Using CGI on the Web

Professor Don Colton, BYU Hawaii

September 20, 2004

In this section of the course, we learn to use CGI and Perl to create web pages with dynamic content (content that differs from time to time, or from user to user). This handout gives enough details to get you started. It does not cover many of the finer details that you can learn from other sources. It simplifies a great deal in some cases. The intent is to get you started. Once you are able to do a little, we will guide you to do more.

For convenience I will assume your login name is `aa999`. Naturally if your login name is different, you should use it in place of `aa999` in the instructions below.

1 Directories and Permissions

To set up your directories and permissions properly, type the following commands.

```
mkdir ~/public_html
mkdir ~/public_cgi
chmod 711 ~/. ~/public_html ~/public_cgi
```

The `mkdir` command will fail if the directory already exists. Don't worry about that.

On the first line, we create a `~/public_html` directory (folder) for your normal web pages. The name `~/public_html` is specified in the configuration file that the web server is using. It is the default for **apache**. When you have your own server you can change the configuration file to specify a different directory if you like.

The tilde character (`~`) in `~/public_html` is an old UNIX symbol to indicate the home directory. `~/` is your home directory. `~don/` is my home directory.

On the second line, we create a `~/public_cgi` directory (folder) for your CGI programs. The name `~/public_cgi` is also specified in the configuration file that the web server is using. It is not the default. We invented it. When you have your own server you can change the configuration file to specify a different directory if you like.

chmod

The third line looks very cryptic. In it we set the access permissions for dot (the current directory, which is your home directory) to `drwx--x--x`. We also set the permissions for `~/public_html` and `~/public_cgi` the same.

The `rwx` means the owner (you) can (r)ead, (w)rite, and e(x)ecute files in those directories. The first `--x` means everybody else in your workgroup can only execute. The second `--x` means every other user on your computer (but not in your workgroup) can only execute. The web server itself is a user on your computer, but probably not in your workgroup.

This permission setting is needed to allow the web server to use your files. Notice that 711 converted to three-bit (octal) binary is 111 001 001. Do you see the similarity between `111001001` and `rwx--x--x`?

Another standard feature of the UNIX file system is that a single dot refers to the "current" directory. Two dots refers to the "parent" directory. In this case, dot would mean your home directory (where you start when you log in).

2 Simple Web Pages (No CGI)

The most simple web page has no formatting at all. Just type some words into a file using your favorite

text editor. Save it. I will assume you called it `sample.txt`. To enable access to the file, use `chmod` to grant permission. Use the following commands:

```
emacs ~/public_html/sample.txt
(type a bunch of stuff, save, exit)
chmod 644 ~/public_html/sample.txt
```

Now start a browser and type in this url:

```
http://cs.byuh.edu/~aa999/sample.txt
```

It should display the contents of your file, just as you typed it.

2.1 Common Errors

There are several common errors that first-time programmers tend to encounter. If your page is not working, here are some things to check.

Forbidden: If you get a message like this, make sure your file permissions and directory permissions are set properly. Those are the `chmod` lines above. Sometimes people will get overzealous and chmod 711 on the actual text file. The right setting for the text file is chmod 644, which normally happens automatically without any extra effort on your part.

Not Found: If you get a message like this, make sure that your file is in your `~/public_html` directory. Sometimes people will accidentally create another `public_html` directory inside their main `~/public_html` directory, or will create their `public_html` directory inside their `cs201` directory. Use the `pwd` command (print working directory) to find out where you are editing and storing your file.

3 Web Pages With HTML

For our next step, we will create a web page that includes "mark up" tags to specify formatting and identify types of content. HTML stands for "hyper text markup language." In this section we will briefly identify the main markup elements you need to know. Remember that we are (here) greatly simplifying the world of HTML, and you can find more powerful and complex markup commands discussed elsewhere. Our purpose here is simply to get you started.

`<doctype` is used to tell the browser what type of document it is receiving, in what "language" it is written. In the programming labs we give you a standard doctype line to use. We will not explain it. It should be the first line in your web page.

`<html>` starts your document. `</html>` ends your document. Everything else goes between those tags.

`<head>` starts the heading portion of your document. `</head>` ends the heading portion of your document. The heading portion includes the title of your web page, which should be between the `<head>` and `</head>` tags.

`<title>` starts the title portion of your document. `</title>` ends the title portion of your document. Naturally the title goes between them.

`<body>` starts the content (body) portion of your document. `</body>` ends the content (body) portion of your document. All the content of your document goes between those two tags.

`<h1>` and `</h1>` delimit a main heading in your document. The words of the heading go between them.

`<h2>` and `</h2>` delimit a second-level main heading in your document. The words of the heading go between them.

`<p>` is used to end a paragraph.

`<br>` is used to break to a new line in the same paragraph.

`<hr>` is used to draw a horizontal rule across the web page.

That should get us started. Here is a sample web page using html.

```
<html><head><title>Aloha</title>
</head><body>
<h1>Aloha from BYUH!</h1>
This is a short web page!<p>
Here is a line:<p>
<hr>
Here are two more lines:<p>
```

```
<hr><hr>
Well, that's all folks!<p>
</body></html>
```

Build this page, save it, and set the permissions as follows:

```
emacs ~/public_html/sample.html
(type in the lines above, save, exit)
chmod 644 ~/public_html/sample.html
```

Now start a browser and type in this url:

```
http://cs.byuh.edu/~aa999/sample.html
```

It should display the contents of your file, but instead of `<hr>` you should see a horizontal rule (line).

4 A Simple CGI Program

Here is the exact same web page done up as a Perl CGI program.

```
#!/usr/bin/perl -Tw
print "Content-type: text/html\n\n";
print "<html><head><title>Aloha</title>\n";
print "</head><body>\n";
print "<h1>Aloha from BYUH!</h1>\n";
print "This is a short web page!<p>\n";
print "Here is a line:<p>\n";
print "<hr>\n";
print "Here are two more lines:<p>\n";
print "<hr><hr>\n";
print "Well, that's all folks!<p>\n";
print "</body></html>\n";
```

Basically every line of the web page is replaced by a `print` statement. There is one major difference: the first line printed says `Content-type`. Normally when we display a web page, the server knows what kind of page it is. Often this is based on the last few letters of the file name, but this may vary from server to server. A Macintosh server may behave differently than a Linux server, which may behave differently than a Microsoft server, etc.

The client (our web browser) does not know what kind of server it is using, or what the file naming rules are for that operating system. Normally the server tells the "mime type" to the client. When we run a CGI program to create a web page, the server no longer knows what kind of page it is, so our program must inform the client.

4.1 A Shorter Version

The perl print statement allows a multi-line string to be printed. This can save you a lot of typing if you are copying code from a web page that you created using DreamWeaver or some other tool.

```
#!/usr/bin/perl -Tw
print "Content-type: text/html

<html><head><title>Aloha</title>
</head><body>
<h1>Aloha from BYUH!</h1>
This is a short web page!<p>
Here is a line:<p>
<hr>
Here are two more lines:<p>
<hr><hr>
Well, that's all folks!<p>
</body></html>
";
```

The thing to watch for is quote marks and dollar signs within the string. If they are double quotes they need to be escaped by adding a back-slash before each one. The same is true for dollar signs.

It is particularly important to make sure there are no spaces at the end of the Content-type line. Because the syntax is so unforgiving, I recommend that your first print line look like this:

```
print "Content-type: text/html\n\n
```

Putting the `\n\n` on that line will guarantee the right syntax is passed on to the browser.

4.2 Testing Locally

To run this program, we must type it in and give it a name. Let's say we called it `sample`.

Note: on some servers, you must give your program a name that ends with either `.cgi` or `.pl` because those are the two file extensions that your server will recognize as CGI programs. This varies by server. You can control it from the configuration file.

```
emacs ~/public_cgi/sample
(type in the program just above)
chmod 711 ~/public_cgi/sample
```

Next we set the file access permissions to `711` making the file executible by the server. All CGI programs should have permissions of `711`, whether they are written in perl or C or some other language.

On some operating systems, you may need to set the file access permissions to `755` making the file both readable and executible by the server. In such cases, since perl is a scripting language, the server must be able to both read and execute our program. With a compiled C program, the server only needs to execute the program.

The main thing to notice is that you have a couple of possibilities. Try one. If it does not work, try the other.

We can test our program by running it directly where it is.

```
~/public_cgi/sample
```

When you type in the command to run your program, you should see the following output on your screen:

```
Content-type: text/html

<html><head><title>Aloha!</title>
</head><body>
<h1>Aloha from BYUH!</h1>
This is a short web page!<p>
Here is a line:<p>
<hr>
Here are two more lines:<p>
<hr><hr>
Well, that's all folks!<p>
</body></html>
```

4.3 Testing On The Web

When it runs properly (like this), we can move to the next step and run it from the browser. Type in this URL:

```
http://cgi.cs.byuh.edu/~aa999/sample
```

Notice that we are using `cgi.cs.byuh.edu` instead of `cs.byuh.edu`.

This should run your CGI program, giving you the same results as for `sample.html`.

4.4 Why CGI?

You may be asking why we would do CGI when we can do html and be satisfied. The answer is that we can create a customized web page, possibly different for every person that views the page. We can display information about their bank account balances, or information about the books they ordered. Programming gives us the chance to have dynamic content that can look different to everyone. Without programs we are restricted to static content, which looks the same to everyone.

5 Web Forms

Web forms provide the input to CGI programs. They consist of blanks into which the user can type information, and buttons or boxes of several types that can be checked or pressed. A web page can have any number of forms.

5.1 `<form>`

Each form starts with a `<form>` command and ends with a `</form>` command. Forms cannot be nested within one another. Here is a sample `<form>` command:

```
<form method=post action="bar.cgi">
```

5.2 <input>

Within the form, the most important item is the <input ...> item. These create the data entry areas, the check boxes, and the buttons that can be pressed to communicate with your CGI program.

```
<input type=button   name=x value="y">
<input type=checkbox name=x value="y">
<input type=file     name=x value="y">
<input type=hidden   name=x value="y">
<input type=image    name=x value="y">
<input type=password name=x value="y">
<input type=radio    name=x value="y">
<input type=reset    name=x value="y">
<input type=submit   name=x value="y">
<input type=text name=x value="y" size=20>
```

There are many resources on the web to show you examples of the <input> command. Each input will have a name and a value. The name and value are sent to your CGI program.

5.3 Example

In this example, there is a form with three visible inputs: nuts, bolts, and enter. When the user keys in values for nuts and bolts, and presses the enter button, a string of information is sent to the CGI program.

```
<form method=post action="bar.cgi">
<input type=hidden name=f value=1>
<input type=text
 name=nuts value="" size=20>
<input type=text
  name=bolts value="" size=20>
<input type=submit name=done value="enter">
</form>
```

If the user keys in the values 19 for nuts and 27 for bolts, and clicks on the enter button, the CGI program will receive a single line of standard input with exactly the following content:

```
f=1&nuts=19&bolts=27&done=enter
```

A regular expression can be used to recognize this line and extract the data from it. This will result in the value 19 being placed into the variable nuts and the value 27 being placed into the variable bolts.

```
chomp ( $line = <STDIN> );
$line=~/nuts=(\d+)&bolts=(\d+)&done=enter/;
$nuts = $1;
$bolts = $2;
```

An if statement can be used to identify and respond to any of several forms.

```
chomp ( $line = <STDIN> );
if ( $line eq "" ) { &sub1 }
if ( $line =~ /^f=1&/ ) { &sub2 }
if ( $line =~ /logout=/ ) { &sub3 }
```

6 Tables

When you want to align the elements of your web page into neat columns, it is handy to use a table. You can start a table using a <table> command, and end it using a </table> command. Within the table, there are rows, each of which starts with a <tr> (table row) command. It is not necessary to use a </tr> at the end of the row. Within the row are data items. Each is introduced by a <td> (table data) command. It is not necessary to use a </td> at the end of the data item. Here is a sample table.

```
<table>
  <tr>
    <td>upper left
    <td>upper right
  <tr>
    <td>lower left
    <td>lower right
</table>
```

You can nest tables inside one another.

7 Programming Hints

Now you know how to use the components that make up a CGI program. Let's take it from the other side, top down. How do we design a CGI program?

7.1 Interactive versus Non

This may come as a complete surprise, but it is a foundation of the way that CGI programs work. Your CGI program is non-interactive. By interactive, we mean that your program can ask questions and get answers in some sort of conversation with the user. Typically an interactive program has a sequence of steps like this:

```
print "Hardware Store\n";
print "Enter the number of nuts: ";
$nuts = <STDIN>;
print "Enter the number of bolts: ";
$bolts = <STDIN>;
print "Enter the number of clips: ";
$clips = <STDIN>;
$total = $nuts + $bolts + $clips
print "You entered $total things.\n";
exit;
```

In an interactive program, you ask a question, get an answer, ask another question, get another answer, and so on.

7.2 Interactive is Good

It is nice to have a conversation between the user and the machine. The path is chosen as a result of intermediate results with many opportunities for adjustment as time goes by.

We can compare a bird to an arrow. When a bird flies, it can constantly adjust its course depending on its surroundings. Maybe it sees a tasty insect, so it swoops toward it and has a snack. Maybe it goes around a branch, or alights on a wire. All these decisions can be made "on the fly."

An arrow also flies, but it makes no decisions. Once it has left the bow, it flies straight, influenced by the wind, gravity, and the direction and velocity it started with. No more decisions occur. Eventually the arrow hits something and stops. If the original direction and velocity are carefully chosen (or lucky) the arrow may hit the center of the target.

Compare an arrow being directed to the center of a target, and a bird to finding its way home to its nest. Which is easier? Which is more reliable? Mid-course corrections make the flight interactive. This is a good thing because it makes life much easier and results more reliable.

7.3 Interactive is Missing

Mid-course corrections are exactly what we do **not** have when working with CGI.

CGI is non-interactive. You do not ask any questions. At least not in the sense shown above. Instead you get one line of input when your program starts. After that you create whatever output you deem necessary. Then your program ends. No more input. One line. At the start. That's it.

This non-interactive business is pretty harsh.

But we can fake it.

7.4 The Chess Prodigy

You may have seen or heard about a person that can play chess with dozens of people at the same time. Imagine this scene. The chess prodigy is some young boy or girl with an amazing gift for playing the game. Around the room there are dozens of tables arranged in a large circle. Inside the circle stands the chess prodigy. On each table is a chess board, and behind each table is an opponent. The boards are set. The matches begin.

Each opponent makes an opening move. The young chess prodigy walks quickly from table to table around the circle making his reply. By the time he returns to the first player, it is time for his second move. He continues around the circle, playing dozens of games at once and winning most or all of them.

Question: must the prodigy actually remember any of the games that he plays, or can he simply make the best move in the context of the chess board he can see?

7.5 Simulating a Conversation

We can fake a conversation by making sure we get enough context in each input. The context will help us remember what we still need to do to arrive at

our eventual goal.

At the top of your CGI program, you read one line of input. Inspect that line to see what it says. Based on that result, you can take a different path within your program. Look at this simplified CGI version of the interactive dialog above.

```
$reply = <STDIN>;
if ( $reply eq "" ) {
  print "Hardware Store\n";
  print "Enter the number of nuts: ";
  exit }
if ($reply =~ /nuts=(\d*)/) { $nuts = $1;
  print "Enter the number of bolts: ";
  exit }
if ($reply =~ /bolts=(\d*)/) { $bolts = $1;
  print "Enter the number of clips: ";
  exit }
if ($reply =~ /clips=(\d*)/) { $clips = $1;
  $total = $nuts + $bolts + $clips
  print "You entered $total things.\n";
  exit }
```

Instead of just receiving the number of nuts, we receive `nuts=23`, indicating both the number of nuts and the fact that it is nuts we are talking about. We record the result and ask the next question. Then our program ends.

The trick here is that like the chess prodigy, our program does not need to remember what steps it has completed. The incoming information provides that clue. Based on the incoming information we know what to do next.

7.6 Correcting the Lies

Every simplified thing is, to some extent, a lie. It does not tell the truth, the whole truth, and nothing but the truth. Instead, it tells the partial truth. It leaves out the whole truth. We simplify for educational purposes. Later we add the missing details.

We crafted our simulated conversation to illustrate an interactive appearance even when the underlying program was non-interactive. We said that context would help us do the right thing next. It does.

We skimped a bit on the details. There are two major issues that we did not address. First, do we have to make the user type `nuts=5` instead of simply typing `5`? Second (and more tricky), how does our program remember the value for nuts and bolts when it comes time to print the total? Isn't that information lost between separate runnings of the program? We will handle each of this problems in the following sections.

7.7 Providing Context

The `nuts=` context problem is easiest to solve, so we will handle it first. Recall this program fragment.

```
$reply = <STDIN>;
if ( $reply eq "" ) {
  print "Hardware Store\n";
  print "Enter the number of nuts: ";
  exit }
```

We must improve the program fragment by providing a bit more information.

```
$reply = <STDIN>;
if ( $reply eq "" ) {
  print "Content-type: text/html\n\n";
  print "<html><head>";
  print "<title>Hardware Store</title>\n";
  print "</head><body>";
  print "<h1>Hardware Store</h1>\n";
  print "<form method=post>\n";
  print "Enter the number of nuts: ";
  print "<input type=text size=20";
  print "  name=nuts value=\"\">";
  print "</form></body></html>\n";
  exit }
```

The `input` line specifies the name (`nuts`) and provides space (`size=20`) for the answer to be keyed in by the user. Because our `form` does not specify a different `action` (a program to be run), our program will be run again once the user types in a value and presses `enter`.

7.8 Carrying Forward Information

The second issue was how our program can remember the value for nuts and bolts when it comes time

to print the total. In fact all information is lost between separate runnings of the program. The best way is often provided by the use of hidden fields.

A hidden field is a special `input` field that actually has no input. It has a name and a value, but there is no way for the user to change it, or even see it (except by viewing the page source).

Here is an example for the nuts, bolts, and clips task:

```
...
print "<form method=post>\n";
print "<input type=text size=20";
print "  name=bolts value=\"\">";
print "<input type=hidden";
print "  name=nuts value=$nuts>";
print "</form>\n";
...
```

This technique lets us carry forward information from step to step. It does create one small problem in this case, though. We end up with the input line for bolts looking like this:

```
bolts=7&nuts=5
```

Without any other changes to our program, it will be recognized by the `/nuts=/` test, and we will repeat the bolts question. Here is another small change that could help out:

```
if ( $reply =~ /q=1/ ) {
  $reply =~ /nuts=(\d*); $nuts = $1;
  ...
  print "<input type=hidden";
  print " name=q value=2>";
  print "<input type=hidden";
  print "  name=nuts value=$nuts>";
  print "<input type=text size=20";
  print "  name=bolts value=\"\">";
  print "</form>\n";
  ...
```

In this case, we are using a hidden field `q` that tells us which question we are asking. `q=1` appears in the nuts question, `q=2` appears in the bolts question, and so on.

7.9 Defeating Hackers

A clever programmer will notice immediately that the data passed in hidden fields can actually be modified (with some difficulty) and returned to your program with different values. For example, if the user's credit limit was passed along from screen to screen, the user could actually hack the web page to change his credit limit and your CGI program would believe it.

In a case like this, it is a good idea to make up a random-looking, unpredictable key and use it to store the sensitive data in a database. Then pass the key in a hidden field. When the key comes back, look in the database for everything else. This prevents the hacker from changing his credit limit because it is not directly available. The key itself is hack-resistant because a modified key is not likely to match the database. This prevents the hacker from accessing someone else's data.

This technique has some side benefits as well. It can be used to carry forward a large amount of information without transmitting it back and forth. Also it can be used to avoid transmitting information that may be sensitive, such as ID numbers and passwords.

One could create a random-looking key by using the `rand()` function in Perl, and then verifying that the key is not already in use before using it.

8 Summary

CGI programming is fun and interesting. It gives us the ability to let others run our programs from anywhere in the Internet. It lets us carry out eCommerce. We have reviewed HTML, including forms and tables. We have learned how to structure a non-interactive CGI program so that it simulates a conversation, working in multiple steps. We have learned how to maintain data from one screen to another. We have looked at an important way to prevent hackers from misleading our programs.

We have scratched the surface pretty deeply, but there are still other features of CGI that we have not even begun to address, such as Cookies and SSL (secure sockets layer). But this should be enough to give you a good start.

Appendix

Validator

The `w3.org` consortium provides a free web page validation service. Go to their web site at

`http://validator.w3.org`

and type in the URI of your web page. It will tell you if your page has the correct syntax or not, and how to fix it.