More CGI Programming

Here-docs

HTML::Template

Cookies

File Uploading

Taint Checking

'Here-Doc'

- A special way of printing out large amounts of text
- This can actually be done in any Perl program, but I find it most useful in CGI, when you have to print large amounts of HTML

```
print <<HTML_END;
<html><head>
  <title>$title</title>
<head>
<body>
...
HTML_END
```

Here-Doc notes

- The ending Here-Doc marker must be on a line containing the marker followed by a newline. NOTHING ELSE.
  - including no leading spaces
- Output will be formatted exactly as you type it, including newlines, spaces, and tabs
- If starting marker is enclosed in double-quotes, or not enclosed in any quotes, all variables will be interpolated
  - `print <<"END_HTML";`
  - `print <<END_HTML;`
- If starting marker enclosed in single-quotes, variables will not be interpolated
  - `print <<'END_HTML';`
- By convention, the Here-Doc marker is in all caps

HTML::Template

- It is often a good idea to separate the design of your webpage from the programming logic of your CGI script.
- HTML::Template allows you to create the entire layout of your document in a separate file.
- Your CGI script will open this template, and simply fill in some variable values
- The only output your CGI script will print is the HTTP header and the return value of the template's **`output()`** method.

Template variables

- Your Template file is simply an HTML document, with some additional HTML-like tags.
- In your Template file:

```
<p>Hello <TMPL_VAR name="name">, you are
<TMPL_VAR name="age"> years old.</p>
```

- in your CGI script:

```
use CGI qw/:standard/;
use HTML::Template;
```

```
my $tpl = HTML::Template->new(
  filename=>'simple.tpl'
);
print header;
```

```
my $name = param('name');
my $age = 2010 - param('yob');
```

```
$tpl->param(name=>$name, age=>$age);
```

```
print $tpl->output();
```

Template loops

- In Template, **`TMPL_LOOP`**, with **`TMPL_VAR`**'s within.

```
<table>
<TMPL_LOOP name="info">
<tr>
  <td><TMPL_VAR name="id"></td>
  <td><TMPL_VAR name="color"></td>
</tr>
</TMPL_LOOP>
</table>
```

- In CGI, array of hashrefs.

```
my @props = (
  {id=>15, color=>'red'},
  {id=>27, color=>'blue'},
);
```

```
$tpl->param(info=>\@props);
```

Template if/else

- `<TMPL_IF name="error">`
  `<h1>ERROR: <TMPL_VAR name="error"></h1>`
  `</TMPL_IF>`
- if you call **`$tpl->param(error=>$err)`**, the template if block will be executed if and only if **`$err`** is a true value. If no error parameter is passed, or if **`$err`** is false, the template if will not be executed.
- `<TMPL_IF name="error">`
  `<!--stuff-->`
  `<TMPL_ELSE>`
  `<!--else stuff-->`
  `</TMPL_IF>`

Special Loop Variables

- Add an additional parameter to new():
- `my $tpl = HTML::Template->new (`
  `filename=>'simple.tpl',`
  `loop_context_vars => 1,`
  `);`
- Now, four pseudo-variables are available:
  - `__first__`, `__last__`, `__inner__`, `__odd__`
- `<TMPL_LOOP name="vals">`
  `<TMPL_IF __last__>and</TMPL_IF>`
  `<TMPL_VAR name="arg">`
  `</TMPL_LOOP>`

HTML::Template triviata

- If your syntax-highlighting editor balks at these non-HTML tags, you can make them comments:
  - **`<!--TMPL_VAR name="age"-->`**
- You can include one template in another:
  - **`<TMPL_INCLUDE name="other.tpl">`**
- If template value may contain HTML that you want escaped:
  - **`<TMPL_VAR name="page" escape="HTML">`**
  - **`$tpl->param(page=>$p);`**
  - if **`$p`** contains '**`<`**', '**`<`**' will actually be passed.

Importing Variables

- It can be tedious (and inefficient) to make several calls to param() to keep getting the same parameter value.
- We'd like to be able to refer to those parameters as normal Perl variables.
  - Well, thanks to CGI.pm, we can
- **`import_names()`** – takes current parameter list and creates scalar and array variable for each parameter name, with the parameter's value(s)
  - scalar is first value with that name, array is all
- defaults to being imported to package **`Q`**
  - You can change it by passing a string into **`import_names()`**
  - it won't allow you to import into **`main::`**

import_names()

- From the inputs example posted last week, if outputs.cgi called **`import_names`**, the following variables would spring into existence, with the following values:
- **`$Q::MyText`** → "This is a text field";
- **`@Q::MyText`** → ("This is a text field");
- **`$Q::fruits`** → 'Apples';
- **`@Q::fruits`** → ('Apples', 'Oranges');
  - (assuming both boxes were checked)
- **`$Q::colors`** → 'blue';
- **`@Q::colors`** → ('blue');
- etc

File Uploading

- One input method we did not talk about last week is a file-upload field.
- To use file-uploading feature, you must use a special kind of form.
  - Add **`ENCTYPE="multipart/form-data"`** to **`<form>`**
  - Or, **`start_multipart_form()`** instead of **`start_form()`**
- **`<input type='file' name='uploaded'>`**
- **`filefield(-name=>'uploaded')`**
- Creates a field in which the user can enter the name of the file to send to the server. Also creates a 'Browse' button to search the local machine.
- User enters name or path of a file to upload.
- The form is submitted, and the CGI script can then access this file.

Getting the File

- To get the name of the file the user wants to upload, use the **`param()`** function.
- **`my $file = param('uploaded');`**
- Return value of this **`param`**() call is a 'magic' variable
- If you use **`$file`** as a string, it will be the name of the file.
- If you use **`$file`** as a filehandle, it will be a filehandle opened for reading
-
```
print "Contents of file $file: <br>\n";
while (my $line = <$file>){
   print $line, br, "\n";
}
```

That's Great for Text Files…

- But users can upload any kind of file.
- You need to find out what kind of file it was.
- **`uploadInfo()`** function. Returns a reference to a hash containing info about the file.
- **`$file = param('uploaded');`**
- **`$info = uploadInfo($file);`**
- **`$type = $info->{'Content-Type'};`**
- **`$type`** may contain "text/html", "text/plain", "image/jpeg", etc etc…

If File is not Text

- Need a function to read from binary files.
- **`read($fh, $buffer, $size)`**
  - **`$fh`**→filehandle to read
  - **`$buffer`**→scalar in which to store data
  - **`$size`**→max number of bytes to read
  - returns number of bytes read
- **`my $file = param('uploaded');`**
-
```
open my $upload, '>', 'binary.jpg'
   or die "Cannot open… $!";
```
-
```
my $buf;
while (read($file,$buf,1024)) {
   print $upload $buf;
}
close $upload;
```
- This is not CGI-specific. **`read()`** can be used in any Perl-script as an alternative to **`<$fh>`**

Cookies

- A cookie is a (usually very small) piece of text that a server sends to a web browser for later retrieval.
- Can be used to 'track' a user's preferences, or any other information the user has told the server.
- When web browser requests a page, web server sends the page along with the tiny text file. Web browser saves text file
- When web browser requests same page (or another page on same server), sends the cookie file along with the request, so server can identify the client.
- (This all assumes client has not disabled cookies)

To Set a Cookie

- Create the cookie
- **`cookie()`** function. Takes many (mostly optional) parameters:
  - **`-name=>`** Name of the cookie
  - **`-value=>`** Value of the cookie – can be any scalar, including array reference or hash reference
  - **`-expires=>`** Expiration date/time of the cookie
  - **`-path=>`** Path to which cookie will be returned
  - **`-domain=>`** Domain to which cookie will be returned
  - **`-secure=>`** 1 if cookie returned to SSL only

Cookie Expiration

- Expires: absolute or relative time for cookie to expire
  - **`+30s`** ➔ in 30 seconds
  - **`+10m`** ➔ in 10 minutes
  - **`+1h`** ➔ in one hour
  - **`-1d`** ➔ yesterday (ASAP)
  - **`now`** ➔ immediately
  - **`+3M`** ➔ in 3 Months
  - **`+10y`** ➔ in 10 Years
  - **`Wed, 22-Apr-2010 16:00:00 EDT`** ➔ On Wednesday, 4/22/2010 at 4pm EDT.

Cookie Path

- 'region' of server to check before sending back the cookie.
- Set a cookie with **`-path => '/perl/s10/'`**
- Only CGI scripts in **`/perl/s10`** (and its subdirectories) will receive the cookie.
- By default, path is equal to the path of the current CGI script.
- To send cookie to all CGI scripts on server, specify **`-path => '/'`**

Cookie Domain

- domain (or partial domain) to which to send cookie back
- must contain at least 2 periods (so one can't send a cookie to all .com domains)
- if I set cookie's **`-domain => '.rpi.edu'`**, cookie will be sent to scripts on www.rpi.edu, www.cs.rpi.edu, cgi.cs.rpi.edu, etc
- if set to .cs.rpi.edu, cookie only sent to www.cs.rpi.edu, cgi.cs.rpi.edu, cgi2.cs.rpi.edu, etc
- if set to www.cs.rpi.edu, cookie sent only to pages on www.cs.rpi.edu
- Note that both domain and path must match cookie parameters in order to be sent.

Cookie Created, Now Send it.

- To send a cookie to the web browser:

```
my $cookie = cookie( ... );
print header(-cookie=>$cookie);
```

- To send more than one cookie, use array reference

```
my $cookie1 = cookie( ... );
my $cookie2 = cookie( ... );
print header(
  -cookie=>[$cookie1, $cookie2]
);
```

Retrieve the Cookies

- To retrieve a cookie that has already been stored on the web browser's machine, to check its value:
- Once again, use the **`cookie()`** function.
- This time, don't use any –value parameters. Just give the name
- **`my $mycookie = cookie('lallip');`**
- **`$mycookie`** now has value of the cookie named 'lallip'.

Tainting

- When your CGI script is executed, it is executed as though you yourself ran it.
  - At least on CSNet - other servers are different
- It has all permissions that you have.
- This is exceedingly dangerous.
- **`my $string = param('s');`**
  `system("echo $string");`
  - What if the parameter the user gave to your URL was: `hello world ; rm -rf *`

Trust No One

- If you enable "taint checking", Perl will mark all data external to the program as tainted, and will not let you use it to run any system commands or open any files, etc.
- On the shebang, after perl, add: ' -T'
  - (If you get an error about "too late for -T", try changing your shebang from /usr/bin/env perl to /usr/local/bin/perl)

Tainted File

- **`#!/usr/bin/env perl -T`**
  `use CGI qw/:standard/;`
  `my $string = param('s');`
  `system("echo $string");`
- Error: "Insecure dependency while running with -T"
- Perl knows that you haven't bothered making sure **`param('s')`** is "okay".
- You need to verify the contents of the parameter before you use it to run anything external.

Un-tainting

- The *only* way of telling Perl that a value is okay to be used (ie, is not tainted) is via regular expressions:
- **`if ($string =~ /^([a-z0-9 ]+)$/i){`**
  `$string = $1;`
  `} else {`
  `die "Not an allowed file!";`
  `}`
- The captured variables (**`$1`**, **`$2`**, **`$3`**) are untainted.

Not a substitute for your brain

- Taint checking merely lets you know that something might be amiss. It is still possible to do something dumb:
- **`if ($string =~ /^(.*)$/) {`**
  `$string = $1;`
  `}`
- Completely bypasses the taint checking.
- This is more of Perl's "Who am I to stop you from shooting yourself in the foot?" philosophy.

Environment is also tainted

- If you ever run an external program, you must EXPLICITLY set your PATH variable.
  - Otherwise, Perl says it's possible someone has changed the environment so that **`/my/evil/programs/ls`** is found first in the path, rather than **`/bin/ls`**
- **`$ENV{PATH} = '/bin:/usr/bin:/usr/local/bin';`**
- For more information, **`perldoc perlsec`**