How CGI Scripting Works

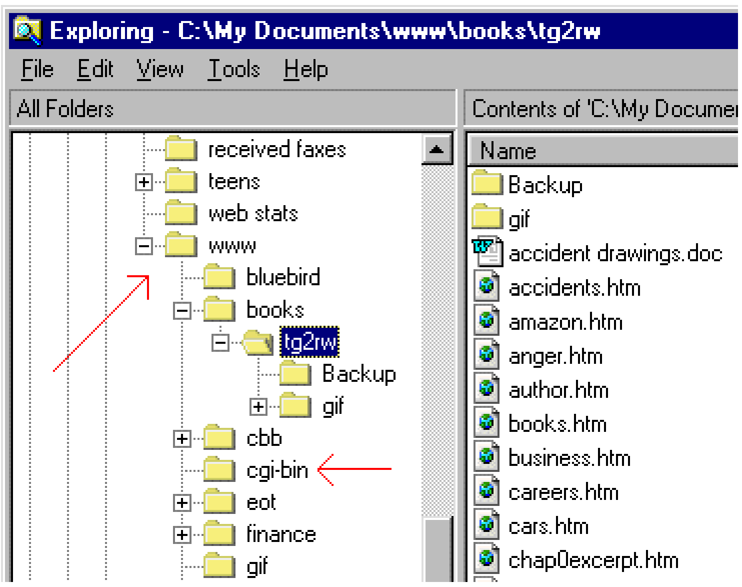
by Marshall Brain

The article How Web Pages Work discusses the basic features of HTML and shows you how to create Web pages that contain text and graphics. It also shows you how to get your page "on the air" with a hosting service. One of the questions frequently asked by new Web site designers once they get their site up is, "What is CGI Scripting and how can I use it on my site?" or, "How do I create interactive forms on my site?"

In this edition of **HowStuffWorks**, we will answer your questions about CGI scripting and show you how to create your own scripts. You'll also learn a bit about Web servers in the process. Let's get started!

Web Servers

As described in the article How Web Servers Work, Web servers can be pretty simple. At their most basic, Web servers simply retrieve a file off the disk and send it down the wire to the requesting browser. Let's say you type in the URL **http://www.bygpub.com/books/tg2rw/author.htm**. The server gets a request for the file **/books/tg2rw/author.htm**. If you look at the following figure, you can see how the server resolves that request:



During setup, the Web server has been instructed to understand that **c:\My Documents\www** is the server's root directory. It then looks for **/books/tg2rw/author.htm** off of that root. When you ask for the URL **http://www.bygpub.com/books/tg2rw/**, the server understands that you are looking for the default file for that directory. It looks for several different files names to try to find the default file: index.html, index.htm, default.html, default.htm. Depending on the server, it may look for others as well. So the server turns **http://www.bygpub.com/books/tg2rw/** into **http://www.bygpub.com/books/tg2rw/index.htm** and delivers that file. All other files must be specified by naming the files explicitly.

This is how all Web servers handle *static* files. Most Web servers also handle **dynamic files** -- through a mechanism called the **Common Gateway Interface**, or CGI. You have seen CGI in all sorts of places on the Web, although you may not have known it at the time. For example:

- Any guest book allows you to enter a message in an HTML form and then, the next time the guest book is viewed, the page will contain your new entry.
- The WHOIS form at Network Solutions allows you to enter a domain name on a form, and

the page returned is different depending on the domain name entered.
- Any search engine lets you enter keywords on an HTML form, and then it dynamically creates a page based on the keywords you enter.

All of these dynamic pages use CGI.

The CGI Mechanism

On most Web servers, the CGI mechanism has been standardized in the following way. In the normal directory tree that the server considers to be the root, you create a subdirectory named **cgi-bin**. (You can see this directory in the figure on the previous page.) The server then understands that any file requested from the special cgi-bin directory should not simply be read and sent, but instead should be **executed**. The output of the executed program is what it actually sent to the browser that requested the page. The executable is generally either a pure executable, like the output of a C compiler, or it is a PERL script. PERL is an extremely popular language for CGI scripting.

Imagine that you type the following URL into your browser: **http://computer.howstuffworks.com/cgi-bin/search.pl**. The server recognized that **search.pl** is in the cgi-bin directory, so it executes **search.pl** (which is a PERL script) and sends the output from the execution to your browser.

You can write your own scripts and try out CGI yourself provided that:

- You know a programming language such as C or PERL.
- You have access to a Web server that handles CGI scripts. If you have paid a Web hosting service to host your Web site, then chances are you have access to CGI scripting through your host. Check with the hosting service for details. If not, then you can experiment by installing a Web server on your home machine and learning to use it. The second option is a bit more complicated, but you are guaranteed to learn a lot in the process!

Simple CGI Scripts

Assuming that you have access to a cgi-bin directory (see the previous section), and assuming that you know either the C programming language or PERL, you can do a whole bunch of interesting experiments with CGI to get your feet wet. Let's start by creating the simplest possible CGI script.

In the article How Web Pages Work, we examined the simplest possible HTML Web page. It looked something like this:

```
<html>
  <body>
     <h1>Hello there!</h1>
  </body>
</html>
```

The simplest possible CGI script would, upon execution, create this simple, static page as its output. Here is how this CGI program would look if you wrote it in C:

```
#include <stdio.h>

int main()
{
  printf("Content-type: text/html\n\n");
  printf("<html>\n");
  printf("<body>\n");
  printf("<h1>Hello there!</h1>\n");
  printf("</body>\n");
  printf("</html>\n");
  return 0;
}
```

On my Web server, I entered this program into the file **simplest.c** and then compiled it by saying:

```
    gcc simplest.c -o simplest.cgi
```

(See How C Programming Works for details on compiling C programs.)

By placing **simplest.cgi** in the **cgi-bin** directory, it can be executed. You can try it out now by typing in or clicking on this URL: http://computer.howstuffworks.com/cgi-bin/simplest.cgi. As you can see, all that the script does is generate a page that says, "Hello there!" The only part that is unexpected is the line that says:

```
  printf("Content-type: text/html\n\n");
```

The line "Content-type: text/html\n\n" is special piece of text that must be the first thing sent to the browser by any CGI script. As long as you remember to do that, everything will be fine. If you forget, the browser will reject the output of the script.

You can do the same thing in PERL. Type this PERL code into a file named **simplest.pl**:

```
#! /usr/bin/perl
print "Content-type: text/html\n\n";
print "<html><body><h1>Hello World!";
print "</h1></body></html>\n";
```

Place the file into your cgi-bin directory. On a UNIX machine, it may help to also type:

```
  chmod 755 simplest.pl
```

This tells UNIX that the script is executable. You can try it out now by typing in or clicking on this URL: http://computer.howstuffworks.com/cgi-bin/simplest.pl.

You have just seen the basic idea behind CGI scripting. It is really that simple! A program executes and its *output* is sent to the browser that called the script. Normal output sent to stdout is what gets sent to the browser.

The whole point of CGI scripting, however, is to create **dynamic content** -- each time the script executes, the output should be different. After all, if the output is the same every time you run the script, then you might as well use a static page. The following C program demonstrates very simple dynamic content:

```
#include <stdio.h>

int incrementcount()
{
  FILE *f;
  int i;

  f=fopen("count.txt", "r+");
  if (!f)
  {
     sleep(1);
     f=fopen("count.txt", "r+");
     if (!f)
       return -1;
  }

  fscanf(f, "%d", &i);
  i++;
  fseek(f,0,SEEK_SET);
  fprintf(f, "%d", i);
  fclose(f);
  return i;
}

int main()
{
  printf("Content-type: text/html\n\n");
  printf("<html>\n");
  printf("<body>\n");
  printf("<h1>The current count is: ")
  printf("%d</h1>\n", incrementcount());
  printf("</body>\n");
  printf("</html>\n");
  return 0;
}
```

With a text editor, type this program into a file named **count.c**. Compile it by typing:

```
    gcc count.c -o count.cgi
```

Create another text file named **count.txt** and place a single zero in it. By placing **counter.cgi** and **count.txt** in the **cgi-bin** directory, you can run the script. You can try it out now by typing in or clicking on this URL: http://computer.howstuffworks.com/cgi-bin/count.cgi. As you can see, all that the script does is generate a page that says, "The current count is: X," where X increments once each time you run the script. Try running it several times and watch the content of the page change!

The **count.txt** file holds the current count, and the little **incrementcount()** function is the function that increments the count in the **count.txt** file. This function opens the **count.txt** file, reads the number from it, increments the number and writes it back to the file. The function actually tries to open the file twice. It does that just in case two people try to access the file simultaneously. It certainly is not a foolproof technique, but for something this simple it works. If the file cannot be opened on the second attempt, -1 is the error value returned to the caller. A more sophisticated program would recognize the -1 return value and generate an appropriate error message.

Forms: Sending Input

We have seen that the creation of CGI scripts is pretty easy. The Web server executes any executable placed in the cgi-bin directory, and any output that the executable sends to stdout appears in the browser that called the script. Now what we need is a way to send input into a script. The normal way to send input is to use an HTML **form**.

You see forms all over the Web. Any page where you have been able to type something in is a form. You see them in search engines, guest books, questionnaires, etc. The home page for HowStuffWorks.com contains at least two mini-forms, one for the "How did you get here?" sidebar and one for the suggestions sidebar (yes, a single HTML page can contain multiple forms). You create the form on your HTML page, and in the HTML tags for the form you specify the name of the CGI script to call when the user clicks the **Submit** button on the form. The values that the user enters into the form are packaged up and sent to the script, which can then use them in any way it likes.

You have actually been seeing this sort of thing constantly and may not have known that it was happening. For example, go to http://www.lycos.com, type the word "test" into the "Search for:" box and press the "Go Get It!" button. The URL of the result page will look like this:

```
http://www.lycos.com/cgi-bin/pursuit?matchmode=and
                    &cat=lycos&query=test&x=10&y=9
```

You can see that the Lycos home page is a form. Lycos has a script in the cgi-bin directory named **pursuit**. The form sends five parameters to the script:

1. matchmode=and
2. cat=lycos
3. query=test
4. x=10
5. y=9

The third one is the search string we entered. The other four mean something to the script as well. The CGI script queries the Lycos database for the word "test" and then returns the results. That's the heart of any search engine!

Let's create a simple form to try this out. Create a file named **simpleform.htm** and enter the following HTML into it:

```
<html>
<body>
  <h1>A super-simple form<h1>
  <FORM METHOD=GET ACTION="http://computer.howstuffworks.com/
cgi-bin/simpleform.cgi">
  Enter Your Name:
  <input name="Name" size=20 maxlength=50>
  <P>
  <INPUT TYPE=submit value="Submit">
  <INPUT TYPE=reset value="Reset">
  </FORM>
</body>
</html>
```

You can click on this URL to try it out: http://computer.howstuffworks.com/simpleform.htm.

As you can see, the HTML code specifies the creation of a form that uses the GET method sent to the CGI script at **http://computer.howstuffworks.com/cgi-bin/simpleform.cgi**. Inside the form is a text input area plus the standard Submit and Reset buttons.

The file **http://computer.howstuffworks.com/cgi-bin/simpleform.cgi** referenced by the form is a C program. It started life as this piece of C code placed in a file named **simpleform.c**:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  printf("Content-type: text/html\n\n");
  printf("<html>\n");
  printf("<body>\n");
  printf("<h1>The value entered was: ")
  printf("%s</h1>\n", getenv("QUERY_STRING"));
  printf("</body>\n");
  printf("</html>\n");
  return 0;
}
```

It was compiled with the following command:

```
gcc simpleform.c -o simpleform.cgi
```

And it was placed in the cgi-bin directory. This program simply picks up the value sent by the form and displays it. For example, you might see the following:

```
The value entered was: Name=John+Smith
```

Name is the identifier for the text input field in the form (each input field on a form should have a unique identifier), and **John+Smith** is a typical name that might be entered on the form. Note that the "+" replaces the space character.

From this example, you can see that the basic process of setting up a form and getting data from a form into a CGI script is fairly straightforward. Here are a couple of details to keep in mind:

- Each input field on the form should have a unique identifier.
- The form needs to use either the GET or the POST method. The GET method has the advantage that you can see the form's values in the URL sent to the script, and that makes debugging easier.
- There are definite limits to the number of characters that can be sent via the GET method, so POST is preferred for large forms.
- Data that comes in via the GET method is received by looking at the QUERY_STRING environment variable (usually read with the **getenv** function in C or the $ENV facility in PERL). Data that comes in via the POST method is available through STDIN using **gets** in C or **read** in PERL.
- The data that comes in is going to have all of the fields concatenated together in a single string, and many characters will be substituted and therefore need translation. For example, all spaces will be replaced with pluses.

The QUERY_STRING environment variable brings up the topic of **environment variables** in general. There are a number of environment variables that you can examine in your CGI scripts, including:

- AUTH_TYPE
- CONTENT_LENGTH
- CONTENT_TYPE
- GATEWAY_INTERFACE
- HTTP_ACCEPT
- HTTP_USER_AGENT
- PATH_INFO
- PATH_TRANSLATED
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_HOST
- REMOTE_IDENT
- REMOTE_USER
- REQUEST_METHOD
- SCRIPT_NAME

- SERVER_NAME
- SERVER_PORT
- SERVER_PROTOCOL
- SERVER_SOFTWARE

There are all sorts of interesting pieces of information buried in these environment variables, including the length of the input string (CONTENT_LENGTH), the METHOD used (GET or POST -- REQUEST_METHOD lets you figure out whether to look in STDIN or QUERY_STRING for the input), the IP address of the user's machine (REMOTE_ADDR), and so on. For complete descriptions of these variables, see CGI Environment Variables.

Creating a Real Form

A real form is going to be made up of a variety of input areas, and it will require some amount of code in the script to undo the character mappings and parse out the individual strings. Let's start by looking at the standard input controls on a form. They include:

- **Single-line text input**
- **Multi-line text input**
- **Selection lists**
- **Check boxes**
- **Radio buttons**
- **Specialized buttons** for submitting or clearing the form

You can combine these controls with other static text and graphics as you would on any other page.

Here are several examples that demonstrate the use of the different control tags:

Single-line edit

The word "input" identifies a single line edit area. The "name" field provides an identifier for the control to the CGI script and should be unique for each control on the form. The "size" field indicates the width, in characters, of the input area on the form. "Maxlength" constrains the maximum number of characters in the input area. "Value" sets an initial value.

```
Enter Name: <input name="Name" size=30 maxlength=50
value="Sample">
```

Typically, the input area is preceded by a piece of static text identifying the purpose of the input field. Shown here is the static text "Enter name:".

You can add the value "type=int" to constrain input to integer values. By default, the type is "text" and it accepts any characters.

Multi-line edit

A multi-line edit area is similar to a input area. You define a name for the control, and you define its size on the form in rows and columns. Anything you put prior to the </textarea> tag will appear in the control as a default value.

```
<textarea name="Company Address" cols=30
rows=4></textarea>
```

Check boxes

A check box is a specialized form of an input area with the type set to "checkbox".

```
<input type=checkbox name="Include" value=1>
```

The value will be returned if the checkbox is selected.

Radio buttons

Radio buttons are similar to check boxes, but they're grouped together visually:

```
Choose the search area:<br>
<input type=radio CHECKED name=universe value=US-STOCK>
Stocks
<input type=radio name=universe value=CA-STOCK>
Canadian Stocks
<input type=radio name=universe value=MMF>
Money Markets
<input type=radio name=universe value=MUTUAL>
Mutual Funds
```

Note that the default radio button can be marked with the word CHECKED. Also note that all radio buttons in the same group have the same name.

Selection lists

A selection list offers the user a choice from a number of options. The tag for a selection list lets you specify the number of visible rows in the "size" field, as well as the values for all of the options.

```
Select an Option<br>
<SELECT size=2 NAME="Option">
    <OPTION> Option 1
    <OPTION> Option 2
    <OPTION> Option 3
    <OPTION> Option 4
</SELECT>
```

The word MULTIPLE creates a multi-selection capability.

Specialized buttons

The following tags create two specialized buttons, one to submit the form to the server and one to reset the form:

```
<INPUT TYPE=submit value="Submit">
<INPUT TYPE=reset value="Reset">
```

Putting It All Together

Let's say that you would like to create a simple questionnaire for one of your Web pages. For example, you would like to ask for the reader's name, sex, age and comment, and then process it in a CGI script. The HTML form might live in a file named http://computer.howstuffworks.com/survey.htm and look like this:

```
<html>
  <body>
    <h1>HSW Survey Form<h1>
    <FORM METHOD=POST ACTION="http:
//www.howstuffworks.com/cgi-bin/survey.cgi">
    Enter Your Name:
    <input name="Name" size=20 maxlength=50>
    <P>Enter your sex:
    <input type=radio CHECKED name=sex value=MALE>Male
    <input type=radio name=sex value=FEMALE>Female
    <P>Select your age<br>
    <SELECT size=2 NAME=age>
      <OPTION> 1-10
      <OPTION> 11-20
      <OPTION> 21-30
      <OPTION> 31-40
      <OPTION> 41-50
      <OPTION> 51-60
      <OPTION> 61 and up
    </SELECT>
    <P>Enter Your Comment:
    <input name="Name" size=40 maxlength=100>
    <P>
    <INPUT TYPE=submit value="Submit">
    <INPUT TYPE=reset value="Reset">
    </FORM>
  </body>
</html>
```

Click on this URL to see the form in action: http://computer.howstuffworks.com/survey.htm.

The CGI script referenced by this form will receive four different pieces of data: the name, age, sex and comment of the reader who submits the form. The script will have to parse out the four values and handle all of the character transformations. The file http://computer.howstuffworks.com/survey.c was used to create the script **survey.cgi** and is perhaps 100 lines long.

Summary

In this quick tour of CGI scripting, we have seen that:

- A CGI script is a program -- generally a C program or a PERL script.
- On most servers, CGI scripts live in a directory named **cgi-bin**. The script is executed when the script's URL is requested by a browser.
- Anything that the script sends to STDOUT will be sent to the browser. The string "Content-type: text/html\n\n" should be the first thing sent. After that, anything goes; but typically, valid HTML tags for a valid HTML document are sent.
- Input is sent to the script by creating an HTML form whose ACTION specifies the script's URL.
- When a script receives the data from the form, it has to parse out the different strings and convert all of the modified characters. We saw a simple C program that can perform these tasks. The CGI library for PERL (see the next page) makes the conversion easy for PERL scripts.

If you were doing this on a real Web site, you would typically store the results from each survey into a text file or a database so that you could look at the results later. That's easy to do from either a C program or a PERL script.