

## Common Gateway Interface (CGI)

---

### Overview

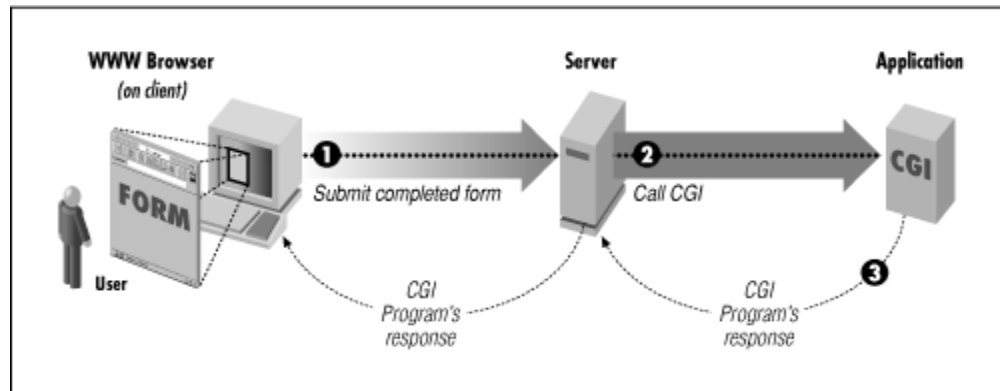
---

The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon **retrieves** is **static**, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is **executed** in real-time, so that it can output **dynamic** information.

For example, let's say that you wanted to "hook up" your Unix database to the World Wide Web, to allow people from all over the world to query it. Basically, you need to create a CGI program that the Web daemon will execute to transmit information to the database engine, and receive the results back again and display them to the client. This is an example of a *gateway*, and this is where CGI, currently version 1.1, got its origins.

The database example is a simple idea, but most of the time rather difficult to implement. There really is no limit as to what you can hook up to the Web. The only thing you need to remember is that whatever your CGI program does, it should not take too long to process. Otherwise, the user will just be staring at their browser waiting for something to happen.

CGI is not thus a language. It's a simple protocol that can be used to communicate between Web forms and your program.



Simple Diagram of CGI

---

### Specifics

---

Since a CGI program is executable, it is basically the equivalent of letting the world run a program on your system, which isn't the safest thing to do. Therefore, there are some security precautions that need to be implemented when it comes to using CGI programs. Probably the one that will affect the typical Web user the most is the

fact that CGI programs need to reside in a special directory, so that the Web server knows to execute the program rather than just display it to the browser. This directory is usually under direct control of the webmaster, prohibiting the average user from creating CGI programs. There are other ways to allow access to CGI scripts, but it is up to your webmaster to set these up for you. At this point, you may want to contact them about the feasibility of allowing CGI access.

If you have a version of the NCSA HTTPd server distribution, you will see a directory called /cgi-bin. This is the special directory mentioned above where all of your CGI programs currently reside. A CGI program can be written in any language that allows it to be executed on the system, such as:

- ? C/C++
- ? Fortran
- ? PERL
- ? TCL
- ? Any Unix shell
- ? Visual Basic
- ? AppleScript

It just depends what you have available on your system. If you use a *programming language* like C or Fortran, you know that you must compile the program before it will run. If you look in the /cgi-src directory that came with the server distribution, you will find the source code for some of the CGI programs in the /cgi-bin directory. If, however, you use one of the *scripting languages* instead, such as PERL, TCL, or a Unix shell, the script itself only needs to reside in the /cgi-bin directory, since there is no associated source code. Many people prefer to write CGI scripts instead of programs, since they are easier to debug, modify, and maintain than a typical compiled program.

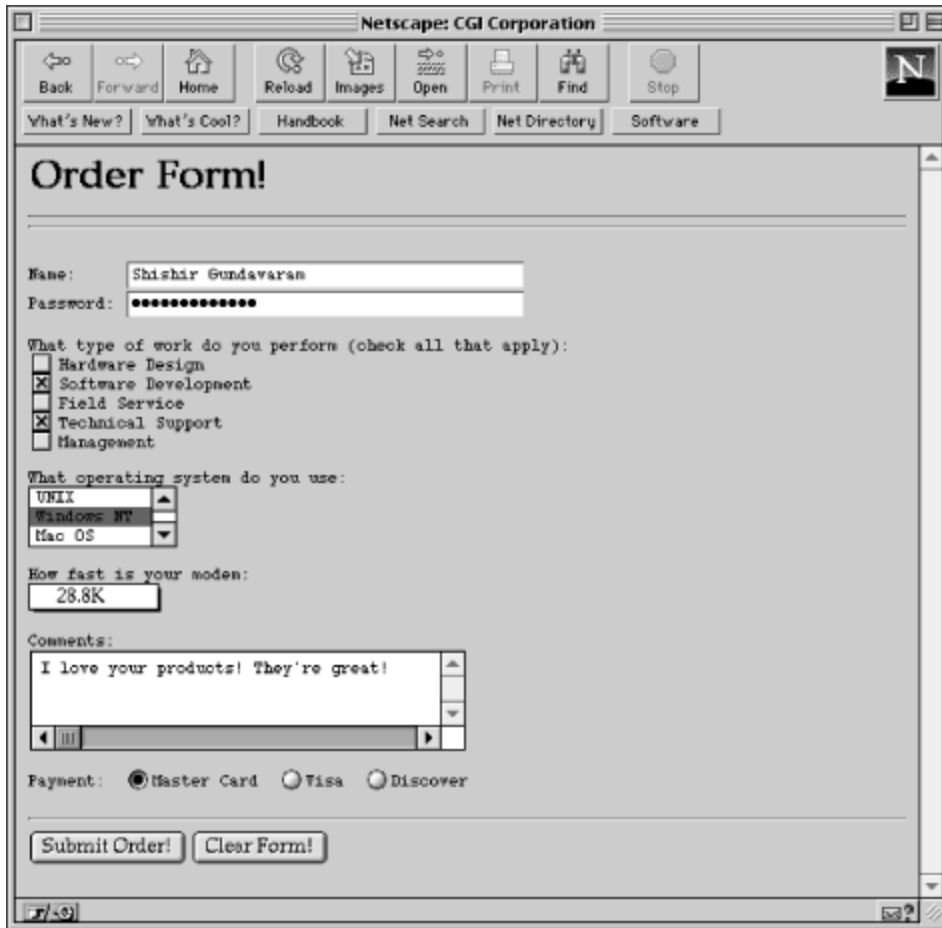
## **CGI Applications**

CGI turns the Web from a simple collection of static hypermedia documents into a whole new interactive medium, in which users can ask questions and run applications. Let's take a look at some of the possible applications that can be designed using CGI.

### **Forms**

One of the most prominent uses of CGI is in processing forms. Forms are a subset of HTML that allow the user to supply information. The forms interface makes Web browsing an interactive process for the user and the provider. The following figure shows a simple form.

## Simple form illustrating different widgets



The image shows a Netscape browser window titled "Netscape: CGI Corporation". The browser's toolbar includes buttons for Back, Forward, Home, Reload, Images, Open, Print, Find, and Stop. Below the toolbar are navigation links: "What's New?", "What's Cool?", "Handbook", "Net Search", "Net Directory", and "Software".

The main content area displays an "Order Form!". The form contains the following elements:

- Name:** A text input field containing "Shishir Gundavaran".
- Password:** A text input field filled with 10 dots.
- What type of work do you perform (check all that apply):** A list of checkboxes:
  - Hardware Design
  - Software Development
  - Field Service
  - Technical Support
  - Management
- What operating system do you use:** A selection list with options: "UNIX", "Windows NT", and "Mac OS". "Windows NT" is currently selected.
- How fast is your modem:** A text input field containing "28.8K".
- Comments:** A text area containing "I love your products! They're great!".
- Payment:** Radio buttons for "Master Card" (selected), "Visa", and "Discover".
- Buttons:** "Submit Order!" and "Clear Form!".

As can be seen from the figure, a number of graphical widgets are available for form creation, such as radio buttons, text fields, checkboxes, and selection lists. When the form is completed by the user, the Submit Order! button is used to send the information to the server, which executes the program associated with the particular form to "decode" the data.

Generally, forms are used for two main purposes. At their simplest, forms can be used to collect information from the user. But they can also be used in a more complex manner to provide back-and-forth interaction. For example, the user can be presented with a form listing the various documents available on the server, as well as an option to search for particular information within these documents. A CGI program can process this information and return document(s) that match the user's selection criteria.

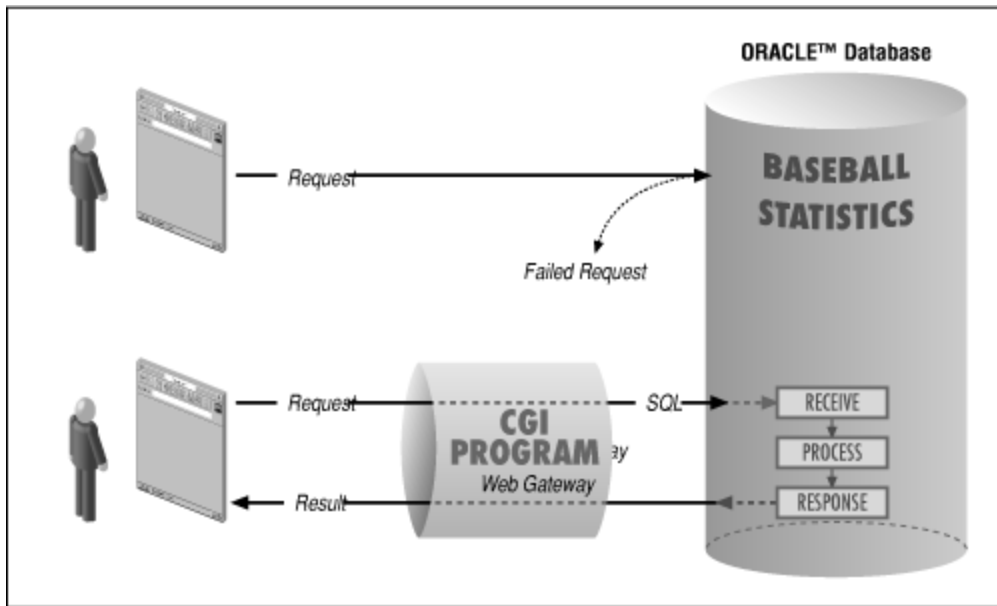
## Gateways

Web gateways are programs or scripts used to access information that is not directly readable by the client. For example, say you have an Oracle database that contains baseball statistics for all the players on your company team and you would like to provide this information on the Web. How would you do it? You certainly cannot point

your client to the database file (i.e., open the URL associated with the file) and expect to see any meaningful data.

CGI provides a solution to the problem in the form of a gateway. Once you have the information, you can format and send it to the client. In this case, the CGI program serves as a gateway to an Oracle database, as shown in below.

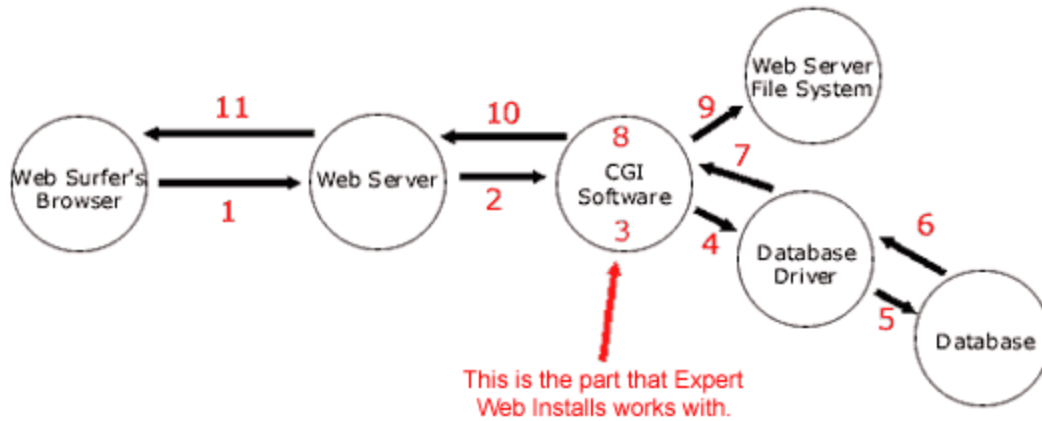
### A gateway to a database



Similarly, you can write gateway programs to any other Internet information service, including Archie, WAIS, and NNTP (Usenet News). In addition, you can amplify the power of gateways by using the forms interface to request a query or search string from the user to retrieve and display *dynamic*, or *virtual*, information which are beyond the scope of this course.

CGI is great for smaller tasks not requiring a large amount of traffic to be pumped through them. It is also available on the largest number of web hosting providers, and relatively easy to use. Use `mod_perl`, C w/ the apache API, PHP, ASP, JSP, Cold Fusion, Zope, or similar technology when working with super massive high traffic sites. Hosting for these high performance platforms can be prohibitively expensive for up and coming businesses however. And unless you're getting 20+ requests a second on your cgi scripts, you probably won't notice any slowness. That number of 20 of course varies wildly based on the hardware and network capacity of your web hosting situation.

Here's a diagram to help you better understand CGI:



And here's what the 11 steps mean:

1. The web surfer fills out a form, and clicks submit. The information in the form is sent over the internet to the web server.
2. The web server "grabs" the information from the form, and passes it to the CGI Software.
3. The CGI Software then performs whatever validation of this information that is required. For instance, it checks to see if an email address is valid. If this is a database program, the CGI Software prepares a database statement, to either add, edit or delete information from the database.
4. The CGI Software then executes the prepared database statement, which is passed to the database driver.
5. The database driver acts as a middleman, and performs the requested action on the database itself.
6. The results of the database action are then passed back to the database driver.
7. The database driver sends the information from the database to the CGI Software.
8. The CGI Software takes the information from the database, and manipulates it into the format that is desired.
9. If any static html pages need to be created (or similar task needs to be performed), the CGI program accesses the web server computer's file system, and reads, writes, and/or edits files.
10. The CGI Software then sends the result it wants the web surfer's

browser to see back to the web server.

11. The web server sends the result it got from the CGI Software back to the web surfer's browser.

When you create an html form like this:

```
<html>
<body>
<form action="/cgi-bin/myscript.cgi" method="post" >
<p>Your name: <input type="text" size="20" name="your_name"><br>
Your email address: <input type="text" size="50" name="email_address">
<input type="submit" value="Print out my name and email!" >
</form>
</body>
</html>
```

You can choose either "get" or "post". Only use get in situations where you need the user to be able to bookmark the resulting page. Otherwise, use post. Using get you can only have 1024 bytes sent to the program, which in a lot of cases isn't enough data.

When the user submits this form, the data made available to your script looks something like this:

[your\\_name=name+father&email\\_address=myemail@somehost.com&x=54&y=102](#)

Actually, because of url encoding, it may look different depending on what browser you send it from. But that's not important. In smart cgi programming, we use what's called a "library" or "module" which shields us from doing the hard stuff like splitting up that string that we get from the form.

So, we use our library to grab the value of the inputted data, then send what's called an "HTTP Header" back to our browser, then we type html code in our perl / c / c++ script to mark up the data we want to send back to the browser.

In a nutshell, that's all there is to cgi. It does many, many great things, including splitting up the data we get from the form into a very easily usable format for us, automatically!

### **How do I get information from the server?**

Each time a client requests the URL corresponding to your CGI program, the server will execute it in real-time. The output of your program will go more or less directly to the client.

A common misconception about CGI is that you can send command-line options and arguments to your program, such as

```
command% myprog -qa blorf
```

CGI uses the command line for other purposes and thus this is not directly possible. Instead, CGI uses environment variables to send your program its parameters. The two major environment variables you will use for this purpose are:

#### ? QUERY\_STRING

QUERY\_STRING is defined as anything which follows the first ? in the URL. This information could be added either by an ISINDEX document, or by an HTML form (with the GET action). It could also be manually embedded in an HTML anchor which references your gateway. This string will usually be an information query, i.e. what the user wants to search for in the database, or perhaps the encoded results of your feedback GET form.

This string is encoded in the standard URL format of changing spaces to +, and encoding special characters with %xx hexadecimal encoding. You will need to decode it in order to use it.

If your gateway is not decoding results from a FORM, you will also get the query string decoded for you onto the command line. This means that each word of the query string will be in a different section of ARGV. For example, the query string "forms rule" would be given to your program with `argv[1]="forms"` and `argv[2]="rule"`. If you choose to use this, you do not need to do any processing on the data before using it.

#### ? PATH\_INFO

CGI allows for extra information to be embedded in the URL for your gateway which can be used to transmit extra context-specific information to the scripts. This information is usually made available as "extra" information after the path of your gateway in the URL. This information is not encoded by the server in any way.

The most useful example of PATH\_INFO is transmitting file locations to the CGI program. To illustrate this, let's say you have a CGI program on your server called `/cgi-bin/foobar` that can process files residing in the DocumentRoot of the server. You need to be able to tell foobar which file to process. By including extra path information to the end of the URL, foobar will know the location of the document relative to the DocumentRoot via the PATH\_INFO environment variable, or the actual path to the document via the PATH\_TRANSLATED environment variable which the server generates for you.

---

### **How do I send my document back to the client?**

I have found that the most common error in beginners' CGI programs is not properly formatting the output so the server can understand it.

CGI programs can return a myriad of document types. They can send back an image to the client, and HTML document, a plaintext document, or perhaps even an audio clip. They can also return references to other documents. The client must know what

kind of document you're sending it so it can present it accordingly. In order for the client to know this, your CGI program must tell the server what type of document it is returning.

In order to tell the server what kind of document you are sending back, whether it be a **full document** or a **reference** to one, CGI requires you to place a short header on your output. This header is ASCII text, consisting of lines separated by either linefeeds or carriage returns (or both) followed by a single blank line. The output body then follows in whatever native format.

? A **full document** with a corresponding MIME type

In this case, you must tell the server what kind of document you will be outputting via a MIME type. Common MIME types are things such as text/html for HTML, and text/plain for straight ASCII text.

For example, to send back HTML to the client, your output should read:

```
Content-type: text/html

<HTML><HEAD>
<TITLE>output of HTML from CGI script</TITLE>
</HEAD><BODY>
<H1>Sample output</H1>
What do you think of <STRONG>this?</STRONG>
</BODY></HTML>
```

? A **reference** to another document

Instead of outputting the document, you can just tell the browser where to get the new one, or have the server automatically output the new one for you.

For example, say you want to reference a file on your Gopher server. In this case, you should know the full URL of what you want to reference and output something like:

```
Content-type: text/html
Location: gopher://httprules.foobar.org/0

<HTML><HEAD>
<TITLE>Sorry...it moved</TITLE>
</HEAD><BODY>
<H1>Go to gopher instead</H1>
Now available at
<A HREF="gopher://httprules.foobar.org/0">a new location</A>
on our gopher server.
</BODY></HTML>
```

However, today's browsers are smart enough to automatically throw you to the new document, without ever seeing the above since. If you get lazy and



don't want to output the above HTML, NCSA HTTPd will output a default one for you to support older browsers.

If you want to reference another file (not protected by access authentication) on your own server, you don't have to do nearly as much work. Just output a partial (virtual) URL, such as the following:

Location: /dir1/dir2/myfile.html

The server will act as if the client had not requested your script, but instead requested `http://yourserver/dir1/dir2/myfile.html`. It will take care of most everything, such as looking up the file type and sending the appropriate headers. Just be sure that you output the second blank line.

If you do want to reference a document that is protected by access authentication, you will need to have a full URL in the Location:, since the client and the server need to re-transact to establish that you access to the referenced document.

Advanced usage: If you would like to output headers such as Expires or Content-encoding, you can if your server is compatible with CGI/1.1. Just output them along with Location or Content-type and they will be sent back to the client.

## **Structure of a CGI Script**

Here's the typical sequence of steps for a CGI script:

1. Read the user's form input.
2. Do what you want with the data.
3. Write the HTML response to STDOUT.

The first and last steps are described below

### **Reading the User's Form Input**

When the user submits the form, your script receives the form data as a set of name-value pairs. The names are what you defined in the INPUT tags (or SELECT or TEXTAREA tags), and the values are whatever the user typed in or selected. (Users can also submit files with forms, but this primer doesn't cover that.)

This set of name-value pairs is given to you as one long string, which you need to parse. It's not very complicated, and there are plenty of existing routines to do it for you.

If that's good enough for you, skip to the next section. If you'd rather do it yourself, or you're just curious, the long string is in one of these two formats:

**"name1=value1&name2=value2&name3=value3"**  
**"name1=value1;name2=value2;name3=value3"**

So just split on the ampersands or semicolons, then on the equal signs. Then, do two more things to each name and value:

1. Convert all "+" characters to spaces, and
2. Convert all "%xx" sequences to the single character whose ascii value is "xx", in hex. For example, convert "%3d" to "=".

This is needed because the original long string is **URL-encoded**, to allow for equal signs, ampersands, and so forth in the user's input.

So where do you get the long string? That depends on the HTTP method the form was submitted with:

- ? For GET submissions, it's in the environment variable **QUERY\_STRING**.
- ? For POST submissions, read it from STDIN. The exact number of bytes to read is in the environment variable **CONTENT\_LENGTH**.

### **What is the difference between GET and POST?**

GET and POST are two different methods defined in HTTP that do very different things, but both happen to be able to send form submissions to the server.

Normally, GET is used to get a file or other resource, possibly with parameters specifying more exactly what is needed. In the case of form input, GET fully includes it in the URL, like

```
http://myhost.com/mypath/myscript.cgi?name1=value1&name2=valu  
e2
```

GET is how your browser downloads most files, like HTML files and images. It can also be used for most form submissions, if there's not too much data (the limit varies from browser to browser).

The GET method is *idempotent*, meaning the side effects of several identical GET requests are the same as for one GET request. In particular, browsers and proxies can cache GET responses, so two identical form submissions may not both make it to your CGI script. So don't use GET if you want to log each request, or store data or otherwise take an action for each request.

Normally, POST is used to send a chunk of data to the server to be processed, whatever that may entail. (The name POST might have come from the idea of posting a note to a discussion group or newsgroup.) When an HTML form is submitted using POST, your form data is attached to the end of the POST request, in its own object (specifically, in the message body). This is not as simple as using GET, but is more versatile. For example, you can send entire files using POST. Also, data size is not limited like it is with GET.

All this is behind the scenes, however. To the CGI programmer, GET and POST work almost identically, and are equally easy to use. Some advantages of POST are that you're unlimited in the data you can submit, and you can count on your script being called every time the form is submitted. One advantage of GET is that your entire form submission can be encapsulated in one URL, like for a hyperlink or bookmark

---

## **Sending the Response Back to the User**

First, write the line

```
Content-type: text/html
```

plus another blank line, to STDOUT. After that, write your HTML response page to STDOUT, and it will be sent to the user when your script is done. That's all there is to it.

Yes, you're generating HTML code on the fly. It's not hard; it's actually pretty straightforward. HTML was designed to be simple enough to generate this way.

---

## **Useful CGI Environment Variables**

In order to pass data about the information request from the server to the script, the server uses command line arguments as well as environment variables. These environment variables are set when the server executes the gateway program. CGI scripts have access to 20 or so environment variables, such as QUERY\_STRING and CONTENT\_LENGTH.

---

## **Specification**

The following environment variables are not request-specific and are set for all requests:

? SERVER\_SOFTWARE

The name and version of the information server software answering the request (and running the gateway). Format: name/version

? SERVER\_NAME

The server's hostname, DNS alias, or IP address as it would appear in self-referencing URLs.

? GATEWAY\_INTERFACE

The revision of the CGI specification to which this server complies. Format: CGI/revision

---

The following environment variables are specific to the request being fulfilled by the gateway program:

? SERVER\_PROTOCOL

The name and revision of the information protocol this request came in with. Format: protocol/revision

? SERVER\_PORT

The port number to which the request was sent.

? REQUEST\_METHOD

The method with which the request was made. For HTTP, this is "GET", "HEAD", "POST", etc.

? PATH\_INFO

The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH\_INFO. This information should be decoded by the server if it comes from a URL before it is passed to the CGI script.

? PATH\_TRANSLATED

The server provides a translated version of PATH\_INFO, which takes the path and does any virtual-to-physical mapping to it.

? SCRIPT\_NAME

A virtual path to the script being executed, used for self-referencing URLs.

? QUERY\_STRING

The information which follows the ? in the URL which referenced this script. This is the query information. It should not be decoded in any fashion. This variable should always be set when there is query information, regardless of command line decoding.

? REMOTE\_HOST

The hostname making the request. If the server does not have this information, it should set REMOTE\_ADDR and leave this unset.

? REMOTE\_ADDR

The IP address of the remote host making the request.

? AUTH\_TYPE

If the server supports user authentication, and the script is protected, this is the protocol-specific authentication method used to validate the user.

? REMOTE\_USER

If the server supports user authentication, and the script is protected, this is the username they have authenticated as.

? REMOTE\_IDENT

If the HTTP server supports RFC 931 identification, then this variable will be set to the remote user name retrieved from the server. Usage of this variable should be limited to logging only.

? CONTENT\_TYPE

For queries which have attached information, such as HTTP POST and PUT, this is the content type of the data.

? CONTENT\_LENGTH

The length of the said content as given by the client.

---

In addition to these, the header lines received from the client, if any, are placed into the environment with the prefix HTTP\_ followed by the header name. Any - characters in the header name are changed to \_ characters. The server may exclude any headers which it has already processed, such as Authorization, Content-type, and Content-length. If necessary, the server may choose to exclude any or all of these headers if including them would exceed any system environment limits.

An example of this is the HTTP\_ACCEPT variable which was defined in CGI/1.0. Another example is the header User-Agent.

? HTTP\_ACCEPT

The MIME types which the client will accept, as given by HTTP headers. Other protocols may need to get this information from elsewhere. Each item in this list should be separated by commas as per the HTTP spec.

Format: type/subtype, type/subtype

? HTTP\_USER\_AGENT

The browser the client is using to send the request. General format:  
software/version library/version.

---

**System Requirements:**

- 1) Most operating systems will work with Perl/CGI
- 2) Your web server must be configured for CGI. You may need to talk to your systems administrator or web hosting provider about this.
- 3) You must have perl / c++ / C installed.

Writing your first script:

Create a new text file in a plain text editor such as Microsoft Wordpad. I use Textpad, a shareware program. Name it first.cgi. Make sure to save this file in plain text only. Do not save it with any special formatting or it won't work properly.

```
#include <stdio.h>

void main() {

    /** Print the CGI response header, required for all HTML output. **/
    /** Note the extra \n, to send the blank line.          **/
    cout<<"Content-type: text/html\n\n" ;

    /** Print the HTML response page to STDOUT. **/
    cout<<"<html>\n" ;
    cout<<"<head><title>CGI Output</title></head>\n" ;
    cout<<"<body>\n" ;
    cout<<"<h1>Hello, world.</h1>\n" ;
    cout<<"</body>\n" ;
    cout<<"</html>\n" ;

    exit(0) ;
}
```

Now, go to your web browser such as internet explorer and type in

http://yourfullpath/cgi-bin/first.cgi

You can also have a look at a cgi written in c : [hello\\_c.txt](#) that demonstrates basic CGI programming, and use of the getcivars() routine.

Because a CGI program runs each time a user requests the page, the CGI program can customize the page for each user. More power comes, however, when you include an HTML form. A form on a Web page can have the name of a CGI program as an action element. When the user fills in the form in the browser and clicks the

Submit button, the information gets sent right to the CGI program. The CGI program can easily discover what information the user typed in and use this information to build a custom Web page.

CGI programs must invariably parse plain text; Perl's high-level syntax, flexibility, and text-manipulation routines make it an ideal language in which to program CGI.

However, Perl and other very high-level scripting languages have limitations. One downside is their size. The Perl executable can be as much as ten times larger than CGI C binaries. While some of the CGI libraries for Perl greatly simplify programming, some do so at a cost in server performance. Since most servers fork a separate process every time a CGI program is invoked, overhead can grow rapidly on a high-traffic site with lots of CGI access.

Some Web servers (most notably Netscape and Apache) have their own APIs. These allow you to code your CGI programs as extensions to the server, thus avoiding the overhead created by forking new processes. Communicating with these APIs generally means coding your CGI programs in C.

Since Perl contains powerful pattern-matching operators and string manipulation functions, it is very simple to decode form information. Unfortunately, this process is not as easy when dealing with other high-level languages like c and c++, as they lack these kinds of operators. However, there are various libraries of functions that make the decoding process easier.

## Examples

### 1. Browser identification – c++

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main ()
{
    char *http_user_agent;
    cout<<"Content-type: text/plain\n\n";
    http_user_agent = getenv ("HTTP_USER_AGENT");
    if (http_user_agent == NULL) {
        cout<<"Oops! Your browser failed to set the HTTP_USER_AGENT ";
        cout<<"environment variable!\n";
    } else if (!strncmp (http_user_agent, "Mosaic", 6)) {
        cout<<"I guess you are sticking with the original, huh?\n";
    } else if (!strncmp (http_user_agent, "Mozilla", 7)) {
        cout<<"Well, you are not alone. A majority of the people are ";
        cout<<"using Netscape Navigator!\n";
    } else if (!strncmp (http_user_agent, "Lynx", 4)) {
        cout<<"Lynx is great, but go get yourself a graphic browser!\n";
    } else {
        cout<<"I see you are using a browser.\n", http_user_agent;
        cout<<"I don't think it's as famous as Netscape, Mosaic or Lynx!\n";
    }
    exit(0);
}
```

```
}
```

## 2. Simple Multiplication (using form processing) - c

```
<FORM ACTION="/cgi-bin/yourpath/mult.cgi">  
<p>Please specify the multiplicands:  
<INPUT NAME="m" SIZE="5">  
<INPUT NAME="n" SIZE="5">  
<INPUT TYPE="SUBMIT" VALUE="Multiply!">  
</FORM>
```

### The Code

```
#include <stdio.h>  
#include <stdlib.h>  
int main(void) {  
    char *data;  
    long m,n;  
    printf("%s%c%c\n", "Content-Type: text/html; charset=iso-8859-1",13,10);  
    printf("\n");  
    printf(" Multiplication results\n");  
    data = getenv("QUERY_STRING");  
    if(data == NULL) printf(" Error! Error in passing data from form to script.");  
    else if(sscanf(data,"m=%ld&n=%ld",&m,&n)!=2)  
        printf(" Error! Invalid data. Data must be numeric.");  
    else printf(" The product of %ld and %ld is %ld.",m,n,m*n);  
    return 0; }
```

For forms which use METHOD="POST", CGI specifications say that the data is passed to the script or program in the standard input stream (stdin), and the length (in bytes, i.e. characters) of the data is passed in an environment variable called CONTENT\_LENGTH. When reading the input, the program must not try to read more than CONTENT\_LENGTH characters.