# Table of Contents

**Introduction**............................................................................................................................................................... 1
  - How the Web Really Works ........................................................................................................................................... 1
  - GET and POST ............................................................................................................................................................ 1

**Perl and CGI**............................................................................................................................................................... 2
  - Here Document Quoting ................................................................................................................................................ 2
  - The CGI.pm Module ...................................................................................................................................................... 2
  - Returning a Web Page .................................................................................................................................................. 3
  - Sending Mail ................................................................................................................................................................. 4

**Maintaining State**....................................................................................................................................................... 5
  - Hidden Fields ............................................................................................................................................................... 5
  - Persistent HTTP Cookies ............................................................................................................................................. 5

**Security!** ..................................................................................................................................................................... 6

**For More Information** ................................................................................................................................................. 7
  - CPAN .......................................................................................................................................................................... 7
  - Web Resources ............................................................................................................................................................ 7
Introduction

The Common Gateway Interface (CGI) is not a programming language. Rather, it is a simple standard governing how a Web server interacts with and runs scripts to process forms and complete Web requests. Any program in any language that outputs a Web page can be a CGI program. Perl is particularly well suited to CGI because of its powerful text processing capabilities and its simple memory management.

How the Web Really Works

All actions on a Web server are initiated when a user somewhere on the Internet sends a request to the server through his or her Web browser. The protocol that the browser uses to communicate with the server is called HTTP, HyperText Transfer Protocol. The Web server receives the request and logs it. The request contains a variety of information, including what host the request is coming from, what browser is making the request, and what page is being requested. Normally, the request refers to a static file on the server machine, and the HTTP server retrieves the file and sends it back to the user.

However, CGI allows for more than just static pages to be served. To the remote Web browser, all URLs look alike, but on the server side, some of them point to static pages and some of them point to scripts. When a request for a script is processed, the server runs the script and passes it all of the information that the server collected. This information can also include input the user entered into a fill-out form. The CGI script does whatever processing its author wishes, which may include file I/O, database access, or even making other network requests. Once it is done, it outputs a Web page.

Sometimes, outputting the Web page is the main purpose of the script. For example, the online shuttle schedule at http://www.hcs.harvard.edu/cgi-bin/shuttle does not take any input from a form, or do any file processing. It simply constructs a Web page from its own internal shuttle schedule tables, and this page is presented to the user. This is still a dynamic page because the page content varies based on the time of day. On the other hand, some CGI programs return a Web page almost as a formality. For instance, after filling out the forms to enter a new event into the Harvard College Online Calendar (http://www.college.harvard.edu), a page is returned that says, “Thank you for submitting your event; it will appear on the College Web pages within 15 minutes.” If that were all it did, it wouldn’t be a very useful script. Instead, its major purpose is the database access that takes place between the time the request is sent and the page is returned.

GET and POST

An HTML form always has the <FORM> tag containing the METHOD and ACTION parameters. ACTION is the URL of the CGI script that will process the form. METHOD can be one of two values, GET or POST. These govern how the browser submits the data to the Web server, and how your CGI script receives the data. GET appends the data to the URL that is requested. This is useful because it allows users to bookmark the page, complete with form information. On the other hand, the length of the URL string is limited. For any larger form with more than one or two fields, you should use the POST method, which passes the data in a separate stream. The CGI.pm module understands both, and will automatically retrieve the data from the right location.
Perl and CGI

Here Document Quoting
Perl's text processing and painless memory management features make it ideal for CGI scripts. One feature that is frequently used is a special form of quoting, called “here document” quoting, that allows you to quote a whole HTML page. Normally, you would have to write:

```perl
print "<HTML>
print "<HEAD>
print "<TITLE>$my_title</TITLE>
print "<BODY>
print "<H1>$my_title</H1>
print "Hello, $name! Please check out the Web page for
print "</BODY>
print "</HEAD>
print "</HTML>
```

Putting in all of these print statements and making sure to escape all double quotes in the HTML tags can be quite cumbersome and makes the page hard to read. Instead, you can quote a multi-line string like this:

```perl
print <<"EOF";
<HTML>
<HEAD>
<TITLE>$my_title</TITLE>
</HEAD>
<BODY>
<H1>$my_title</H1>
Welcome, $name! Please check out the Web page for
</BODY>
</HTML>
EOF
```

You can put any string between the double quotes after the print statement; Perl quotes what follows until it sees the string you gave by itself on a line. Be sure you do not forget the semicolon on the print line. That line is the Perl statement, and requires a semicolon to end it just as any other statement does. If the string is in double quotes, then Perl will interpolate variables into the string just as with any other double-quoted string.

The CGI.pm Module
If this CGI script was launched because a form was submitted by the remote user, the Web server encodes the form data in a special way according to the CGI specification, known as url-encoding. To retrieve the information from the form, you can call upon a fairly standard Perl 5 module, CGI.pm. You load the module with the use command, which works like C's #include. Perl 5 modules are object-oriented, so you need to create one CGI object, which you can then query using the param function.

For example, say that you are writing a script to accept a form with two text fields, name and email:

```perl
#!/usr/local/bin/perl
use CGI; # load the CGI module
$query = new CGI; # create an instance

$name = $query->param('name'); # retrieve the name field
$email = $query->param('email'); # retrieve the email field
```
After doing this, you can use the information you have collected as usual, culminating in outputting a Web page. This module can do a lot more; for the complete documentation on the CGI.pm library, go to:

http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html

Returning a Web Page

An HTTP request is formatted very much like an e-mail message: it has a series of headers, one per line, followed by a blank line and then the HTML document, GIF or JPEG image, or other file. For the incoming request, the server removes the headers and places them into environment variables for you. When sending a request back, the server automatically adds almost all of the necessary headers, such as the date and the type of Web server.

However, you need to give one piece of information that the server does not know: the type of file you are sending. CGI scripts almost always return an HTML document, which is of type `text/html`. So, your program should output its page like this:

```plaintext
print "Content-Type: text/html\n\n";
print "<HTML>\n<head>\n"; # etc.

# or, like this:
print "<\nEOF";
Content-Type: text/html

</HTML>
</HEAD>
etc.
EOF
```

Be sure that there is always a blank line (two newlines) before the actual document.

There is one fairly common alternative to returning an HTML page. If your script's primary function is to process form data and the page it returns does not need to be customized, you can return a redirection rather than a document. To do this, simply write:

```plaintext
print "Location: http://whatever.url/you/want.html\n\n";
```

Again, be sure to have two newlines. The browser will receive this information and then go request the page you are pointing to, displaying that page rather than a page from your script. The URL does not have to be on the same server as the CGI; this redirection technique is sometimes used for navigational pages, where a user perhaps selects a page from a drop-down box and then submits the form. A CGI reads in the form input and redirects the browser to the appropriate remote page.
Sending Mail

While this is not CGI-specific, it is commonplace enough in CGI scripts that it is useful to include here. Here’s how to send mail from any Perl script (CGI or not):

```perl
open(MAIL, "| /usr/sbin/sendmail -t") || &error("Unable to fork.\n");
print << "EOF";
To: recipient@host.domain
From: your@email.address
Subject: You have a great Web page!

This is an e-mail message.
EOF
close(MAIL) || &error("Unable to send mail to recipient@host.domain.\n");
```

Note that you should check what the path is to the sendmail program. Sometimes just writing `sendmail` with no path will work, but many times it won’t and you will be puzzled as to why your CGI isn’t working. You can find it out by typing `which sendmail` at the command prompt. On both hcs and fas `/usr/sbin/sendmail` will work.

In a non-CGI script, you might write

```perl
open(FILE, "file.txt") || die("Can’t open file.txt.\n");
```

However, in a CGI your script must return a Web page; printing the error message to STDOUT doesn’t do much good. Therefore, you should create a short error function that displays a Web page explaining the error.

In case you are interested: the reason that there are two possible calls to the error function is because of the way that writing to a pipe works. When opening a file normally, Perl knows whether the open() succeeded when it computes the return value of open(). But when you open a pipe to write to or read from a program, Perl really does not invoke the sendmail program until after the open() call has completed. Therefore, if there are any errors running sendmail, they will manifest themselves by having the return value of close() be 0. Open() is 0 if Perl is not able to fork (that is, create a child process), usually because of critically low system resources. This is not a very important distinction for writing scripts, however.
Maintaining State

HTTP is a stateless system. To the Web server and to the CGI scripts, every request is completely independent of every other one. While the server logs everything, it does not keep track of a user’s session history. In addition, a CGI script handles one and only one request before exiting. The script is started when the request comes in, and when it finishes outputting its HTML file, it dies.

This means that when designing a more complex set of scripts, it is important to look at the problem from the perspective of the CGI script itself. When it is first run it has no knowledge of what has come before. Everything it learns comes from the server, from information encoded in the form, or from other parts of the system such as files and databases. If a script wants to leave any information for future versions of itself or for other scripts, it cannot talk to them directly. Instead, it needs to use an intermediary. There are three ways to do this: HIDDEN fields in a form, server-side files or databases, and client-side cookies.

Server-side files or databases are used if the data the user enters should persist for a long time, such as days or months. For very short-term use, however, such as keeping information from page to page during one session with the user, the overhead involved is usually too high.

Hidden Fields
If you are presenting a series of forms to the user, you can store the data entered in earlier forms as HIDDEN fields in the later forms. Just add fields to the form with type HIDDEN instead of type TEXT. For example:

```html
<FORM ACTION="..." METHOD="POST">
  <INPUT TYPE="HIDDEN" NAME="lastname" VALUE="Alpert">
  <INPUT TYPE="HIDDEN" NAME="firstname" VALUE="David">
</FORM>
```

These fields will be transmitted just as other fields in the form, but the user will not see them in the form unless he/she looks at the source. Note that the values of the fields are in the page source, so you should not store secret information such as passwords this way.

Persistent HTTP Cookies
Cookies are a special feature of most modern browsers that allow a Web server to store strings in a special file, cookies.txt, on the user’s computer. A site can only see the cookies that it has stored, and cannot access any other information on the user’s hard drive. Cookies persist for a fairly long time (at least a month) before being expired, so a site can store information that will be available when the user returns.

Many sites that require registration, like the New York Times (http://www.nytimes.com), store the user’s password as a cookie so that he or she does not have to type it upon each visit. Some more creative uses of cookies, such as Netscape’s PowerStart (http://search.netscape.com/custom/index.html), include allowing a user to create his or her own personalized view of a site; when the user visits, CGI scripts will build the user’s personalized page by looking at the stored cookie information.

Cookies have been relatively controversial, since some advertising firms have begun to track a user’s visits across many different Web sites and compile detailed demographic information, which is a privacy concern. Therefore, newer browsers allow users to be warned when a site submits cookies, or to reject them altogether. Combined with the fact that some browsers do not support them at all, you cannot rely on cookies, but they can be a useful feature to add more convenience to a Web site.

The CGI.pm module contains functions to help you use cookies; refer to the CGI.pm documentation at http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html. For more information about cookies, check out http://www.cookiecentral.com/.
Security!

It is important to treat all input from the user as totally untrusted. It is very easy to assume that all form input will be what you expect, and design a script around that basis, but then leave large security holes. For example, the sendmail function can be called with the address on the command line, like this:

```perl
open (MAIL, " | sendmail perl@hcs.harvard.edu");
```

Imagine a form that allowed someone to send mail – say, an electronic postcard – to any of their friends. You might imagine the CGI script reading in the address into scalar $recipient and then sending a message like this:

```perl
open (MAIL, " | sendmail $recipient");
print MAIL "\"EOF\";
Hello!

You have an electronic postcard from $sender.
You can pick it up by going to http://www.hcs.harvard.edu/postcards.

Thank you,
The Imaginary HCS Postcard Project
EOF
```

But just imagine what would happen if a malicious user entered perl@hcs.harvard.edu; rm -rf * into the “Recipient” field of the form. Then the command passed to Unix would read:

```perl
sendmail perl@hcs.harvard.edu; rm -rf *
```

which would execute two commands, one to send a mail message and another to wreak havoc on the system. So, one important thing to remember is never to pass user input to the command line. In general, watch out for metacharacters like closing quotation marks, semicolons, and other characters. If you are using the input in a place where metacharacters could cause undesirable side effects, be sure to pass the input through a function that removes or escapes them.

This does not only apply to free-text fields. Imagine a CGI script to send mail to different HCS project leaders, for example. This hypothetical form has a drop-down selection box with several different e-mail addresses. Even though the user’s choices are constrained on the form, this would still be just as much of a security risk if the input was not checked. That is because a malicious user could copy the source to the form and build an identical form, but with the selection box changed to a text field. Your CGI has no way to tell what form sent it the data; do not assume that the user necessarily used the form that you intended.

Therefore, also keep in mind that just because you saved some information into a hidden field, it does not necessarily mean the same information will come back; just because you have a set of radio buttons does not mean that the user could substitute some other information. It is important to error check in any case.
For More Information

CPAN
There are hundreds of freely available Perl modules that can do almost anything you might need. To find them, go to http://www.perl.com/CPAN/, the Comprehensive Perl Archive Network.

The DBI and DBD modules create a consistent interface to many different databases, and greatly simplify database access. Another useful module is LWP, which provides routines for you to write a Web client. This module allows you to retrieve Web pages from within a Perl program, and do almost anything a Web browser can do.

Web Resources
There is a large wealth of resources relating to Web publishing and CGI on the Web. Here are a few:

- Webreference.com: http://www.webreference.com
- Yahoo WWW Index: http://www.yahoo.com
- Matt's Script Archive: http://www.worldwidemart.com/scripts
- CGI.pm Documentation: http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html
- Cookie Central: http://www.cookiecentral.com/
- Comprehensive Perl Archive Network: http://www.perl.com/CPAN/