by Peter Collinson, **Hillside Systems**

PAUL SCHULENBURG

Common Gateway Interface

This time last year, I came out as a Perl programmer with an article that gave you the basics of Perl ("Getting Started with Perl," October 1998, Page 22, `http://sw.expert.com/C2/SE.C2.OCT.98.pdf`). I started to learn Perl because I wanted to create interactive forms for the Web. Scripts supporting the Web are often called CGI scripts because they use a standard interface supposedly supported by all Web servers, called the Common Gateway Interface. (Just in case you are a little thrown by the "supposedly" in the previous sentence, I haven't come across a Web server yet that doesn't support CGI. I also haven't looked at all the Web servers in the world.) CGI defines an environment in which any process is run by the Web server. A process run by the server can be written in any language. We talk loosely about "CGI scripts" because it's usually more convenient to write them in a scripting language. CGI dictates the inputs to any process that the server runs and proscribes its output. This article examines the CGI interface and how scripts fit into the system.

Before looking at how CGI scripts work, we need to look a little at HTTP, the Hypertext Transfer Protocol, which is the fundamental method of communication over the Web. The job of a Web server is to respond to an HTTP request from a client. If the request is for a file, perhaps holding HTML, an image, movie or sound, then the server will find the file and send it.

HTTP is a text protocol, messages consist of an initial line, which is either a request from a browser or a response line from the server, followed by more data. The "more data" may be a file, and the type of information in that file is specified using the conventions adopted for MIME. MIME stands for Multimedia Internet Mail Extension and started out as way of sending more than just text via electronic mail. The protocol was picked up as a convenient technology to manage file typing by HTTP. Unlike many Internet protocols, HTTP is fairly bidirectional. For example, most of the time, the message contains a file that is sent by the server to the client, and MIME is used to find out how the file should be displayed. However, the client can also send a file to the server using the same protocol elements.

Many Web servers are set up so that when an HTTP request from the browser asks for a file with the suffix `.cgi`, then the server will not send the file. Instead it will expect to find an executable program and will run it. The job of the program is to generate appropriate output. Web servers can also be configured so that a request for a file in a particular directory, or directory tree, will be executed as a script. This special directory is often called `cgi-bin`.

Web servers have adopted the "special directory approach" for security reasons. It's dangerous to allow Joe Random-User to run programs on your

machine, even if they are your programs, so most Web servers place constraints on CGI scripts that are intended to promote safety. It's good practice to hide scripts so that they may be executed but not downloaded. I generally place my scripts outside the tree of files that can be downloaded by a browser.

Of course, hiding the script contents is an example of "security by obscurity," which is generally a poor foundation for real security. However, it does prevent Joe's close friend, Arnold Bad-Guy, from examining the scripts in order to find security holes.

When writing a CGI script, we are creating a program that is to be run as a subprocess by the Web server. We need to find a way of getting information from the server into the running script. There are a finite number of ways that are used to supply a subprocess with data. The first that springs to mind is the use of program arguments. Actually, CGI scripts don't use program arguments, probably because the solution was considered "too UNIX," or because the standards that have been set for program argument styles are too restrictive.

When writing a CGI script, we are creating a program that is to be run as a subprocess by the Web server.

The second method of passing data from a process to a child process is via the "environment," a set of *name=value* pairs that are inherited by the child from its parent when the child is created. UNIX usually uses the environment for storing useful pieces of global context information that processes may need to know. The *name=value* structure of the environment is a good match for the data that comes from the form that the user has filled in using their browser. Each input box or input feature (like a checkbox) on a form is given a name and will be given some value by the user that needs to be tied to this name and sent to the Web server.

As a final method of sending data into a child, we can pass the subprocess an open file and ask it to read data from that file descriptor. This technique is used commonly in shell redirection. For example,

```
$ ls -l /bin | more
```

will cause the shell to run the `ls` command and divert its output into standard input of the `more` command. The `more` command is passed an open file descriptor and reads the data; it's actually oblivious to the source of the data it is processing. As we shall see, passing an open file is used by CGI scripts in some circumstances.

The Protocol

When the client sends a request to the server that results in the execution of a script, the server will preload several environment variables that the running CGI script will inherit and can later inspect. CGI specifies the names of several of these variables as "standard" parts of the CGI protocol. Your server may provide other environment variables. I often run a simple shell CGI script containing the shell's `set` command to dump out the environment variables that are provided by the server, usually to check that I know the correct name of a particular variable (see below).

I am not going to list the variables here, but I will talk about them generally. The information splits into three categories. First, there are variables that tell the script something about the server being used to run the script. For example, the script can find out the port number the server is using. This may be useful if you have two parallel servers using the same source tree, but one is a secure server and the other uses normal nonencrypted communication. Your CGI script is able to test the port number and do different things for users of each server.

Second, there are some variables that give the script basic information about the browser and system that made the request for the data. So, for example, the script can identify the remote IP address or the type of browser. It cannot identify the person at the other end, nor their email address.

Third, there are variables that contain information about the current transaction. As I said, HTTP uses the MIME protocol to exchange typed information about data blocks, and the MIME type of the incoming data is made available to the script in the `CONTENT_TYPE` variable.

The designer of the form that calls the CGI script has two choices, `GET` and `POST`, that define the way information from the form is passed to server. The `METHOD=` parameter specifies the choice to the `FORM` HTML element placed on the Web page. Each method uses a different way to send the user's form information to the server when the submit button is pressed.

The names `GET` and `POST` refer to commands in the HTTP protocol. When a browser asks for a regular page it sends a request like the following:

```
GET /main.html HTTP/1.1
```

The above request is asking the server for the page `/main.html` and indicating that it's able to cope with Version 1.1 of the HTTP protocol. If all is well, the server will respond with a message containing the appropriate data.

When the browser is told to send data from a form using a `GET` method request, it will send a `GET` request and will add a question mark along with the form contents to the end of the URL in the `GET` statement. It creates an extended URL containing the address of the page and the parameters to be sent to the script. If there are a great many *name=value* pairs on the form, or the values are large, then this initial request to the server could be larger than it can comfortably deal with. Also, when the URL is processed by the server, any string following a question mark ends up in an environment variable (`QUERY_STRING`) that is made available to the CGI script. It's

desirable to limit the space taken up by environment variables.

So for larger forms, the `POST` method is preferred. Here, the data from the form is not sent as part of the URL, but follows the initial request line as a MIME-encoded file in the body of the message that's sent to the server. Because uploaded data is MIME-encoded, the server knows the size and the type of data and will pass these values into the script using the environment variables: `CONTENT_LENGTH` and `CONTENT_TYPE`. In addition, the script expects to read the set of parameters sent by the form from its standard input channel. It's the server's task to ensure the data that's been sent from the browser is made available to the script ready to be input and scanned for *name=value* pairs.

There's one further complication to both the `GET` and `POST` methods. I mentioned that HTTP is a text protocol, and using text to convey information has its own problems. It's often the case that in any text application we have to "steal" characters to use in the application itself. Think about the UNIX password file: it uses the colon character to separate fields, and this means we can't have a login name that contains a colon. The colon is a "stolen" character.

HTTP uses an encoding method to ensure data is passed intact. In a ***POST*** or a ***GET*** message, the named input boxes on the form are sent using this method.

To get around this, HTTP uses an encoding method to ensure data is passed intact. To encode the data for use in URLs, character spaces are replaced by plus signs (+) and any nonalphanumeric character is replaced by a percent sign (%), followed by the two-digit hexadecimal representation of the character. You'll sometimes see this when people have home pages related to their account name that are accessed by the UNIX convention of tilde (~) followed by their user name. Their URL might be

```
http://www.domain.com/~pc
```

but is sometimes written in encoded form as

```
http://www.domain.com/%7Epc
```

Actually, most encoding routines pass more than alphanumeric characters through transparently. They tend to also send underscore, backslash, minus and period untranslated, so URLs contain the characters we expect to see.

In a `POST` or a `GET` message, the named input boxes on the form are sent using this encoding method. The names and values are encoded, and each pair is made into a string of the form *name=value*, these strings are then concatenated into one string, where each pair is separated by the ampersand character (&). For example, a `POST` method form that contains two variables `NAME` and `PHONE` will send its data as

```
NAME=Joe+Random&PHONE=890+8394
```

The `GET` method will add this string to the URL, after a question mark. The `POST` method will transmit this string as part of the HTTP protocol, and the script will read it from its standard input channel.

Output from the Script

The job of the Web server is to send a response to the user's request. CGI scripts are responsible for sending back a file of data to the client. The Web server arranges things so the standard output of the script will send data to the client's browser. However, the output needs to be MIME-encoded and a script needs to start by sending a MIME header that specifies the type of data that is to follow. The MIME header is separated from its data by a blank line, so if you look at Perl CGI scripts that send HTML, you'll often see them start with

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

which specifies that the data that follows is HTML. Notice that there is an extra blank line output by this statement. The header of the message is separated from the body of the message by a blank line. Having output the header, the script can now happily send further HTML statements.

I talked earlier about writing a CGI script that will dump out all the environment variables set by the server. We can now do that; it's a shell CGI script:

```
#!/bin/sh
echo 'Content-type: text/plain'
echo
set
```

You need to place this script in your `cgi-bin` and name it, say `shset`. The script is executed by the shell. We are using the standard `#!` magic character pair at the start of the file to tell the system that this is an executable file. To work, the script needs to have execute permission set:

```
chmod +x shset
```

The script prints the MIME header line, a blank line and then uses the standard shell command to dump the values of the environment. You can now call this from a browser by typing a URL like the following:

```
http://www.domain/cgi-bin/shset
```

You'll get a screenful of information that shows you the various values that are established for CGI scripts by your server.

Because the CGI script is actually supplying part of the HTTP protocol that is sent back to the browser, it's easy to

make it use extra capabilities of the HTTP protocol. For example, when I create pages that say "Thank you for submitting that data," I generally tell the browser to wait for 10 seconds and then head off to some other location, perhaps the home page of the server or the page the user was viewing before they began filling in the form. Doing this is easy, you add a line like the following into the header:

```
Refresh: 10; URL=//http.domain/index.html
```

I generally output this before the `Content-type` statement. The URL needs to be absolute for this to work.

Decoding CGI Arguments

I don't have the space here to go into the nitty-gritty of decoding URL arguments. If you look at the books mentioned at the end of this article, you'll find some example scripts that will do it for you. Perl now has a standard module that can do all the hard work, called `CGI.pm`. I don't use this personally, for a number of reasons.

I find that it's actually very hard to get a handle on how to use the `CGI.pm` library for simple applications. Ideally, I need to see and understand some working code, and the examples I seek are not readily available, or at least, I haven't found them yet. If I cannot gain confidence with a system for simple applications, then I am unlikely to graduate to using it for complex ones.

I get very queasy about using libraries to do things for me because although libraries often make the task easier, they inevitably hide the nastiness of the task in hand. So when you use a library, you probably don't have to know what problems it is trying to solve. This is fine when it works. But when it fails, your debugging job is harder because you suddenly have to read and understand someone else's code. Worse, that someone else is likely to be an expert programmer using Perl constructs that are not exactly blindingly obvious. (Please note I am talking generally here, and definitely not pointing the fickle finger of hate at the author of `CGI.pm`, whose code I have not examined).

Finally, if you use a library, then you need to buy into the way that the author thinks you should do things. In the case of `CGI.pm`, this means that you write a program that contains both the code to generate the look of the form and the same code that understands the data, checks it for veracity and performs the main action of the form. I feel the approach leads to too much stuff in the script, and the separation between the "look of the page" and the "action of the page" isn't sufficiently clear-cut.

Please don't send me hate mail complaining that the CGI library is perfectly OK for you and it's easy to use. I am sure that this is correct; mileage varies.

Actually, to read form arguments you only need around 15 lines of Perl, and you can find those lines in all of the books I refer to below. The first routine decodes the URL format, changing the hexadecimal encoded values into "real" characters. The second routine deals with `POST`ed arguments. It reads several bytes from its standard input channel where the Web server places the `POST` values. The routine knows how many bytes to read because it can examine the `CONTENT_LENGTH` variable in the environment using Perl's standard `ENV` associative array.

My routine for decoding `POST` input was acquired from a book (undoubtedly one in the list below) at some time in the past. It first reads all the data into a string and then splits this string into an array of strings using the ampersand character as a separator. It now has several *name=value* strings that it needs to process further. It splits each of the parameter strings into two separate variables at the equals character (=) and applies the decode URL routine to both variables. Finally, it stores the argument into an associative array, so I can access the value of a box on the form using some code like the following:

```
if ($FORM{NAME} eq "")
```

If you are bewildered by all this talk of associative arrays, don't be. An array in a programming language is an object that can hold several values, and we normally access those values using a numeric index. An associative array is a similar object, a single variable that can hold several values, but the values are accessible by using a name or text string as the index. Associative arrays are well suited to storing form parameters because they allow us to access the value by using the name part of the *name=value* pairs. In Perl, we tell the language we are using an associative array by using the curly-brace syntax. So the `$FORM{NAME}` statement above says: `FORM` is an associative array and we want to look up the value that's associated with the string `NAME`. The dollar symbol (`$`) at the start tells Perl that we want to treat the result as a single entity, in this case a string, which we test against the null string.

Further Reading

Until recently there was not much information readily available about HTTP and its workings. I used to surf over to the World Wide Web Consortium (W3C) site at `www.w3.org` to look at the standards documentation. I still do this from time to time. Another source of basic information I use is *Managing Internet Information Services* by Cricket Liu, Jerry Peek, Russ Jones and Adrian Nye (published by O'Reilly & Associates Inc., 1994, ISBN 1-56592-051-1). This book is quite old now and some of its information is a little dated. I am also using a recently updated publication by O'Reilly & Associates entitled *Webmaster in a Nutshell*, 2nd Edition (by Stephen Spainhour and Robert Eckstein, 1999, ISBN 1-56592-325-1), which contains a comprehensive and concise description of HTTP and also has some CGI programming information. There are a great many books with samples of CGI programming.

The *SW Expert* editing team point out a good starting point on the Web is `http://www.ncsa.uiuc.edu`, the information is somewhat dated but otherwise OK.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.