# CHAPTER 8

# Using HTTP-LIB.PL

## OVERVIEW

Most people think that retrieving and displaying HTML documents over the Internet is the exclusive realm of Web browsers. But you might want a Perl program to go out on the Internet, grab HTML documents, and let you manipulate the information before displaying it. **HTTP-LIB.PL** is a set of routines that allows you to do that.

**HTTP-LIB.PL** allows you to perform two main functions. First, it can connect to a Web server on the Internet and retrieve any HTML document; by default, it strips out the header information that the Web server sends with the document. In addition, the HTTP library can submit form variables using either the `POST` or the `GET` method of CGI.

# Installation and Usage

## Using HTTP Lib

HTTP-LIB has two main functions that are designed to be called by other programs. `HTTPGet` and `HTTPPost` take the same four parameters and return a scalar variable that contains the HTML output by the server that they are connected to. The parameters are the URL, the server, the port number to connect to, and a reference to a scalar string that contains form data.

The string that contains the form data must be in the same format as a string that would be passed as form data if we were using the `GET` method: All the key and value pairs must be separated by `&`, and the key and value pairs must not have any illegal characters. Spaces are allowed only because there is a routine to automatically convert spaces to `%20` codes within HTTP-LIB. The following code snippet would post data "test=test1" and "last_name=smith" to **http://www.eff.org/~eritct/Scripts/http.cgi**. `$buf` is returned with the output from the `HTTPGet` command.

```
$buf = &HTTPGet("/~erict/Scripts/http.cgi",
     "www.eff.org",80,"test=test1&last_name=smith");
```

As with the sockets-based E-mail program in Chapter 7, there is a global operating system variable called `$http_os`. By default, this variable is set to "UNIX." If you are using an NT server version of Perl that does not support the `SELECT()` command, you should set this variable to "NT." In addition, if you are using a version of Perl that does not support the sockets package (**Socket.pm**), remove the line that says `use Socket` and replace the `$AF_INET` and `$SOCK_STREAM` variable assignment values with 2 and 1, respectively. On some systems, these values may be different. For example, on Solaris, `$SOCK_STREAM` should be set to 2.

## Example of HTTP-LIB.PI in Action Using SAMPLE.CGI.

HTTP-LIB includes a set of CGI files that shows an example of how the library works. The files are **sample.html, sample.cgi, test1.cgi, and test2.cgi**.

## SAMPLE.HTML

The following code listing, **SAMPLE.HTML**, contains a reference to **test1.cgi** and **test2.cgi**. Each CGI program uses a different method to connect to the server as a browser and send sample form data to **SAMPLE.CGI** (see Figure 8.1).

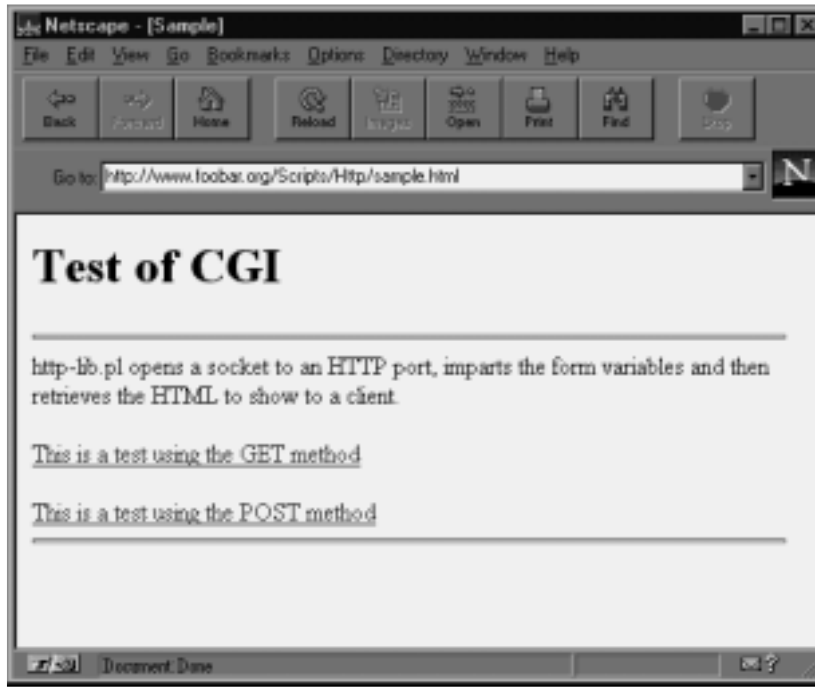


***Figure 8.1*** *HTML form testing* ***HTTP-LIB.PL****.*

```
<HTML><HEAD><TITLE>Sample</TITLE></HEAD>
<BODY>
<H1>Test of CGI</H1>
<HR>
http-lib.pl opens a socket to an HTTP port, imparts the
form variables, and then retrieves the HTML to show to
a client.
<p>
```

```
<A HREF=test1.cgi>This is a test using the GET
method</A><p>
<A HREF=test2.cgi>This is a test using the POST
method</A>
<HR>
</BODY></HTML>
```

## test1.cgi

**test1.cgi**, shown next, follows the standard format of a simple CGI program except that it `requires` HTTP-LIB and uses the `HTTPGet` subroutine to pass the value `what` to the form variable `whatever`. Finally, it takes the output from `HTTPGet` and prints it to the user's browser. Figure 8.2 shows an example of this output. The **test2.cgi** program is identical to **test1.cgi** except that it uses the `HTTPPost` function instead of `HTTPGet`.

```
#!/usr/local/bin/perl

#
#  This is a sample cgi program
#  to demonstrate that the http-lib.pl
#  library works.
#

require "cgi-lib.pl";
print &PrintHeader;
&ReadParse;

require "http-lib.pl";

$in = "whatever=what";
$buf = &HTTPGet("/scripts/Http/sample.cgi",
     "www.eff.org",80,$in);

print $buf;
```

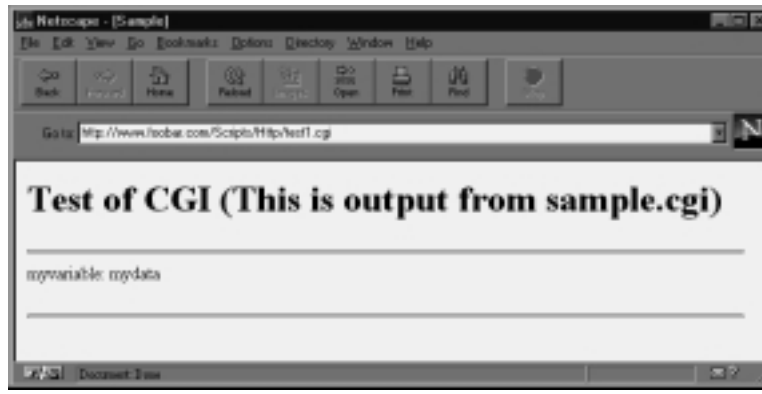


***Figure 8.2*** *Sample output from* ***test1.cgi*** *using* ***HTTP-LIB.PL****.*

## DESIGN DISCUSSION

The magic behind HTTP-LIB is done using sockets programming. Basically, *sockets* is another word for Telnet except that, instead of using Telnet to connect to a UNIX server and run programs, you are using sockets to connect to a service such as an E-mail service (SMTP, POP) or an HTML service (HTTP). Whenever a Web browser connects to a Web server, it "plugs" itself into the Web server's "socket." HTTP-LIB operates by acting like a miniature Web browser: It connects itself to a Web server, retrieves the HTML data, and returns it to your CGI program. The CGI program then displays or otherwise deals with the HTML data.

### Main Script

By default, we use the **Socket.pm** package in Perl 5. If you are not using Perl 5 or lack the **Socket.pm** file, you can delete the following line and

set the $AF_INET and $SOCK_STREAM variables manually in the Perl program according to the instructions in the usage section of this chapter. $http_os should be set to "NT" if you are using a Windows NT version of Perl. $http_os is set to "UNIX" by default.

```
use Socket;
$http_os = "UNIX";
```

## HTTPGet Subroutine

The HTTPGet function connects to the Web server using the GET method. This is the normal method that is used when a location is typed into a Web browser manually. The routine uses the URL, hostname, port, and a string containing the urlencoded form variables. If the string is blank, the HTTPGet function acts like a normal Web browser and retrieves the document specified by the URL path.

Before we delve into the this routine though, lets step back and look at an example of how a Web browser communicates with the Web server. If you went to the http://www.foobar.com/index.html URL, the Web browser is actually connecting to port 80 of www.foobar.com and attempting to retrieve index.html in the root directory. The Web browser translates this into the following request:

```
GET /index.html HTTP/1.0
Accept: text/html
Accept: text/plain
User-Agent: Mozilla/1.0
```

In response to this, the Web server would return a header followed by the HTML document. If the retrieval of the document is a success, the success code (200) is returned as part of the header. If an error occurs, a three digit error code will be sent instead of "200".

```
HTTP/1.0 200 Document follows
Date: Wed, 10 Jul 1996 01:59:51 GMT
Server: NCSA/1.4.1
```

```
Content-type: text/html
Last-modified: Wed, 10 Apr 1996 18:15:28 GMT
Content-length: 10

<HTML>
<HEAD>
<TITLE>FooBar Page</TITLE>
</HEAD><BODY>
Nothing To Say Really
</BODY></HTML>
```

HTTP-LIB.PL operates by sending the specific HTTP document request directly to the Web server. Then, when the document is returned with the header as shown above, HTTP-LIB.PL strips off the document information header and returns just the HTML code back to you.

```
sub HTTPGet {
    local($url, $hostname, $port, $in) = @_;
    local($form_vars, $x, $socket);
    local ($buf);
```

The first thing we need to do is to open the socket; here, we call a subroutine further down in the program. A socket consists of a hostname and a port number. Generally, the HTTP port number is 80.

```
    $socket = &OpenSocket($hostname, $port);
```

The form variables that are passed to this routine must be sent so that there are no illegal characters in the URL. Illegal characters are ones that might be mistaken for other things, such as a forward slash or a space. The `&FormatFormVars` converts spaces to a `%20` code.

```
    $form_vars = &FormatFormVars($in);
# we need to add the ? to the front of the passed
# form variables
```

The `$url` is set equal to the URL plus the form variables separated with a `"?"`:

```
    $url .= "?" . $form_vars;
```

Finally, we print the `GET $url` information to the Web server that we are connected to. After this information is sent, the Web server should send back the document with a success code (200) or an error code with the appropriate error message.

```
# The following sends the information
# to the HTTP Server.
    print  $socket <<__END_OF_SOCKET__;
GET $url HTTP/1.0
Accept: text/html
Accept: text/plain
User-Agent: Mozilla/1.0

__END_OF_SOCKET__
```

`&RetrieveHTTP` returns the HTML code that the server outputs in response to the `GET` method. Finally, the `$buf` variable that contains the HTML code is returned to the program that called `HTTPGet`.

```
   # RetrieveHTTP retrieves the HTML code
    $buf = &RetrieveHTTP($socket);
    $buf;

} #end of HTTPGet
```

## HTTPPost Subroutine

The `HTTPPost` function connects to the Web server using the `POST` method, the preferred method that CGI programs use to pass form variables. Instead of passing the variables as command-line parameters to the URL, `HTTPPost` passes the variables as part of the `STDIN` stream of the Web server. Thus, the Web server can read the variables as part of the normal `STDIN` input. The routine uses the URL, hostname, port, and a string containing the urlencoded form variables. If the string is blank, `HTTPPost` acts like a normal Web browser except that the content length of the `POST` is zero.

```
sub HTTPPost {
    local($url, $hostname, $port, $in) = @_;
```

```
    local($form_vars, $x, $socket);
    local ($buf, $form_var_length);
```

As with HTTPGet, we open the socket connection to the Web server specified by the $hostname and $port variables. HTML is usually served through port 80.

```
    $socket = &OpenSocket($hostname, $port);
# The following sends the information to the HTTP Server.
```

Again, the form variables need to be formatted properly.

```
    $form_vars = &FormatFormVars($in);
```

We need the length of the final form variables in order to pass the content-length information to the Web server for our POST information.

```
$form_var_length = length($form_vars);
```

The following code sends the POST command to the Web server along with the content-length; the POSTed variables are printed at the end as part of the normal output stream. The variables should be equal in size to content-length.

```
    print $socket <<__END_OF_SOCKET__;
POST $url HTTP/1.0
Accept: text/html
Accept: text/plain
User-Agent: Mozilla/1.0
Content-type: application/x-www-form-urlencoded
Content-length: $form_var_length

$form_vars
__END_OF_SOCKET__
```

Finally, the HTML code is retrieved into the $buf variable and the information is returned to the program that called HTTPPost.

```
    $buf = &RetrieveHTTP($socket);
    $buf;

} #end of HTTPPost
```

## FormatFormVars Subroutine

The `FormatFormVars` function converts the spaces in the form variables to their special encoded form of `%20`. Spaces are not recognized as valid characters within variables in a URL.

```
sub FormatFormVars {
    local ($in) = @_;

    $in =~ s/ /%20/g;

    $in;
} # FormatFormVars
```

## RetrieveHTTP Subroutine

`RetrieveHTTP` retrieves the HTTP output and returns it to the `HTTPGet` and `HTTPPost` functions.

```
sub RetrieveHTTP {
    local ($socket) = @_;
    local ($buf,$x, $split_length);
```

First, the `read_sock` function is called with a timeout value of 6 seconds to get the initial 1024 bytes of the HTML document. The returned buffer is tested to see whether it contains a 200. If it does, we figure that the success (200) status code has been sent, so we continue reading the rest of the socket as if it were a normal input file. The rest of the HTML document is appended to `$buf`.

```
    $buf = read_sock($socket, 6);

    if ($buf =~ /200/) {
        while(<$socket>) {
            $buf .= $_;
        }
    }
```

We strip off the HTTP header by looking for two newlines or two newlines preceded by carriage returns. Web servers do not always use the same standard header delimiters.

```
    $x = index($buf, "\r\n\r\n");
    $split_length = 4;

    if ($x == -1) {
        $x = index($buf, "\n\n");
        $split_length = 2;
    }
#
# The following actually splits the header off
    if ($x > -1) {
        $buf = substr($buf,$x + $split_length);
    }
```

Finally, the socket is closed and we return the buffer containing the HTML code to `HTTPGet` or `HTTPPost`.

```
    close $socket;

$buf;
} # End of RetrieveHTTP
```

## OpenSocket Subroutine

The `OpenSocket` function opens the connection to the Web server.

```
sub OpenSocket {
    local($hostname, $port) = @_;

    local($ipaddress, $fullipaddress, $packconnectip);
    local($packthishostip);
    local($AF_INET, $SOCK_STREAM, $SOCK_ADDR);
    local($PROTOCOL, $HTTP_PORT);
```

The following variables are set using values defined in the **Socket.pm** library. If your version of Perl does not have the sockets library, you can substitute default values, such as 2 for `AF_INIT` and 1 for `SOCK_STREAM`. If 1 does not work for `SOCK_STREAM`, try using 2. `$SOCK_ADDR` is a special variable that holds the IP address, port, and other relevant socket information.

```
    $AF_INET = AF_INET;
    $SOCK_STREAM = SOCK_STREAM;
```

```
    $SOCK_ADDR = "S n a4 x8";

    $PROTOCOL = (getprotobyname('tcp'))[2];

    $HTTP_PORT = $port;
    $HTTP_PORT = 80 unless ($HTTP_PORT =~ /^\d+$/);
    $PROTOCOL = 6 unless ($PROTOCOL =~ /^\d+$/);
```

We call the `gethostbyname` function to convert the hostname we are connecting to into a real IP address. `$fullipaddress` is the ASCII equivalent of the IP address delimited with periods. We calculate this in case we want to trouble-shoot whether the hostname lookup actually worked. `$packconnectip` is the host to connect to, given in the special `$SOCK_ADDR` format. `$packthishostip` is also in the `$SOCK_ADDR` format and designates this local host.

```
# Ip address is the Address of the
# host that we need to connect to
    $ipaddress = (gethostbyname($hostname))[4];

    $fullipaddress =
      join (".", unpack("C4", $ipaddress));

    $packconnectip = pack($SOCK_ADDR, $AF_INET,
             $HTTP_PORT, $ipaddress);
    $packthishostip = pack($SOCK_ADDR,
                $AF_INET, 0, "\0\0\0\0");
```

The socket first needs to be created. Then we bind the socket to the local host (`$packthishostip`) and connect the bound socket, using the connect statement, to the address designated by `$packconnectip`.

```
socket (S, $AF_INET, $SOCK_STREAM, $PROTOCOL) ||
     &web_error( "Can't make socket:$!\n");

bind (S,$packthishostip) ||
     &web_error( "Can't bind:$!\n");

connect(S, $packconnectip) ||
     &web_error( "Can't connect socket:$!\n");
```

We select the socket handle and issue the `$| = 1` command to turn off buffering. Basically, we want all commands that get transmitted to the socket to be sent right away. `STDOUT` is then selected as the current default output. The handle to the socket is returned to the routine that called `OpenSocket` so that other routines can use the socket.

```
    select(S);
    $| = 1;
    select (STDOUT);

S;
} # End of OpenSocket
```

The `read_sock` command reads the next 1024 characters waiting to be read in the socket. It gets passed a handle to the socket to be read along with a timeout value in seconds. If there is no input on `read_sock` in that many seconds, `read_sock` returns with nothing in the read buffer.

```
sub read_sock {
    local($handle, $endtime) = @_;
    local($localbuf,$buf);
    local($rin,$rout,$nfound);
```

`$endtime` is the timeout. We set `$endtime` to `$endtime + time` and later keep looping until `time` is greater than `$endtime` + the original time that `read_sock` was called. Additionally, we clear the buffer.

```
# Set endtime to be time + endtime.
    $endtime += time;

# Clear buffer
    $buf = "";
```

`$rin` is set to be a vector of the socket file handle. Basically, this Perl code sets up `$rin` as input to a special form of the `select()` function that checks whether there is input waiting to be read from the file handle. Because the `read` command blocks the operating system from doing anything else if there is nothing to read, it makes sense that we want to check for this condition first.

Note that the Windows NT version of Perl does not support the `select` statement in this form, so we read the socket even if it happens to block us on Windows NT.

```
# Clear $rin (Read Input variable)
    $rin = '';
# Set $rin to be a vector of the socket file handle
    vec($rin, fileno($handle), 1) = 1;

# nfound is 0 since we have not read anything yet
    $nfound = 0;
```

The following code loops until there is something to read or we time out. If there is something read, `$nfound` equals the number of bytes waiting to be read.

```
read_socket:
while (($endtime > time) && ($nfound <= 0)) {
# Read 1024 bytes at a time
    $length = 1024;
# Preallocate buffer
    $localbuf = " " x 1025;
     # NT does not support select for polling to see if
     # There are characters to be received.
     # This is important because we don't want to block
     # if there is nothing being received.
    $nfound = 1;
    if ($http_os ne "NT") {
# The following polls to see if
# there is anything in the input
# buffer to read.  If there is, we
# will later call the sysread routine
     $nfound = select($rout=$rin, undef, undef,.2);
         }
}
```

If we have found characters waiting to be read as part of the `$nfound` command, we read the socket using the `sysread` command. Finally, the contents of the buffer are returned to the routine that called the `read_sock` function.

```
    if ($nfound > 0) {
     $length = sysread($handle, $localbuf, 1024);
```

```
     if ($length > 0) {
         $buf .= $localbuf;
         }
    }

$buf;
}
```

## web_error Subroutine

`web_error` is a subroutine whose purpose is to print errors as HTML before exiting the program. Most Perl programs use the `DIE` subroutine to exit the program prematurely. Unfortunately, using `DIE` typically does not print any useful information to the user attempting to set up the script. Thus, we use `web_error` to print the message as HTML and then call `DIE`. The `DIE` subroutine in Perl is still useful for a Web server, because the message given by `DIE` is generally printed to the Web server's error log.

```
sub web_error
{
local ($error) = @_;
$error = "Error Occurred: $error";
print "$error<p>\n";

# Die exits the program prematurely
# and prints an error to stderr

die $error;

} # end of web_error
```