CGI Programming Using Perl

Student Workbook

itcourseware
THE SOURCE FOR IT PROFESSIONALS®

CGI Programming Using Perl

Richard Raab

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112.

Contributing Author: Jeff Howell

Editor: Rob Roselius

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Brandon Caldwell, Jeff Howell, and Jim McNally, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 1 - Course Introduction

Notes

Course Objectives

- Describe the basic client/server architecture that makes up the Internet.
- List the request methods and response codes defined by the HTTP standards.
- Write and test CGI programs.
- Use Perl's *CGI.pm* module to write CGI programs.
- Use *CGI.pm* methods in CGI programs to generate HTML pages.
- Use *CGI.pm* methods in CGI programs to generate HTML forms.
- Explain the details of the Common Gateway Interface architecture.
- Use CGI programs to handle input from HTML forms.
- Develop CGI applications that use static and dynamic HTML forms.
- Write CGI applications that maintain state information across web transactions.
- Use a CGI program to interoperate with a database system.
- Take advantage of advanced features and technologies related to CGI.
- Recognize and deal with security issues related to CGI.

Notes

Course Overview

- **Audience**: This is a programming course designed for software development professionals; you will write and manipulate many CGI programs in this course.
- **Prerequisites**: Experience with HTML is essential. Proficiency with text editing on UNIX and basic knowledge of file manipulation are necessary. Experience in programming in Perl, and knowledge of Web Server Administration, are helpful but not necessary.
- **Classroom Environment:**
  - UNIX or Linux web server host and software development environment, with one workstation per student.
  - Perl 5.002 or later, with the *CGI.pm* and *GD.pm* modules installed.
  - Apache or equivalent *httpd* Web Server, configured for CGI.
  - Netscape 2.0 (or later) compatible Web Browser available for each student.

Notes

Suggested References

Gundavaram, Shishir. 2000. ***CGI Programming with Perl, Second Edition***. O'Reilly and Associates, Sebastopol, California. ISBN 1-56592-419-3.

Patchett, Craig and Wright, Matthew. 1997. ***The CGI/Perl Cookbook***. John Wiley & Sons, New York, New York. ISBN 0-471-16896-3.

Siever, Ellen, Spainhour, Stephen and Patwardhan, Nathan. 1999. ***Perl in a Nutshell***. O'Reilly and Associates, Sebastopol, California. ISBN 1-56592-286-7.

Spainhour, Stephen and Eckstein, Robert. 1999. ***Webmaster in a Nutshell, Third Edition***. O'Reilly and Associates, Sebastopol, California. ISBN 0-596-00357-9.

Stein, Lincoln. 1998. ***Official Guide to Programming with CGI.pm***. John Wiley & Sons, New York, New York. ISBN 0-471-24744-8.

Wall, Larry, Christiansen, Tom, and Schwartz, Randal. 1996. ***Programming Perl, Third Edition***. O'Reilly and Associates, Sebastopol, California. ISBN 0-596-00027-8.

Yeager, Nancy and McGrath, Robert E. 1996. ***Web Server Technology***. Morgan Kaufmann Publishers, Inc., San Francisco, California. ISBN 1-55860-376-X

http://stein.chsl.org/WWW/software/CGI/

http://web.golux.com/coar/cgi/

http://www.perl.com/

http://www.w3.org/

Notes

Chapter 2 - The Internet and the Web

Notes

Chapter Objectives

- Describe the mechanism by which computers communicate on the Internet.
- Describe the Domain Name System.
- Explain how client and server programs connect on the Internet.

Notes

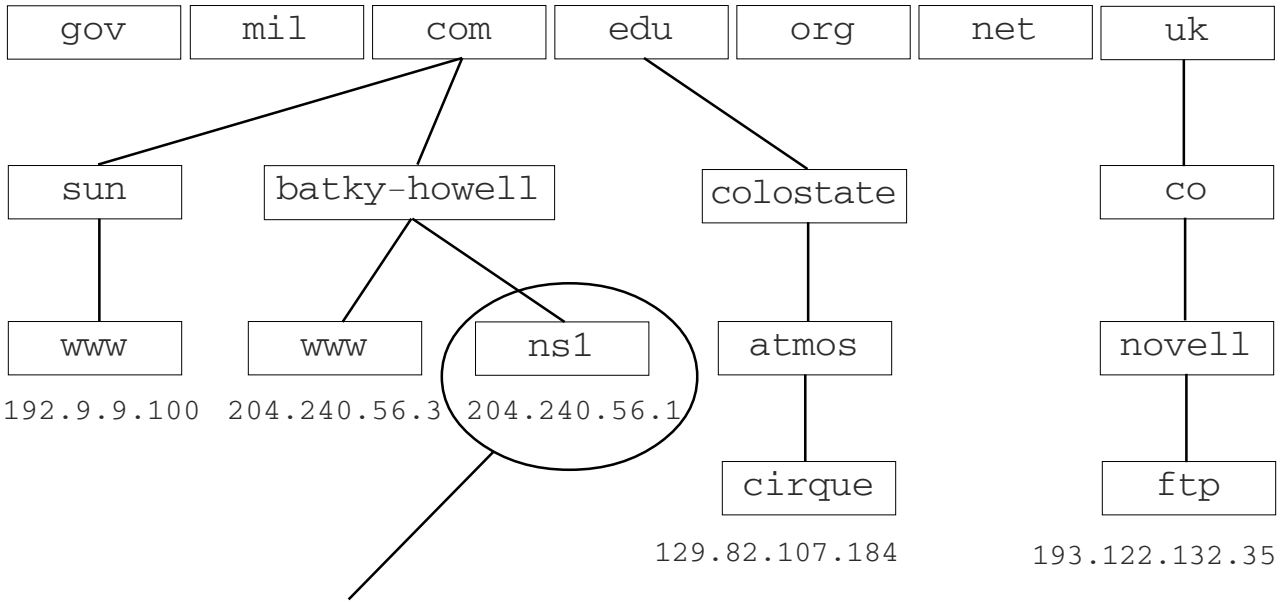
TCP/IP and Ports

- Every host (computer) on the Internet has a unique *IP address* (Internet Protocol address).
  - You have probably seen IP addresses, which look like `204.240.56.3`.
- To request a connection to another Internet host, you must know its IP address.
- Each host accepts and services only certain types of connection request, each type handled by a server program or *daemon*.
- These services are each identified by a *port* number.
  - Many common services have *well-known* port numbers, which by convention are the same on all hosts.
  - Some examples (from */etc/services* on UNIX):

```
netstat  15   # Network status information
ftp      21   # File Transfer Protocol
telnet   23   # Remote terminal emulation
smtp     25   # Simple Mail Transfer Protocol
time     37   # Network Time Service
name     42   # Domain Name Service
http     80   # Hypertext Transfer Protocol
nntp     119  # Network News Transfer Protocol
```

- So, to use services on the Internet, you (or rather, your software) must know both the IP address of the remote host and the port number of the service you need.

Notes



Name server host for the domain `batky-howell.com`. Any request for the IP address of a host in the `batky-howell.com` domain, for example, `www.batky-howell.com`, is serviced by the `named` daemon on this machine. In additon, programs running on hosts within this domain will send any hostname, for example `www.sun.com`, to this nameserver for IP address resolution. If the nameserver doesn't have that hostname in its database, it sends it to the nameserver for the next higher domain (`com`). The request then goes to the nameserver of the appropriate subdomain.

Requests for the IP addresses of other hosts, such as `cirque.atmos.colostate.edu` or `ftp.novell.co.uk`, are serviced by the nameservers for those domains (`atmos.colostate.edu` and `novell.co.uk`) and for their parent domains (`colostate.edu`, `edu`, and `co.uk`, `uk`).

DNS vs. Hosts Files

- For the convenience of users, hosts on the Internet are given names.
- When we reference a host by name, TCP/IP programs must first resolve the name into an IP address.
- To get the IP address of a host, a program can simply look up the hostname in a local file (*/etc/hosts* or *system32\drivers\etc\hosts*) and retrieve the IP address it finds on that line.
  - Each machine must have its own copy of the *hosts* file; keeping them all up to date is a chore.
  - A *hosts* file can't possibly list all the hosts on the Internet.
- The Domain Naming System (DNS) allows a single machine to maintain the list of hostnames and IP addresses for a network.
  - The Domain Naming System is hierarchical; if the name server daemon can’t find the requested hostname, it passes the request to a higher level server.
- To get the IP address of a host, a program sends a query containing the hostname to a name server daemon; the name server daemon then returns the corresponding IP address.
  - Note that since the name server daemon is running on another machine, we must already know *its* IP address.
- Some UNIX systems provide a DNS lookup command (*nslookup*, for example) to directly query a DNS server.

Notes

What is an association?

To communicate, two processes must establish an association, which is a 5-tuple that completely identifies the two processes (a client on one computer and a server on another) that make up a connection:

{*network protocol, local-address, local process, foreign-address, foreign process*}

What is an endpoint?

An endpoint, or socket, (also known as a “half-association”), specifies half of a connection:

{*network protocol, address, process*}

What is a port?

A port is an integer that is used to identify a specific process on a computer. It is the *process* member above. A **well-known port**, used by server programs, is an identifier of a service provided by a machine, and an **ephemeral port** (or "temporary" port) is used by a client. Ephemeral ports are allocated from the operating system by the client program, and then returned to the O.S. for reuse when the client/server transaction is complete.

Servers and Clients

- TCP/IP provides peer-to-peer communication, but does not specify when or why peer applications interact.
- The fundamental justification for Client/Server computing comes from the problem of rendezvous.
  - Because TCP/IP does not provide any mechanism to start a program when a message arrives, a server program must be waiting to accept requests before they arrive.
- Definitions:
  - A *client* is an application that initiates peer-to-peer communication in pursuit of a service (or services).
    - A browser is a client application.
  - A *server* is a program that waits for service requests from clients and processes them.
    - A server executes on its host system, doing work on behalf of the client.

Notes

Each client program establishes a connection to a server daemon on another host. It then converses with the server daemon using a simple vocabulary which is defined by a *protocol*.

A protocol is simply an agreement between those who create server daemon programs and those who write the programs that connect to those daemons, on the language the two programs will speak. The agreement is traditionally hammered out in an *RFC* (Request For Comment), which is published by someone who wants to establish a new kind of service. Others then contribute ideas, comments, and sample programs, and eventually the RFC becomes the standard defining the new service. This loosely organized, open, and collaborative process has resulted in the Internet as we know it today. RFCs are assigned unique numbers; following is a list of some "famous" RFCs (though there are actually dozens of RFCs covering different aspects of each of these services):

RFC 791	The Internet Protocol
RFC 821	The Simple Mail Transfer Protocol
RFC 854	The TELNET Protocol
RFC 959	The File Transfer Protocol
RFC 822	Format of Internet Text Messages (specifies header syntax which is used by several protocols.)
RFC 1034	Domain Names (DNS)

See *http://info.internet.isi.edu/in-notes/rfc/* for a complete list.

These protocols are open and available, and anyone can write programs which use them. The programs we discuss in this chapter were first written many years ago, but are still very widely used. Modern software for using Internet services incorporates all of these protocols - file transfer, mail, news, address resolution - plus a few new protocols and standards such as *HTTP* (Hypertext Transfer Protocol) and *HTML* (Hypertext Markup Language) into a single interface, hiding the details from users; anyone can now easily access the global Internet without knowing anything about the underlying protocols.

Of course, some of us still need to know what's going on behind the scenes...

The *telnet* program allows you to specify a port number to use, instead of the well-known TELNET port number 23. Thus, you can use *telnet* to establish an interactive session with *any* network service daemon, not just the telnet daemon! Telnet is often used in this way by programmers and system administrators to debug and diagnose network server daemons. It can also be used mischievously or maliciously - for example, someone who knows the details of the SMTP (Simple Mail Transfer Protocol) could use it to connect to an email server daemon and forge messages. This would take some skill and cleverness; system and network administrators are also skillful and clever at detecting such abuses.

Client/Server Protocols

- A protocol is simply an agreement between those who create server daemon programs and those who write the client programs that connect to those daemons, on the language the two programs will speak.

- The actual vocabulary used by most Internet protocols is human-readable.

- Most of these protocol languages specify *requests* and *responses.*

  Example requests:

  FTP: `USER`, `PASS`, `CWD`, `CDUP`, `LIST`, `PORT`, `RETR`, `QUIT`
  SMTP: `HELO`, `RCPT`, `MAIL`, `SEND`, `QUIT`
  HTTP: `GET`, `POST`, `HEAD`, `PUT`

- These protocols are open and available, and anyone can write programs which use them.

Notes

Chapter 6 - Generating HTML

Notes

Chapter Objectives

- Use the online documentation for *CGI.pm*.
- Use *CGI.pm* to generate HTML in Perl CGI programs.
- Describe the rules of *CGI.pm* pertaining to general HTML methods.

Notes

For any HTML tag, *CGI.pm* allows you to generate the tag by calling a function of the same name.

```
print strong('Special Offer!');
```

If you want to use a non-standard tag, say so by listing its name, along with the `:standard` tag, when you include *CGI.pm* in your program with `use`. This works even if the tag is unknown to *CGI.pm*! If you call an HTML method that *CGI.pm* has never seen before, it will automatically create a method that generates valid HTML.

```
use CGI(':standard', 'fireworks');
...
print fireworks('Holiday Sale!');
# This will produce: <FIREWORKS>Holiday Sale!</FIREWORKS>
```

And, since all browsers silently ignore any tags they don't understand, it won't make your HTML output non-portable.

Hands On:

Modify *order.cgi* to create a centered heading colored red.

```
#!/usr/bin/perl

use CGI qw(:standard);

print header();
print start_html( -TITLE=>'Domoniques Pizza');
print h1( {-align=>'center'},
          font( {-color=>'red'}, "Domoniques Pizza") );
print end_html();
```


Introduction to CGI.pm

- To use *CGI.pm*, you must first include it in your Perl program with the `use` command.

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

  - This makes all of *CGI.pm*'s standard HTML-generating methods available in your program.

```
print start_html();
```

- *CGI.pm*'s methods return text strings containing HTML tags.

```
print hr();  # Creates a horizontal rule tag: <HR>
```

  - Parentheses are optional if you're not passing parameters.

```
print hr;
```

- For HTML tags that enclose text, pass the text to the method.

```
print h1('Section One: Introduction');
# Creates: <H1>Section One: Introduction</H1>
```

- For HTML tags with optional attributes, you can pass the attributes as parameters in a hash reference. (More about this soon.)

```
print font( { -size=>'+2', -color=>'red' },
            'Order now!' );
# <FONT size="+2" color="red">Order now!</FONT>
```

Notes

Hands On:

First, find where the file containing *CGI.pm* is located. The instructor will help you, perhaps by executing the command **`perl -V`** to find the location of Perl's libraries on the classroom host. Using Perl's pod2html command, create your own copy of *CGI.pm*'s internal documentation, in HTML format, so you can view it in your browser during the rest of the class.

For example, if the path to the *CGI.pm* file is */usr/local/lib/perl5/CGI.pm*, then the command would be:

```
pod2html /usr/local/lib/perl5/CGI.pm > ~/public_html/CGI.pm.html
chmod 644 CGI.pm.html
```

Load this page in your browser, and take a few minutes to familiarize yourself with the structure of this document.

CGI Objects

In reading *CGI.pm*'s documentation, you'll see that most of the examples use the following style:

```
print $query->textarea( -name=>'feedback',
                        -rows=>'5', -columns=>'40');
```

The variable `$query` represents the content of a CGI `GET` or `POST` request which has been received by your program. To use this variable (you can actually name it anything you want), you must first create it:

```
$query = new CGI;
```

You can then use this variable to call *CGI.pm*'s methods in an "object-oriented" style, and you can also keep track of more than one query in a single program by creating multiple query variables.

But it's not necessary to create a query variable to use *CGI.pm*. See the **Programming Style** section of *CGI.pm*'s documentation.

How to use CGI.pm's Online Documentation

- *CGI.pm* is a program module, written in Perl, and installed in your local Perl software directory.
- Perl has a documentation format that allows a programmer to embed extensive documentation of a program in the program itself.
  - This is called *POD* (Plain Old Documentation).
- Perl includes a number of tools for converting this text to different formats.
  - pod2text
  - pod2man
  - pod2latex
  - pod2html
- *CGI.pm* has extensive internal POD documentation.
  - It starts at the first line that begins with `=head`.

Notes

Example program that demonstrates paired and unpaired tags.

```
#!/usr/bin/perl

use CGI qw(:standard);

print header;
print start_html;
print b("This line is bold.\n");
print i;
print "This line is italic (but not bold).\n", br;
print "This line is italic non-bold too!\n";
print "</i>";  # Turn off italics
print end_html;
```


CGI.pm's Rules for General HTML Tags

- For any general HTML tag name, use a method of the same name to produce the tag in your CGI program.

```
print br;
```

- Tag attributes are passed to a method as a Perl *hash reference* (a list of *attribute=>value* pairs in braces) as the first parameter.

```
print hr({-align=>'left', -size=>'6',-noshade=>1});
```

- The method will generate a tag pair if any parameters other than attributes are passed in.

```
print font( { -size=>'+2', -color=>'red' },
            "Order now!\n" );
```

- The method will generate a single tag if no other parameters are passed in.

```
print b;
print font( { -size=>'1', -color=>'lightgrey' } );
print 'Offer does not apply in all states.',
      'Interest doubles after 1 week. No refunds.';
print '</FONT>';
print '</B>';
```

  - Don't forget to close any non-empty tags you create!

- No error checking is performed on parameters, so:

  - Programs can adapt to changing standards.

  - Less run-time overhead.

Notes

CGI.pm's Rules for HTML Form Tags

- Form element tag attributes are passed to a method as parameters.

```
print textarea(   -name=>'feedback',
                  -rows=>'5', -columns=>'40' );
```

- Some form components (radiobutton groups, popup menus, option lists) have several elements.
  - For these, pass the list of elements to the *CGI.pm* method as a list or hash argument.
  - In the following `popup_menu` call, the values are passed as a list and the labels are passed as a hash.

```
print popup_menu( -name=>'contact',
                  -values=>['t', 'f', 'p'],
                  -default=>'t',
                  -labels=>{'t'=>'Telephone',
                            'f'=>'Fax',
                            'p'=>'Pager'} );
```

Notes

When using named parameters in *CGI.pm* function calls, the first parameter name begins with a hyphen, but the leading hyphen is optional for the rest of the names:

```
print radio_group(
   -NAME => 'shell',
   VALUES=> ['bash','tcsh','ksh','csh'],
   LABELS=> {
      'bash'=>'Bourne-Again Shell',
      'tcsh'=>'Tcsh',
      'ksh'=>'Korn Shell',
      'csh'=>'C Shell'
   }
);
```

If you call `use_named_parameters` before calling any other *CGI.pm* methods, then the leading hyphen on the first parameter is optional.

```
use_named_parameters( 1 );

print radio_group(
   NAME => 'shell',
   VALUES=> ['bash','tcsh','ksh','csh'],
   LABELS=> { 'bash'=>'Bourne-Again Shell',
      'tcsh'=>'Tcsh',
      'ksh'=>'Korn Shell',
      'csh'=>'C Shell'
   }
);
```

But, to avoid potential conflicts with Perl language keywords, it is recommend that you put a leading hyphen in front of all parameter names in the *CGI.pm* function calls.

Capitalization of parameter names is a programmer's style choice. Some people find it improves readability.

Named vs. Positional Parameters in CGI.pm

- *Positional parameters* are the original style for passing parameters to *CGI.pm*'s form element methods.

```
print radio_group( 'shell',             # Name
          ['bash','tcsh','ksh','csh'], # Button values
          'ksh',                       # Default button
          ,                            # Vertical list
          { 'bash'=>'Bourne-Again Shell',
            'tcsh'=>'Tcsh',
            'ksh'=>'Korn Shell',
            'csh'=>'C Shell'
          }                            # Button labels
);
```

  - Note that defaulted parameters need commas.

- *Named parameters* are the preferred style.

```
print radio_group( -NAME=>'shell',
          -VALUES=>['bash','tcsh','ksh','csh'],
          -LABELS=>{ 'bash'=>'Bourne-Again Shell',
                     'tcsh'=>'Tcsh',
                     'ksh'=>'Korn Shell',
                     'csh'=>'C Shell'
                   },
          -DEFAULT=>'ksh'
);
```

- The advantages of named parameters include:
  - You do not have to remember the order of parameters.
  - It's easy to see at a glance what is passed in.
  - You can add new parameters to be incorporated.

Notes

Labs

1. Write a program that uses the `header` subroutine with no parameters. Run the program from the command line to see the HTTP text. What is the default media type used by *CGI.pm*? (Solution: *media.cgi*)

2. Modify the program in #1 to send the media type `image/gif`. Again, run it in off-line mode. (Solution: *mediaGIF.cgi*)

3. Write a program that displays the following two lines, with the word `Blue` colored blue and the word `Red` colored red:

```
Blue is my second favorite color.
Red is my favorite.
```

   (Solution: *blueRed.cgi*)

4. Write a program that displays the header `Electronic Communication Channels`, followed by four lines with the words `Telephone`, `Fax`, `E-mail`, and `Pager`, with the four lines separated by horizontal rules that are left-justified, 5 pixels thick, nonshaded, and take 30% of the width of the browser screen. (Solution: *page1.cgi*)

5. Modify *page1.cgi* so that it contains exactly one Perl `print` statement, which means that the CGI method calls and the lines of text are comma-separated parameters to the `print` statement. (Solution: *page2.cgi*)

6. Examine and run each of the four programs *bi1.cgi*, *bi2.cgi*, *bi3.cgi*, and *bi4.cgi*. Run them from the command line and from a browser. Make sure you understand everything in these programs. (Note: "*bi*" means "bold-italic.")

Notes

Chapter 12 - Additional Web Programming Features

Notes

Chapter Objectives

- Use "extra path information" to send information from a browser to a CGI program.
- Design pages using frames.
- Take advantage of Server Side Includes in HTML documents.
- Show an example of Server Push browser/server interaction.
- Show an example of Client Pull browser/server interaction.
- Use the *GD.pm* module to dynamically generate graphics.

Notes

Extra Path Information

- URLs can contain additional information, known as *extra path information*, just past the filename but before the query string.
- This can be used to hold an identifier.

  ```
  http://server/~s1/class/prog.cgi/getval?submit=GO
  ```

  translates into the following environment variables:

  ```
  PATH_INFO=/getval
  QUERY_STRING=submit=GO
  SCRIPT_NAME=/class/prog.cgi
  ```

- You arrange for extra path info to be sent when you specify a URL, such as a frame `SRC` attribute.

  ```
  <FRAME SRC="$script_name/query" NAME="query">
  ```

  - A CGI app can thus identify which frame the URL was sent from.

Notes

```
#!/usr/bin/perl
# File: frameset.cgi

use CGI qw(:standard);
print header;

# We use extra path info to distinguish between calls to create:
# (1) the frameset, (2) the query form,  (3) the query response

$path_info = path_info;
# If no path info is provided, then create side-by-side frameset
if (!$path_info) {
     print_frameset();  exit 0;
}

# If we get here, then create either the query form the response.
print start_html("Frameset Example");
print_query()      if $path_info=~/query/;
print_response() if $path_info=~/response/;
print end_html;

sub print_frameset {      # Create the frameset
     $script_name = script_name;
     print <<EOF;
<html><head><title>Frameset Example</title></head>
<frameset cols="50,50">
<frame src="$script_name/query" name="query">
<frame src="$script_name/response" name="response">
</frameset>
</html>
EOF
    ;
}

sub print_query {
    $script_name = script_name;
    print h1("Frameset Query");

    ### Here is the line which points to the "response" frame
    print start_form(-ACTION=>"$script_name/response", -TARGET=>"response");
    print "What's your name? ",textfield('name');
    print p, "What's the combination?",
           checkbox_group('words', ['eenie','meenie','minie','moe']);

    print p, "What's your favorite color? ",
           popup_menu('color', ['red','green','blue','chartreuse']);
    print submit;
    print endform;
}

sub print_response {
    print h1("Frameset Result");
    print start_form;
    if (param) {
       print "Your name is <i>",param(name),"</i>\n";
       print "<P>The keywords are: <i>",join(", ",param(words)),"</i>";
       print "<P>Your favorite color is <i>",param(color),"</i>";
    } else {
       print "<b>No query submitted yet.</b>";
    }
}
```


Frames

- Frames allow what look and behave like several independent pages to be contained in a single page.
- Each frame (subwindow) contains a separate document.
- Frames are created with HTML tags.

```
<FRAMESET>  ...  </FRAMESET>
```

- The data in a frame is automatically requested by the browser when it receives a `<FRAME>` tag:

```
<FRAME src="frame_source.html" name="frame_1">
```

- Individual frames can be communicated with independently through the different URL's.
- *CGI.pm* does not provide specific methods for frames.

Notes

SSI was a first attempt to get CGI-like functionality from a web server. It can be useful if handled carefully.

The SSI directives are as follows:

Command	Parameter	Description
`echo`	`var`	Inserts value of special SSI variables and other Environment variables.
`include`		Inserts text of document into HTML file stream.
	`file`	Pathname relative to current directory.
	`virtual`	Virtual path to document on the server.
`fsize`	`file`	Inserts the size of the specified file.
`flastmod`	`file`	Inserts the last modification date and time of the specified file.
`exec`		Executes SSI application and inserts app. output in the current HTML data stream.
	`cmd`	Any application on the host.
	`cgi`	CGI application.
`config`		Modifies various aspects of SSI.
	`errmsg`	Default error message.
	`sizefmt`	Format for size of file.
	`timefmt`	Format for dates.

As you can guess from some of the directives, one of the intended reasons for SSI was to return file statistics to the browser. It is still used for this, although pure CGI can do this in much more detail.

Server Side Includes (SSI)

- Server Side Includes are directives placed in HTML documents that cause the server to output data or execute programs.
- When documents are read by the server, they are parsed, looking for SSI directives.
  - If found, the server follows the directive(s) and embeds the result in the middle of the HTML string sent to the browser.
- SSI `exec` programs are similar to CGI scripts, with several differences:
  - They are external programs, but they are not (necessarily) CGI scripts.
  - They are run on demand by the server, but not based on the URL.
    - The link to them is embedded in the HTML text that the server reads from an HTML document.
- SSI applications can affect performance: once enabled, the server must parse all HTML documents it handles from then on.
- There are security risks associated with SSI: it is possible for someone with access to the server to insert SSI applications which circumvent normal security.
- SSI directives cannot be generated by CGI applications: only static files are parsed for directives.

Notes

As has been mentioned, you must be careful with SSI programs. Since a server with SSI enabled will parse its HTML files for directives, it is possible for CGI applications which allow HTML entry strings from the browser to inadvertently give access to malicious users. Often, guestbook type applications allow users to enter HTML statements which will be parsed when they re-access the server. If someone were to enter:

```
<--#exec cmd="/bin/rm -r *" -->
```

a great deal of damage could occur the next time the entry is accessed!

Speaking of enabling SSI, here are the steps necessary for doing just that for Apache:

1. You must set the extension name of files to be parsed in the server configuration file.

   ```
   AddHandler server-parsed .html
   ```

   enables parsing of all files with the extension *.html*. Some web server administrators prefer to use a special extension (such as *.shtml*) for parsed files, leaving regular *.html* files unparsed to reduce the load on the server.

   ```
   AddType text/html .shtml
   AddHandler server-parsed .shtml
   ```

2. You must tell the server what features you want to enable for a given set of files, either in the server configuration file or in a per-directory access file:

   ```
   Options Includes ExecCGI
   ```

`Includes`	Allows all SSI directives.
`IncludesNoExec`	Allows SSI directives except `exec`.
`ExecCGI`	Allows execution of CGI programs in the given directory.

Remember that both security and performance are affected by SSI.

The exec command

- SSI programs are executed with the `exec` command.

```
<!--#exec cmd="/class/homepage_counter.pl" -->.
```

- In this case, the server is being told:
  - To insert the output of this program.
  - That the program is located at the pathname */class*.
  - It is named *homepage_counter.pl*.
- The output data stream from *homepage_counter.pl* will be inserted into the data being read from the HTML document at this point.

Notes

```
<!-- index.html -->
<HTML>
<HEAD>
<TITLE>My Big Home Page</TITLE>
</HEAD>
<BODY>
<H2 ALIGN="CENTER">Welcome To My Big Home Page</H2>
<HR>
Guess how many people have been here.
<BR>
This many: <!--#exec cgi="homepage_counter.cgi" -->
<br>
Guess how big the file is that counts hits.
<BR>
<!--#config sizefmt="bytes" -->
It is <!--#fsize file="WWW/count.txt" --> bytes.
</BODY>
</HTML>
```

A Page Hit Counter Using SSI

- Page hit counters are a common exercise for beginning web programmers.
  - A simple program maintains a number in a file.
  - An SSI in an HTML page runs the program, which increments the number and includes the new value in the page.

```
#!/usr/bin/perl
#File: homepage_counter.cgi

print "Content-type: text/plain","\n\n";

$count_file = "WWW/count.txt";

if (open (COUNTFILE, $count_file)) {
   $accesses = <COUNTFILE>;
   close (COUNTFILE);
   if (open COUNTFILE, ">".$count_file) {
      $accesses++;
      print COUNTFILE "$accesses\n";
      close(COUNTFILE);
      print $accesses;
   }
   else # Open for write failed
   {
      print "Can't write to the counter file!\n";
   }
}
else  # Initial open failed
{
   print "Can't read from $count_file!\n";
}
```

Notes

This type of capability was originally intended to allow things such as the automatic re-direction of URL's from one location to another, or to allow the autonomous re-painting of a screen. The basic HTML definition allows for Client Pull, where the browser, after waiting x number of seconds, re-submits a query.

While it can be used for animating screens, often by repeatedly sending a sequence of images, it has several drawbacks.

First, since entire browser/server transactions are involved, the time it takes to transfer the data to the browser can be substantial. True animation, as in a cartoon, will not be possible, as it can take several seconds for each "frame" to be sent.

True animation requires that the images creating the animation be produced locally. This can be done with Java or special plug-in applications that run multimedia type video on the browser.

Server Push, where the server itself continues to send data repetitively, is supported only by Netscape.

Animation Description

- It is sometimes desirable to force data transfer or URL traversing from within an application.
- This can be done from either the Server or the Client end.
- The control (at least to set up the process) comes from the server side.
- Depending on the method used, the actual re-painting (data transfer) is controlled from the client or the server.
- This can be used to perform a type of animation, where successive screens are painted to the browser over and over again.
- Remember that each screen is the result of a complete transfer between the server and the browser. This will include all network delays, which can be substantial.

Notes

```
#!/usr/bin/perl
# File: nph-push.cgi

$| = 1;  # Disables buffering

use CGI qw(:standard);

$boundary_string = "\n"."--End"."\n";
$end_of_data = "\n"."--End--"."\n";

$delay = -1;
if (param('delay')) { $delay=param('delay'); }

print "HTTP/1.0 200","\n";

if ($delay > -1) {
    $delay=param('delay');
    $iterations=param('iterations');
    print "Content-type: multipart/x-mixed-replace;boundary=End","\n\n";
    print $boundary_string;

    for ($i=1;$i<=$iterations;++$i) {
        print header;
        print start_html;
        print "Push #$i\n";
        print p, `ps -ef|wc -l`, " processes on ", server_name;
        print $boundary_string;
        print end_html;
        sleep ($delay);
    }
    print $end_of_data;
}
else {
    print header;
    print start_html("Push Example");

    $server = server_name;
    print "<h3>Repeatedly show number of processes on $server</h3>\n";
    print "<hr>\n";

    print startform;

    print "Enter the number of seconds between server pushes: ";
    print textfield('delay','2','2','2'), br;
    print "Enter the number of server pushes: ";
    print textfield('iterations','2','2','2');

    print "<hr>\n";
    print submit('Start');
    print end_form;
    print end_html;
}
```


Netscape's Server Push

- Server Push allows a server to repeatedly send pages to the browser at intervals.
  - The CGI application sends one page, waits, and then sends the next page.
- A simple server push application will use a loop to repeatedly do its job.
- More complex apps can send multiple pages using application-determined order and intervals.
- While reasonably simple to implement, remember that only Netscape browsers will support this type of animation.
- Note that the program on the facing page is named *nph-push.cgi*.
  - "nph" means *No Parse Headers*.
  - If a CGI app file name starts with `nph-`, then *httpd* will not parse any data being sent from the app to the browser, nor will it provide a response line or headers.
  - Note that an nph program must send a response line.

    ```
    print "HTTP/1.0 200","\n";
    ```

  - This line must be sent by the Perl program because *httpd* will not send it.
    - It would if the program's name wasn't *nph-something*.

Notes

This example of client pull is fairly complex. It actually creates a separate Perl CGI application, and then tells the browser to run that CGI application repeatedly.

```
#!/usr/bin/perl
# File: pull.cgi

use CGI qw(:standard);

print header;
print start_html("Client Pull Program");

if (!param) {    # Paint the form
   $server = server_name;
   print "<h3>Repeatedly show number of processes on $server</h3>\n";
   print "<hr>\n";

   print startform;
   print "Enter the number of seconds between client pulls: ";
   print textfield('delay','2','2','2');
   print "<hr>\n";
   print submit('Execute Command');
   print endform;

} else {     # Get information submitted
   $delay = param('delay');

   # Dynamically create the client pull document
   $filename = "pull_command.cgi";
   open (PULL,">$filename");

   print PULL "#!/usr/bin/perl","\n";
   print PULL "# File: $filename","\n";
   print PULL 'print "Refresh: ",',$delay,',"\n";',"\n";
   print PULL 'print "Content-type: text/plain\n\n";',"\n";
   print PULL 'print "Executing ps -ef|wc -l every ",$delay,' seconds";',
              "\n";
   print PULL 'print "\n\nProcesses: ",`ps -ef|wc -l`;',"\n";
   close (PULL);

   # Kick the browser to run our new program repeatedly
   print "<META HTTP-equiv='Refresh' CONTENT='0; URL=pull_command.cgi'>\n";
}
print end_html;
```


Client Pull

- The HTTP “Refresh” header tells a browser to reload a page after a certain number of seconds.
  - This is often used to set up re-direction to a new URL, as in:

```
Refresh: 5; URL=http://foo.bar.org/index.html
Content-type: text/plain

This URL has changed. You will be re-directed
in 5 seconds to http://foo.bar.org/index.html
```

  - The URL in the "refresh" is optional; without it, the browser will re-load the current URL.
- You can use a `META` tag in the document itself, instead of a header:

```
<META HTTP-EQUIV='Refresh' CONTENT='5';
 URL="http://foo.bar.org/index.html">
```

- You can use this to have the browser re-request a CGI program.
- The “Refresh” header sets up the timer:

```
Refresh: 2; URL=http://server/cgi-bin/pullit.cgi
```

  - When the web browser sees the “Refresh” header, it starts an internal timer which waits for a prescribed time period, and then automatically re-requests the URL.
  - Only one “timed refresh” is done; to get repeated refreshes, each data stream sent to the browser must include a new “Refresh” header.

Notes

Following is an example CGI application which uses *GD.pm* to draw an analog clock.

```
#!/usr/bin/perl
#File:  gd_clock.cgi

use GD;

$| = 1;
print "Content-type: image/gif", "\n\n";

$max_length = 150;
$center = $radius = $max_length / 2;
@origin = ($center, $center);
$marker = 5;
$hour_segment = $radius * 0.50;
$minute_segment = $radius * 0.80;
$deg_to_rad = (atan2 (1, 1) * 4) / 180;

$image = new GD::Image ($max_length, $max_length);

$black = $image->colorAllocate (0, 0, 0);
$red = $image->colorAllocate (255, 0, 0);
$green = $image->colorAllocate (0, 255, 0);
$blue = $image->colorAllocate (0, 0, 255);

($seconds, $minutes, $hour) = localtime (time);
$hour_angle = ($hour + ($minutes / 60) - 3) * 30 * $deg_to_rad;
$minute_angle = ($minutes + ($seconds / 60) - 15) * 6 * $deg_to_rad;

$image->arc (@origin, $max_length, $max_length, 0, 360, $blue);

for ($loop=0; $loop < 360; $loop = $loop + 30) {
       local ($degrees) = $loop * $deg_to_rad;

       $image->line ($origin[0] + (($radius - $marker) * cos ($degrees)),
                     $origin[1] + (($radius - $marker) * sin ($degrees)),
                     $origin[0] + ($radius * cos ($degrees)),
                     $origin[1] + ($radius * sin ($degrees)),
                     $red   );
}

$image->line (@origin,
              $origin[0] + ($hour_segment * cos ($hour_angle)),
              $origin[1] + ($hour_segment * sin ($hour_angle)),
              $green   );

$image->line (@origin,
              $origin[0] + ($minute_segment * cos ($minute_angle)),
              $origin[1] + ($minute_segment * sin ($minute_angle)),
              $green   );

$image->arc (@origin, 6, 6, 0, 360, $red);
$image->fill ($origin[0] + 1, $origin[1] + 1, $red);
print $image->gif;

exit(0);
```


The GD.pm Module

- It is possible to have a CGI application create pictures dynamically (or modify existing ones) at runtime.
- Such images must be in a format displayable by the browser.
- There is a C library available called *gd* (for Graphics Draw), written by Thomas Boutell, which allows the creation and manipulation of PNG files.
- It has been built into a Perl module called *GD.pm*, which allows the *gd* functions to be invoked from inside Perl.
- It has tremendous applicability to CGI:
  - Creating graphical pages (charts, graphs, etc.) from database information.
  - Drawing maps and diagrams on the screen dynamically.
  - Manipulating PNG files under user control (zoom, rotate, crop, change color, add graphics, etc.).
- The program on the facing page uses *GD.pm* version 1.19. As of version 1.20, *GD.pm* creates PNG files instead of GIF files.
  - This avoids patent issues associated with the GIF format.
  - PNG is supported by Netscape Navigator 4.04 and higher, and by Microsoft Internet Explorer 4.0 and higher.
  - Some *GD.pm* GIF methods have changed.

Notes

Labs

1. Try modifying some of the attributes of the frames in the `frameset.cgi` example. Can you make one frame bigger than the other? Can you make frames that overlap? Try making the objects in the frames larger than one screen in the browser. What happens?

2. Verify that Server Side Includes cannot be instigated by a CGI program. (Hint: Try to trigger `homepage_counter.cgi` from within a CGI program.)

3. Modify `homepage_counter.cgi` to check if the count file exists and if not then create it. (Solution: *hpcheck.cgi*)

4. Enhance the server push example in this chapter to let the user choose from a list of commands to execute repeatedly, instead of hardcoding the process status. Provide choices for how many processes, who is logged on, and the date/time. (Solution: *nph-push1.cgi*)

5. Repeat #4 fot the client pull example.

6. Experiment with *GD.pm*'s capabilities. Create an HTML file from *GD.pm*'s POD.

Notes