Open Computing "Hands-On" Tutorial: October 1994

Making Web Browsers Talk Back

The World Wide Web, accessed through forms-capable browsers like Netscape and Mosaic, can be used for two-way communication. Here's how to create forms and scripts for collecting information.

By Patrick M. Ryan

The World Wide Web (WWW) is an excellent tool not only for retrieving information from remote sites but also for allowing you to interact with sites in a way similar to transaction processing. Several Web browsers, most notably Netscape and Mosaic, let you enter information into the browser and have that information sent back to a server.

The resources most commonly accessed through WWW are documents written using the Hypertext Markup Language (HTML). With HTML, portions of a document can be treated as hyperlinks (or references) to other Web resources. These elements, which are often textual and sometimes graphical, appear as highlighted objects when viewed in a browser such as Netscape. If you click a mouse or press a key related to one of these highlighted objects, the browser goes out to the net and retrieves that hyperlink.

HTML documents may be retrieved from machines that are running the Hypertext Transfer Protocol (HTTP) daemon. This daemon (HTTPD) listens to a certain port (default 80) for requests for documents within a certain domain on the host system. Mosaic and HTTPD are both products of the National Center for Supercomputing Applications (NCSA).

(See the June 1994 *Open Computing* ‘‘Hands-On’’ section tutorial article, ‘‘Riding the Internet Wave’’ for setting up Mosaic.)

HTML also allows the reader to enter information into the HTML document and have that information passed back to the server machine's HTTP daemon. These types of HTML documents are called forms. The method for passing the information back and processing that information is called the Common Gateway Interface (CGI). Associated with an HTML form is a CGI program or script. The CGI specification describes what CGI programs can expect from standard input, what they should send to standard output, what environment variables they can use, and what may appear on the command line.

Nearly all current browsers support forms. Netscape Navigator and versions of Mosaic later than 2.0 support forms. The research for this tutorial used Mosaic for X version 2.4 and httpd 1.1.

There are many things to learn about implementing HTML documents, but some important items you should learn about include configuring Web server (HTTP daemon program), how to write HTML forms, how the server and CGI program interact, and what security measures to keep in mind.

Server Configuration

Implementing a form system using HTML requires you to write two files: an HTML document and a CGI program to process the input from the form. Both Listing 1 and Listing 2 demonstrate a simple product-ordering system used by the prolific and fictitious Yoyodyne Corp. We assume that their HTTP server resides on `www.yoyodyne.com`.

Configuration of the Web server daemon is a straightforward but long process and is a subject worthy of an article all to itself. The URL (Uniform Resource Locator) for NCSA's excellent documentation on HTTPD configuration can be found at the end of this article.

Once you have an operational HTTP daemon on your system, familiarize yourself with the directory structure of the server. The server has a directive named *ServerRoot* that points to the top of the HTTP daemon's directory tree (often `/usr/local/etc/httpd/`). The server-root directory has several subdirectories, including `conf/`, `icons/`, `logs/`, and `cgi-bin/`. The `conf/` directory contains the server's configuration files. In those files, most directory references are relative to the value of *ServerRoot*.

Look at the file `conf/srm.conf` (the server resource map file). The variable *ScriptAlias* indicates where CGI scripts reside. The first argument to *ScriptAlias* is an alias name (for the actual path name) that HTML forms must use to refer to their associated CGI programs. The second argument is the real path on the system where CGI scripts live. For security reasons, any attempt to reference a CGI program outside that alias directory will generate an error from the server. We need to know the actual location of that directory so that we know where to locate our CGI program.

Form Syntax

Forms are set up in an HTML document using a `FORM` tag. The syntax is `<FORM ACTION = "URL" METHOD = "METHOD"> [form text]</FORM>`. The `ACTION` attribute parameter is a URL that points to the form's CGI program. Usually but not always, this CGI program resides on the same machine as the HTML document. The *METHOD* parameter will have a value of ‘‘POST’’ or ‘‘GET.’’ This parameter indicates how the request will be transmitted to the server. In nearly all cases, it will be ‘‘POST.’’ When using the POST method, the client sends the query data as an Object-Body. The CGI program reads the data on its standard input.

As mentioned before, CGI scripts *must* reside in the directory pointed to by the *ScriptAlias* parameter. A typical value for the alias directory name is `/cgi-bin/`. The URL that points to a CGI program `process_order` would be `"/cgi-bin/process_order"`. Note that this URL does not have any protocol or host information. In the absence of such information, the Web server will look for the CGI script on the same host where the form resides.

As with CGI scripts, the locations of all resources--accessed through the Web server daemon, the documents served, or otherwise --are restricted. The *DocumentRoot* directive points to the top-level directory where these resources may be accessed. (The default value for *DocumentRoot* is `/usr/local/etc/httpd/htdocs/`.) For example, if you have a URL that points to `http://www.yoyodyne.com/products/order.html`, the HTTP daemon on `www.yoyodyne.com` will translate this path into `/usr/local/etc/httpd/htdocs/products/order.html`.

However, if the first part of the URL file path has the form `~user/` the server consults the value of the *UserDir* server configuration directive. If this directive has a directory name value, the server will look for `user`-account home directory, append the value of *UserDir*, and look in that directory for the reference. For instance, if *UserDir* is set to `public_html` and you have a reference to `http://www.yoyodyne.com/~dave/abstract.html`, the server will translate this path to `~dave/public_html/abstract.html` on `www.yoyodyne.com`. The administrator can set *UserDir* to ''DISABLED'' to defeat this feature.

HTML Buttons

An HTML form can make use of three different types of interface elements or tags: `INPUT`, `SELECT`, and `TEXTAREA`.

The general form of an `INPUT` tag is `<INPUT NAME = "`*`NAME`*`">`. The `INPUT` element is a ''standalone'' tag; it has no terminating tag. `NAME` defines the symbolic name for the field value passed back to the server upon submission and must be present for all but `TYPE="submit"` or `TYPE="reset"`. The value for `NAME` does not appear in the displayed document. Usually, any text immediately before or after the `INPUT` tag serves as a label for the tag.

The `TYPE` attribute to the `INPUT` tag indicates which type of input you want:

`text`
: Textual input.

`password`
: Same as `text` but does not echo characters.

`checkbox`
: A button that is either on or off.

`radio`
: A ''one-of-many'' checkbox if multiple radio buttons are grouped with the same `NAME`.

`reset`
: Resets form values to their defaults.

`submit`
: Sends form information back to the server.

The `text` and `password` input `TYPES` values may contain an optional `SIZE=`*`columns,rows`* attribute that indicates the number of columns (characters) and rows displayed for text input. Checkboxes and radio buttons may have an optional `CHECKED` attribute to specify a pre-checked value.

The `submit` and `reset` input `TYPE` values are special. If a user presses the ''Reset'' button, all of the inputs are set back to their default values. Pressing the ''Submit'' button will cause the browser to package up the data entered by the user and send it back to the server. These two values have an optional attribute `VALUE=`*`button-label`*, which, if present, will be used as the button label.

The `SELECT` interface tag allows the user to choose from a list of items in a pop-up menu or scrollable list. The selection items are enclosed between the opening tag `<SELECT>` and the closing tag `</SELECT>`. Each choice in the list begins with an `<OPTION>` element. The `SELECT` tag may optionally contain the `MULTIPLE` attribute to specify a scrollable list. One of the `OPTION` tags may have the `SELECTED` clause to identify the default value to be displayed. Listing 1 shows how a selection list is used.

The `TEXTAREA` element specifies a multiline input field within the tagging pair. The `NAME` attribute defines the symbolic name of the field returned to the server upon submission. The `ROWS` and `COLS` attributes determine the size of the displayed area. Other characters between the opening (<TEXTAREA>) and closing (</TEXTAREA>) tags define text to be initially displayed. When the text-area box is instantiated the user may accept, delete, or edit the initial text, if any.

Interpreting Input From the Server

Once a user presses the ‘‘Submit’’ button, the browser will send the data entered by the user back to the Web server in a compact format. The server will attempt to resolve and validate the URL parameter of the `ACTION` element. Assuming the specified program can be found and is executable, the server will ‘‘fork’’ a copy of itself which in turn ‘‘execs’’ the CGI program.

CGI programs may be written in any language. Simple ones are often written using just the Bourne or C shell. More complicated ones use C, C++, or Perl. I chose Perl for our sample CGI program because of its string-handling and associative-array capabilities.

The server communicates with the CGI program through standard input, standard output, environment variables, and the command line. The input to the form is passed to the program via standard input. The server sends a single newline-terminated line of ASCII text to the CGI program. That line consists of *`name=value`* pairs separated by the ampersand (`&`) character. The data is encoded as in a URL: spaces changed to plus (+) signs, ampersands and equal signs prefaced with backslashes, and other non-text characters encoded in hexadecimal. The *`name`* is taken from the value assigned to the `NAME` attribute of the `INPUT`, `SELECT`, or `TEXTAREA` elements.

Interpreting the CGI program input is a straightforward tokenizing task. Typically, the *`name=value`* pairs are placed into some type of hash table. In Perl, they can be placed into an associative array. I include the `cgi-lib.pl` Perl library--available from NCSA--which contains the routines shown here:

`ReadParse`

Read and parse the input from the server. After this routine has been run, several global variables will exist:

`$in`

a scalar that contains the entire input line.

`@in`

an array of strings of the form *`name=value`*.

`%in`

an associative array where `$in{`*`name`*`}=`*`value`*.

`PrintHeader`

Returns a line meant to be the first line sent back to standard output.

`PrintVariables`

Collects all of the form values and formats them into an HTML unordered list suitable for sending to standard output. This routine is useful for testing.

Once the CGI program has the input from the server in a usable format, the program can do anything it wants with it. Typical uses for HTML forms are bug reports, registration forms, product-ordering forms, and forms used to query a database. Use your imagination.

Responding to the Server

The CGI program sends information back to the server by way of standard output. The standard output of the program is interpreted according to the `Content-Type` line sent first to standard output. From the CGI program's standard output, the server will expect to see a content type setting followed by two newline characters. The `PrintHeader` routine in `cgi-lib.pl` returns `Content-type: text/html\n\n`. Although the content type may be any valid document type, HTML is usually the most useful format. This method is used to create virtual HTML documents on the fly. Bear in mind that the server collects all of the output from the CGI program and waits until the CGI program has terminated before it forwards the text back to the browser.

Writers of HTML forms make use of the CGI program's output in several different ways. Among them are:

- Displaying database query results
- Displaying characteristics of the server system (uptime, current date and time, finger, who, etc.).
- Telling the user what resulted from their input. For example, ‘‘Your order has been received,’’ ‘‘Your bug report is number 9407112.’’
- Returning error messages. For example, ‘‘I'm sorry. That item is out of stock,’’ ‘‘Your password is incorrect,’’ or ‘‘No matches were found in the database.’’

Sophisticated CGI programs may create new HTML documents and forms on the fly.

The HTTP daemon defines a set of environment variables for the CGI program that provide information about the server and about the client where the browser is running. Consult the CGI specification for a complete list of these environment variables. Perl stores environment values in an associative array called `%ENV`. Some of the more important variables are:

`CONTENT_LENGTH`
Length of input string from server in bytes

`CONTENT_TYPE`
Type of input from server

`REMOTE_HOST`
Name of host where query originated

`SERVER_NAME`
Name of machine on which server is running.

These are especially useful for logging information about who is using your form.

When setting up a new HTML form, it is wise to test the output from the form before you associate it with a CGI script that does real work. To that end, you can set your `ACTION` parameter to a URL that will simply echo information about what is being sent by the server. Listing 3 is a simple Perl script that echoes all of the *`name=value`* pairs and all of the environment variables. Alternately, you could point your script at NCSA's test script at URL: `http://hoohoo.ncsa.uiuc.edu/htbin-post/post-query`.

Security

CGI works by allowing an anonymous remote user to start up a program on an HTTP server's host machine. That simple fact should send chills down the spine of any good system administrator. HTTP has a sophisticated authentication system and is unlikely to be foiled by the novice cracker. However, a more experienced and less noble cracker might exploit vulnerabilities present in an insecure CGI program.

CGI programs often do their job by running additional programs on the server system. These programs may be fed input that originated in a submitted HTML form. Although using form input this way is a normal thing to do, a secure CGI program should carefully look at any text being passed to another program. If the CGI program sees any characters that have special meaning to the command shell, the program should either escape them with backslashes or generate an error message and terminate. Consult the manual pages for the Bourne and C shell for a complete list of these characters. Some especially dangerous characters are semicolons, backquotes, and ampersands. Note the `protect()` function in Listing 2.

Say you have an HTML form that returns a user's email address. In your Perl CGI script you copy this value to a variable named `$address`. You want to mail some information to this address so you put the following code in your Perl CGI program:

```
# Send confirmation back to the user.
open(MAIL, "|/bin/mail $address");
print MAIL
   "Your order has been processed.  Your PO number is $number\n";
close MAIL;
```

A legitimate user would enter a value like ‘‘joe@company.com’’. The second argument to Perl's `open()` function would have a value of `|/bin/mail joe@verylarge.com`. No problem.

The unscrupulous cracker could use a bogus address and then follow it with a semicolon and another command. For instance, let's say this cracker entered ‘‘/dev/null; rm -rf /home’’. Unless the CGI program notices the semicolon embedded in the `$address` variable, the `open()` function would be sent the `|/bin/mail /dev/null; rm -rf /home` argument. Here, `/bin/mail` would cheerfully send a message to `/dev/null` and the system would then treat the text after the semicolon as another command and merrily proceed to erase the contents of the `/home` directory tree.

Several languages, including Perl and most command shells, have an `eval` statement that allows them to treat arbitrary pieces of text as executable code. C and C++ programs, as well as Perl, have access to the `fork()`, `exec()`, and `system()` functions. If these functions cannot be avoided, any input they use should be thoroughly examined and sanitized.

Concluding Thoughts

HTML forms are a slick way to allow remote users to interact with your system in a clean, controlled way. Considering how sophisticated this interaction can be, HTML forms are easy to set up. All of the really hard work is done by the HTTP daemons and Web browsers.

Acknowledgments

Thanks to Rob McCool (formerly of NCSA) for writing the HTML documentation on HTTP and CGI and to Steven E. Brenner for writing the Perl routines for tokenizing CGI input. The HTML form I used

as a baseline was created by Diann Smith (Langley Research Center).

For More Information

Here are some pointers to relevant documentation available through the Web:

- NCSA's ``A Beginner's Guide to HTML'' primer document.
- NCSA's ``Mosaic for X version 2.0 Fill-Out Form Support'' document.
- NCSA's Online Documentation for HTTPD
- NCSA's CGI Specification document
- CERN's ``Uniform Resource Locators'' document

Converted to HTML and additional copy/style edit by Walter Zintz

Edited by Becca Thomas / Online Editor / UnixWorld Online / beccat@wcmh.com