CHAPTER 2

Troubleshooting CGI Scripts

OVERVIEW

Web servers and their CGI environment can be set up in a variety of ways. Chapter 1 covered the basics of the installation and configuration of scripts. However, you may encounter situations in which the scripts still do not work. After all, no two servers work the same way, because their system administrators have configured them differently. Thus, what works on one server may not necessarily work on another. The tips and tricks outlined here will help you to get past many of the problems you might encounter.

The first few sections of this chapter will focus on the most common problems that can arise in the course of using scripts. The later sections will discuss some complex but useful troubleshooting tools to add to your repertoire of CGI setup skills.

Server Errors

Whenever the Web server responds to a request from a browser for a document or CGI program, a status code is returned. This status code can tell you a great deal about the sort of problem you are encountering. Table 2.1 contains a list of the possible status codes a Web server might return. The codes serve as a road map to errors that may be occurring in an incorrect script installation. Each error-related code serves as a first step in figuring out what is going wrong with a script's execution.

Table 2.1 *The major HTTP response codes.*

Code	Status	Meaning
2xx (Codes for a successful Processing)		
200	OK	Request was successful.
202	Accepted	Request is still processing but was successful.
204	No response	Request was processed, but there is no data to be sent to the client.
3xx (Codes for Redirection)		
301	Moved	Document was moved permanently to a new location.
302	Found	Document is at a different location on the server. Response typically includes a Location header to direct the browser to a new document.
304	Use Local Copy	Document was found but was not modified since last request.
4xx (Codes for Client Errors)		
400	Bad Request	Syntax of browser's request was not recognized.
401	Unauthorized	Server has restricted the document to be viewed. Typically, the document is protected by password or IP address and authentication has failed.
403	Forbidden	Request was forbidden, typically because of access rights.
404	Not Found	Server could not find the document.

CODE	STATUS	MEANING
5xx (Codes for Server Errors)		
500	Internal Error	Server was not able to carry out the request but does not know the explicit reason. A catch-all error.
501	Not Implemented	Sender knows about the requested service but does not know how to handle it.
502	Service Is Overloaded	Server cannot process more requests than it is currently processing.

204 (No Response)

This typical response is given if the script has executed successfully but has failed or exited before it could return any HTML data other than the "Content-type: text/html" header. The techniques for troubleshooting this status code are basically the same as those used to figure out a 500 (Internal Error) code, described later. However, one item to check out here is the permissions that exist on a particular directory or file.

The script may not have permission to create, write, and read certain data files or subdirectories that it expects to work on. When this happens, the script may work up to the point that it prints the "Content-type: text/html\n\n" message output by all CGI programs. But because a file is not found or cannot be written, the rest of the program logic may not be able to generate the dynamic HTML based on the contents of the data files. As a result, a "Document Contains No Data" or "No Response" error message may pop up. Make sure that any subdirectories required by the script have been created and that it has the appropriate level of access to them. For example, if a Web application needs to write files in a subdirectory such as **Sessions**, you need to give the Web server write access to the subdirectory. This access is discussed more thoroughly in Chapter 1.

401 (Unauthorized)

This code means that the script was placed in a protected area of the Web server and the authentication failed. The script may be protected by

allowing only certain IP addresses or Internet hosts to read the document, or it may be restricted by a username/password combination. The solution to this problem is to place the script in an unprotected area or figure out how to satisfy the security restrictions.

403 (Forbidden)

Typically, the 403 (Forbidden) error occurs when you try to access a script that the Web server does not have the permissions or rights to read and execute. This situation is easy to remedy. On UNIX, merely use the **chmod** command, as specified in Chapter 1, to make the script readable and executable to the Web server.

404 (Not Found)

404 (Not Found) is output when the Web server cannot find the document the browser has requested. This error is also easy to correct. Typically, it is caused by the user typing the URL incorrectly, or perhaps no mapping has been set up from the URL to the actual directory on the Web server. For example, on most Web servers the /**cgi-bin** URL is mapped to another directory structure on the server, such as /**usr/local/etc/httpd/cgi-bin**. If this mapping is not set up, the Web server may attempt to look for the script in a **cgi-bin** directory underneath the document root, which may be an incorrect location.

Since URL directory mappings may be entirely different in nature to the real directory path on the Unix server, you should be very careful of this error. Many virtual servers are setup with a completely different URL structure than what you see when you are logged into the actual UNIX server.

500 (Internal Error)

500 errors can be difficult to solve, because it is a catch-all error message. Anything else that can go wrong with a script that is not caught by the

previous errors is typically sent as a 500 error. Basically, a 500 server error is caused because the Perl program did not print "Content-type: text/html\n\n" before anything else was printed. This "magic header" must always print before anything else in the Perl program. Frequently, the cause of this failure is that the Perl program was not able to run at all. The following are some possible ways to resolve this problem.

First, the script may have syntax errors or some other problem that causes it not to run. An easy way to check for this kind of error is to run the script from the command line using Perl's `-c` parameter to check syntax. If **foobar.cgi** contained an error, you could run `perl -c` on it:

```
$perl -c foobar.cgi
literal @whatnot now requires a backslash at foobar.cgi line 5, within
string
foobar.cgi had compilation errors.
```

The output shows an error in line 5. Specifically, the programmer has placed an `@whatnot` inside a string in Perl. Perl version 5.x must have the `@` symbol escaped with a backslash (`\`) character.

A common cause of `@` symbols not being escaped is that an E-mail address has been entered into the Perl program without escaping the `@` symbol. For example, "selena@eff.org" is wrong for Perl version 5. You must use "selena\@eff.org."

To fix this error, you could inform the original author or go into the script and fix it yourself, depending on your level of expertise. Even if you can fix it yourself, it's considered good etiquette to inform the original programmer that you are encountering a syntax error on your system. In that way, the programmer can fix the problem for other people who may not be as clever at solving Perl problems.

Another cause of 500 errors is that the libraries called by the script may have errors or may not be found in the path. Any libraries, such as **cgi-lib.pl**, that have been downloaded with the script should be run with `perl -c` to check for syntax errors. If that does not reveal the problem, run the script itself from the command line. The following example

shows how you might run the script from the command line to see whether there is a problem with the `require` statement that loads libraries. For example, if **bbs_forum.cgi** is run from the command line and the required file **bbs.setup** is not present, you will see the following output. The error at the bottom tells you that **bbs_forum.cgi** tried to include **bbs.setup** but was not able to. The way to fix this error is to make sure that a file called **bbs.setup** is in the path where **bbs_forum.cgi** expects it.

```
$ bbs_forum.cgi
Content-type: text/html

Can't locate bbs.setup in @INC at bbs_forum.cgi line 99.
```

A similar problem is that the correct path to Perl is not set up in the first line of the script. For example, if the script has `#!/usr/local/bin/perl` in the first line of the file, perhaps Perl is not in the **/usr/local/bin** directory. Chapter 1 contains a section on finding out where Perl is located in your directory. Remember that this magic line must appear by itself on the first line of the script.

It is considered better to run the scripts through `perl -c` before you run them using `perl`. `perl -c` checks for syntax; `perl` actually runs the program. Running the program by itself may display strangely, because it is not running in the environment of the Web server.

If the server is running in a different environment from that of the script, certain utilities and commands may not be available to the Web server. Some Web servers run in a `chroot` environment where they lack basic utilities such as `ls` and `pwd`. If a CGI script relies on calling external commands, it can also cause the script to fail. In this case, you could attempt to copy the binaries of the external commands into the `chroot` directories that the Web server can see, or you may need to contact your Web server administrator to make these utilities available. Note that it is considered bad practice to call external programs when Perl can do the

job internally. You may also want to consider attempting to find a different script that does something similar but without relying on an external command.

If you are in a `chroot` environment, do *not* copy the Perl interpreter itself to any CGI directories. Doing so would have serious security consequences, because any user of your Web site would be able to call Perl directly from Netscape and pass it new Perl commands, telling it to do all sorts of destructive things such as `rm -rf *` to remove all files in all subdirectories. Yikes!

A related problem may be that the script calls an external program whose output displays before the "Content-type: text/html" header has been output. How can this happen? Essentially, Perl buffers data internally before sending it to `STDOUT` (standard output) with print statements. This arrangement can cause a problem when a system call is generated that outputs data and then ends. The system call, because it is completed, will tend to flush its own `STDOUT` before Perl has a chance to flush the statements it has previously buffered internally. The trick to solving this problem is to start the Perl script with the following piece of code:

```
$| = 1;
```

This code sets the current default file handle in Perl (`STDOUT`) to not buffer the output. There is a slight performance loss, because buffering exists to send the data in large blocks at a time, a faster technique than many little requests. However, this method will frequently solve a problem that occurs when `system()` calls in Perl output data before the Perl script has a chance to.

Another problem that you may run across is mismatched ASCII types of files on your server. This mismatch can happen when you download an archived set of scripts to a local machine running a non-UNIX operating system such as DOS or Windows, edit the files there, and then use FTP to transfer the files to your server again. DOS editors typically save files with a CR-LF (carriage return/line feed) combination of two characters (`\R\N`, in

Perl terminology). However, UNIX reads ASCII files with only an LF and no CR. If you edit a file on a DOS machine and FTP the file to the UNIX server, always remember to transfer it using the ASCII mode of the FTP program. This mode will perform the CR-LF to LF translation automatically.

Many Perl scripts may not have a problem with the added CR on UNIX. Generally, a Perl script will be most affected by the CR-LF problem if it uses the "here" method of assigning multiple lines of character output to a string. Basically, with the "here" document method, you define a boundary at which point the lines of character output stop, much as a quotation mark delimits a string on a single line. Unfortunately, if the boundary marker appears on a single line by itself and has a `CTRL-M (CR)` after it, Perl will not recognize it as the boundary marker because of the extra character. The moral of the story is that just because some Perl scripts may work with the method of file transfer you were using, it's not a good idea to assume that all of them will.

Another useful diagnostic tool is the error log of your Web server. If you are sharing a server of an Internet service provider, you may not have access to the Web server's error log. However, if you run your own Web server or if you use one that lets you see and access the error log, it can be an invaluable source of information. Typically, whenever an error occurs in a Perl script, it gets output to a standard error channel (`STDERR`). Anything output to `STDERR`, including syntax errors and runtime errors, typically gets written to the error log. Examining the error log can save you a great deal of time by telling you immediately what the Web server found wrong with the CGI program.

Output Errors

The last problem that may occur when you're setting up scripts is that the output is valid but unexpected. In other words, the script works but the output is wrong. This problem is a complex one to solve, because it generally indicates a logic error in the programming rather than a problem with the way the script was set up.

However, there are a couple of tricks you can use to solve this. First, it is possible, as with the 500 error, that subdirectories and files do not exist that the script expects to see and have read or write access to; or the permissions may be set incorrectly. It cannot be emphasized enough that many of the problems that arise in setting up scripts boil down to access permission problems.

Beyond this solution, there are a few tricks to helping solve logic problems in scripts if you feel up to modifying someone else's Perl code. It is possible to "fool" a script, from the command line, into thinking it is running in the server's environment. To do this is, set up key environment variables before running the script. If the method that is being tested is a `POST`, then you must also have a file that contains the post information that can be redirected to the script as `STDIN`.


WARNING

The trick just described is a more advanced technique that should be used by programmers who feel comfortable modifying Perl scripts. It is a good idea to back up the scripts to another directory in case you want to go back to the original version of one or more of your CGI scripts.

For the following examples of fooling a script into thinking it is getting form data passed, the form variable `name` will be set to `John Smith`, and the form variable `bdate` will be set to `12/2/30`. Because the form variables contain characters that a Web server has trouble recognizing, the variables must be converted to a urlencoded format. This means that the characters such as spaces must be converted to escape characters. The urlencoded form of the above variables is `name=John%20Smith&bdate=12%2f2%2f30`. If you are testing a script to use the `GET` method of passing form data, then the top of the Perl script must be modified to include the following environmental variable settings:

```
$ENV{"REQUEST_METHOD"} = "GET";
$ENV{"QUERY_STRING"} = "name=John%20Smith&bdate=12%2f2%2f30";
```

The script should now run and decode the data as form variables.

If you want to emulate the `POST` method of transferring form data, the process is slightly different. First, you create a file that contains the post data. For this example, we will assume that the value of the `QUERY_STRING`

environmental variable set previously will be placed in a file called **test.in**. Next, we determine the `CONTENT_LENGTH` of the file in bytes. In UNIX, the easiest way to do this is to use the `wc` command:

```
$ wc -c test.in
35
```

The `-c` parameter returns the character count of the file. In this case, the file is 35 characters long.

The following environmental variables must be set at the top of the Perl script:

```
$ENV{"REQUEST_METHOD"} = "POST";
$ENV{"CONTENT_LENGTH"} = "35";
```

Then call the script from the command line and also pass the **test.in** file to the script (**foobar.cgi**):

```
$ foobar.cgi <test.in
```

On the other side of the coin, instead of forcing different form variables at the beginning of the Perl script, you may want to edit the script to print the value of the form variables that have been passed by the Web server. For scripts that use **CGI-LIB.PL** to parse form variables, this is easy. Simply use the `&PrintVariables` routine and pass it the array that you used to read form variables (`%in`, by default) by reference. You would add the following line:

```
print &PrintVariables(*in);
```

In addition, you might want to print environmental variables. This is also easily done with **CGI-LIB.PL**. Merely use this code:

```
print &PrintEnv;
```

The techniques we've described will allow you to test various types of form input to the CGI script. Although some of them may be a little advanced, they are valuable techniques to add to your CGI debugging toolbox.

Conclusion

With the myriad of different ways a Web server can be set up, it should come as no surprise that troubleshooting CGI problems is common when attempting to set up scripts for the first time. The methods described here from tracking down server errors through working out a program's HTML output problems provide a step-by-step prescription for detecting the source of the most common problems you may encounter when trying to use CGI scripts.