting extensively with an NNTP server. The code to do that is in the *getwotc* program, included in your appendices.

What Forms Look Like

- A CGI form should look something like this:

```
<FORM METHOD="POST"
   ACTION="http://www.host.org/cgi-bin/scriptname">

    <!-- user fields go here -->

</FORM>
```

- The ACTION is the script to execute.

- You should always use the POST method, not the old GET method. The POST method has no limitations on input data size because it reads from <STDIN>, but the GET method relies upon argument passing.

Types of Form Fields

- The major types of fields are `<INPUT>` fields, `<SELECT>...</SELECT>` fields, and `<TEXTAREA>...</TEXTAREA>` fields.
- All take a `NAME="some name"` option so you can identify them.
- You may not nest forms, but multiple forms may appear in the same documents.

<INPUT> Fields

- When you make an input field, you must specify one of several types.
  `<INPUT NAME="my name" TYPE="vide infra">`
- Valid input types are:
  - `text`: a text entry field (default)
  - `password`: like text but you can't read the input
  - `checkbox`: an isolated toggle button
  - `radio`: a related toggle button; all with same NAME are highlander buttons (there can be only one)
  - `submit`: send the whole query to the FORM's ACTION, that is, your CGI script
  - `reset`: quickly restore form to its default values
  - `hidden`: pass a value secretly

The HIDDEN INPUT Type

- CGI scripts sometimes rely on hidden fields to pass into from their forms that aren't supposed to be changed by the user. Here's an example of version control checking:

```
<FORM
    METHOD="POST"
    ACTION="http://mox.perl.com/cgi-bin/MxFrontEnd">

<INPUT
    TYPE="hidden"
    NAME="Version_Number"
    VALUE="1.901" >
<INPUT
    TYPE="hidden"
    NAME="Version_Date"
    VALUE="Fri Jul 28 08:18:50 MDT 1995">
```

The SUBMIT and RESET input tags

- Use the **`<INPUT TYPE="SUBMIT">`** tag to fire off the form to your CGI script.
- Some browsers tolerate multiple SUBMITs, but others don't!

```
<INPUT TYPE="submit" VALUE="Walk">
    or
<INPUT TYPE="submit" VALUE="Run">
```

- The RESET tag is used to return the form to a pristine state without having to reload it from the server.

```
<INPUT TYPE="reset" VALUE="Clear Lookup Field">
```

The <SELECT> Tag

- Use **`<SELECT>`** when you want one out of many, or when you many out of many. For example:

```
Are you ready?
 <SELECT NAME="Ready">
    <OPTION SELECTED>No
    <OPTION>Yes
    <OPTION>Maybe
 </SELECT>
 <BR>
```

- The "Ready" field will be set to either "No", "Yes", or "Maybe", depending on the user's selection. If they don't mark anything, it will use "No", the SELECTED default.

Multiple SELECTs

- **Use the MULTIPLE option when you want the users to be able to specify many possibilities:**

```
<SELECT MULTIPLE NAME="colors" SIZE=7>
    <OPTION>White
    <OPTION>Green
    <OPTION>Red
    <OPTION>Black
    <OPTION>Blue
    <OPTION>Gold
    <OPTION>Colorless
</SELECT>
```

Warning on SELECTs

- **Sometimes you find yourself tempted to write this:**

```
<SELECT NAME="colors">
    <OPTION>Blue
    <OPTION>Red
    <OPTION VALUE="Blue">Indigo
    <OPTION VALUE="Red">Crimson
</SELECT
```

- **The problem is that many antequated browsers don’t return "Blue" as they should if you select "Indigo" –> they return "Indigo" instead!**

The <TEXTAREA> Tag

- Use the **`<TEXTAREA>`** tag to give the user a large area to type in. This doesn't work very well in *lynx*, though.

```
<TEXTAREA NAME="HeSaid" ROWS=6 COLS=50>
  This has some default text.
  It may even span multiple lines.
</TEXTAREA>
```

- One annoyance with this is that so far, most browsers give obsequious lip-service to *emacs* bindings only. It would be nice if every browser put up a <USE $EDITOR> button for TEXTAREAs. Maybe someday....

Example Form

```
<HTML>
<HEAD>
    <LINK REV="made" HREF="mailto:tchrist@mox.perl.com">
    <TITLE>
        MoxPerl Fuzzy Card Name Matcher
    </TITLE>
</HEAD>

<BODY>
<FORM METHOD="POST"
      ACTION="http://www.perl.com/cgi-bin/pickcards" >

 <P> Indicate the <i>name of the card</i>
    you want here and hit then return or
    <CODE><B>&lt;Enter Query &gt;</CODE></B>: <br>
    <INPUT NAME="cardname" SIZE=40>
    <INPUT TYPE="submit" VALUE="Enter Query">
            or
    <INPUT TYPE="reset" VALUE="Clear Lookup Field">.
</FORM> </BODY> </HTML>
```

The cgi_handlers.pl Library

- James Tappin <sjt@xun8.sr.bham.ac.uk> has developed a simple perl4 library to handle the basic needs of a CGI program responding to either a GET or POST method. It was written in 1993.

- The library provides four simple utility functions:

  - *get_request()* to read the request into `%rqpairs`
  - *url_decode()* to URL decodes a string or array of strings
  - *html_header()* to send the HTML header back to the caller
  - *html_trailer()* to send the HTML trailer back to the caller

Example Using cgi_handlers.pl

```
#!/usr/bin/perl -w

require 'cgi_handlers.pl';

get_request();

$search_string = $rqpairs{cardname};
$title = "Query Results for ``$search_string''";
html_header($title");

run_user_query(); # TBD

html_trailer();
```

Getting the User's Values

- `get_request()` **loads up all the user's form variables. Thus, if we had an input field NAME'd "cardname", then** `$rqpairs{cardname}` **contains its value.**

Content-Type

- **If you don't execute an `html_header()` call, the user will get a mysterious message about their being a problem with the script. The header should at a minimum contain this:**

```
Content-type: text/html
<html>
      <head>
          <title>$title</title>
      </head>

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

Location Redirects

- However, if you're going to do a redirect, you must *not* output the content-type line. Instead do something like this:

```
$file = "http:/www.host.org/path.html";
print "Location: $file\n\n";
exit;
```

Problems with cgi_handlers

- The `cgi_handlers.pl` library won't properly return a list from a multiple select widget. The `cgi-lib.pl` one will, but they're separated by nulls, requiring that you split them on `/\0/`.
- It also doesn't provide an output formating function. This is important because you can't just blinding output things like < or it will look like a tags.
- Instead, you have to use things something like **<** instead.
- A list of escapes can be found in **http://www.ncsa.uiuc.edu/demoweb/html-primer.html#A1.5.3**

The cgi-lib.pl Library

- **Another popular perl4 library is cgi-lib.pl, written by <S.E.Brenner@bioc.cam.ac.uk> in 1995.**

- **This library supports routines ReadParse, PrintHeader, MethGet, MethPost, MyURL, CgiError, CgiDie, PrintVariables, and PrintVariablesShort.**

- **It's somewhat easier to debug due to the print functions, but is essentially similar to the simple `cgi-handlers.pl` library.**

Perl5 Solutions

- The `CGI.pm` module (current version 1.53 of 6/27/95) was written by Lincoln Stein <lstein@genome.wi.mit.edu>. Extensive documentation can be found in
  `http://www-genome.wi.mit.edu`
  `/ftp/pub/software/WWW/cgi_docs.html`

- This library is object-oriented (but not oppressively so) and offers a wide variety of conveniences:
  - enhanced debugging
  - support for array returns
  - form-creation primitives
  - support statefulness

Example from the CGI.pm docs

```
use CGI;
$query = new CGI;
print $query->header;
print $query->startform;
print "What's your name? ",$query->textfield('name');
print "<P>What's the combination? ",
        $query->checkbox_group('words',['eenie',
                'meenie','minie','moe']);
print "<P>What's your favorite color? ",
        $query->popup_menu('color',['red',
                'green','blue','chartreuse']);
print "<P>",$query->submit;
print $query->endform;
print "<HR>\n";
if ($query->param) {
    print "Your name is <EM>",$query->param('name'),"</EM>\n";
    print "<P>The keywords are: <EM>",
            join(", ",$query->param('words')),"</EM>\n";
    print "<P>Your favorite color is <EM>",
    $query->param('color'),"</EM>\n";
}
```

CGI.pm Example Explained

- The example on the previous slide is both the generator and the consumer of the script in question. The first time your run it, $query−>param is false. After that, it will maintain the state of the last submission.

- While `CGI.pm` is fine for small applications, but the author recommmends: “Although it is easy to get up and running with this module, for more advanced applications, such as complicated state-maintaining behavior and linked forms, you are encouraged to use the complete, full-featured perl CGI modules.”

HTML Result of CGI Query

```
<TITLE>A Simple Example</TITLE>
<A NAME="top">
<H1>A Simple Example</H1>
</A>
<FORM METHOD="POST" >
What’s your name? <INPUT TYPE="text" NAME="name" VALUE="tchrist"><P>What’s
the combination?<P>
<INPUT TYPE="checkbox" NAME="words" VALUE="eenie" CHECKED>
eenie <INPUT TYPE="checkbox" NAME="words" VALUE="meenie" >meenie
<INPUT TYPE="checkbox" NAME="words" VALUE="minie" >minie <INPUT
TYPE="checkbox" NAME="words" VALUE="moe" >moe <P>What’s your favorite
color? <SELECT NAME="color">
<OPTION  VALUE="red">red
<OPTION  VALUE="green">green
<OPTION  VALUE="blue">blue
<OPTION SELECTED VALUE="chartreuse">chartreuse
</SELECT>
<P><INPUT TYPE="submit"  ></FORM>
<HR>
Your name is <EM>tchrist</EM>
<P>The keywords are: <EM>eenie</EM>
<P>Your favorite color is <EM>chartreuse</EM>
<P><A HREF="cgi_docs.html">Go to the documentation</A>
```

Debugging Your Script

- One of the most frustrating things about CGI programming debugging. If you use the `CGI.pm` module, or the `CGI::*.pm` modules described below, you'll be able to avoid a lot of this pain.
- You can pass parameters to your program on the command line:
  ```
  $ perl -d cgi_script n1=v n2=v2
  ```
- Or you can pass them from the stdin or another file:
  ```
  $ cgi_script
  name1=value1
  name2="value2 and more"
  ^D
  ```

The CGI::*.pm Modules

- Originally developed by Tim Bunce, author of many things perlish, these are now maintained by Lincoln Stein <lstein@genome.wi.mit.edu>, the `CGI.pm` module's author.

- You can get these from `http://www-genome.wi.mit.edu` in `/ftp/distribution/software/WWW/CGIperl/`

- Three modules are included:
  - `CGI::Base.pm`: common base class
  - `CGI::Request.pm`: parse client request via a CGI interface
  - `CGI::MiniSvr.pm`: use socket tricks for a stateful server

CGI::Base

Here's an example from the CGI::Base documentation:

```
use CGI::Base;
$cgi = new CGI::Base;         # reads vars from environment
$cgi->var($name);             # get CGI variable value
$cgi->var($name, $value);     # set CGI variable value
@names  = $cgi->vars;         # lists standard CGI variables

$mime_type  = $cgi->accept_best(@mime_types);
$preference = $cgi->accept_type($mime_type);

$cgi->pass_thru($host, $port); # forward request to server
$cgi->redirect($url);          # redirect client

$cgi->done($dump);      # end response, does NOT send </BODY>
$cgi->exit(@log_msgs); # exit, optionally logging messages
```

Using CGI::Base

- If you ask for a new CGI::Base object, you can access CGI variables thru its `var()` method. You may also import the standard CGI variables using the `:CGI` import tag.

```
use CGI::Base qw(:DEFAULT :CGI);

my $cgi = new CGI::Base;
$agent = $cgi->var('HTTP_USER_AGENT')
          || "an unknown agent";
$user = $REMOTE_USER || $REMOTE_IDENT
          || "unknown user";
$host = $REMOTE_HOST || $REMOTE_ADDR
          || "unknown user";
```

More CGI::Base

- The `html_escape()` function should always be used when you are emitting random text from a CGI bin program (assuming your content type is html, that is).

```
@escaped_texts = html_escape(@texts);
print @escaped_texts;
```

- The `html_br_lines()` function will convert `\n` into `<BR>` sequences.

- The `SendHeaders()` function will send and flush HTTP header(s). Currently it's not idempotent.

Logging

- **One of the more useful features of the CGI:: modules is their logging ability:**

```
use CGI::Base;
CGI::Base::LogFile("/scratch/cgi-log")
$cgi = new CGI::Base;

$cgi->log("Query from $user\@$host using $agent");
```

- **A `log_request()` method is also supported to log the full request.**

CGI::Request

- **I mostly just use CGI::Base to get at CGI::Request. Here's an example request tester:**

```
#!/usr/bin/perl

use CGI::Request;
GetRequest();
print FmtRequest();

print "Envariables:<UL>\n";
for $var (sort keys %ENV) {
    print "<LI> $var = ",
          html_escape($ENV{$var});
}
print "</UL>\n";
```

More CGI::Request Usage

- **Where with `cgi-handler.pl` we used the `%rqpairs` array, we now use an object method to access the user fields:**

```
$req = new CGI::Request;
unless ($req) {
  die "Couldn't read request packet: broken browser?";
}
$search_string = $req->param('cardname');
```

Importing Request Variables

- It’s often convenient to import request parameters as regular perl variables.

```
$req->import_names('R');
print "$R::FieldName";
print "@R::SelectMultiField";
```

- If you try to access a select multifield in a scalar context, you’ll get a fatal exception, as in
  `$n = $req->param('SelectMultiField');`

- You may not clobber your main namespace!

Security Concerns: RegExps

- Contrary to popular belief, it's not dangerous to do this:

```
$pat = $req->param("user_pattern");
if ($var =~ /$pat/) { ... }
```

- Even if they were able to sneak something nasty into the pattern, it won't get re-evaluated without an eval. The thing you *mustn't* do is this:

```
$pat = $req->param("user_pattern");
eval "\$var =~ /$pat/";
```

More Security Concerns

- **You should definitely turn on the `-T` flag to taint-test your script, even if you aren't going to run setuid. It'll catch things you don't even think of. This is *extremely important* because using remotely-provided data as anything other than bits to append to a data file is potentially dangerous. Taint checks will stop you from using things from outside your program to affect anything else outside of it, except in a restricted fashion.**

- **Malcolm Beattie has a "safeperl" extension in alpha test right now which seems promising enough that you should even be able eval safely! Get it from `ftp.ox.ac.uk` in `/pub/perl/Safe-a2.tar.gz` and don't miss the README there.**

Safe BackTicks

- The following isn't safe:

```
$pat = $req->param("user_pattern");
@lines = `grep @opts '$pat' @files`;
```

- You don't know that the user didn't put something horrible in the pattern. Instead, you should do something more like this:

```
if (open(GREP, "-|")) {
    @ok = <GREP>;
} else {
    exec '/usr/bin/grep', @opts, $pat, @files
    die "NOTREACHED: grep: $!";
}
```

Headers

- **Unless you're going to be doing redirects, it's best to send your content headers early.**

```
use CGI::Base qw(:DEFAULT :CGI);
use CGI::Request;
SendHeaders();
```

- **Note that if you have a compilation error or a run-time error, you won't catch that, and the user will be mysteriously confused. In perl5.001 and better, you could use a `$SIG{__WARN__}` or `$SIG{__DIE__}` trap. I did this in my experimental `CGI::ErrorWrap.pm` module.**
- **The `SendHeaders()` will be idempotent soon, so this should work better.**

Catching Warnings

- **Here's a sigwarn handler:**

```
BEGIN {
  CGI::Base::LogFile("/httpd/logs/errlog");

  $SIG{__WARN__} = sub {
    SendHeaders();
    print "<STRONG>";
    print "<CODE>WARNING: @_</CODE><BR></STRONG>\n";
    warn "WARNING!! @_";
  };
}
```

Catching Errors

```
$SIG{__DIE__} = sub {
    SendHeaders();
    print <<EODEATH;
    <HR>
    <STRONG>
    <H1>
            FATAL ERROR
    </H1>
<DL>
 <DT>Unexpected Fatal Error in $0:
 <DD> <CODE>
            @_
    </CODE>
</STRONG></DL>
<HR>
Please send mail to $MAILTO_ADMIN for help.

EODEATH
    die "FATAL!! @_";
};
```