HTML for Forms

CWRU

Contact(s):	Curriculum Support Group	**e-mail** edtech@po.cwru.edu	x8600
	http://www.cwru.edu/net/csg/csg.htm		
	Information Services Help Desk	**e-mail** help@po.cwru.edu	x4357 (HELP)
	http://help.cwru.edu		

1. Introduction

The submission of information on a form can add interactivity to your web pages. Forms are normally added through some kind of scripting that requires a program which resides on the Web server itself. This program is written in a language such as Perl, C/C++, TCL, AppleScript, or another Common Gateway Interface (CGI) language. The role of this CGI program is to accept the data which the user inputs and then do something with it. What it does depends on what the program has been written to do. It could e-mail the data to someone, or add an entry to a database, or write out a text file, or create a customized display, or just about anything else you can think of.

Because scripting is an extremely complex topic, this document concentrates on one aspect--the creation of a WWW form using a script built into the Aurora server--while providing a brief introduction to the vocabulary of and resources for scripting.

At the end of the lesson you will be able to:

- List advantages of server-side and client-side scripting
- Create a form with HTML tags
- Make a form work using the Aurora server's mailform CGI script
- Locate templates and generic scripts on the Internet

In order to complete this lesson successfully, you must come with the ability to:

- Write basic HTML tags
- Use e-mail

2. Vocabulary

CGI -- Common Gateway Interface, a class of programming languages which may be used in conjunction with a Web server. A CGI program is used to process the data from an HTML form, for example

client-side -- this kind of scripting involves embedding blocks of script into your HTML pages. The browser then interprets the scripts and performs the scripted activity. If the browser fails to recognize the scripting language, nothing happens.

Java -- an object-oriented programming language developed by Sun Microsystems. Small Java applications are called Java applets and can be run on a computer by a java-compatible Web browser, such as Netscape Navigator or Microsoft Internet Explorer

JavaScript -- a scripting language that is commonly used for client-side scripting, although some servers can interpret it for server-side scripts as well. JavaScript is understood by both Internet Explorer and Netscape browsers, although there are some differences. However, the European Computer Manufacturers Association created a JavaScript standard in 1997, so it is always possible that these differences will eventually disappear.

PERL -- for Practical Extraction and Report Language, a programming language developed by Larry Wall. PERL is known for its abilities to process text and is a popular language for writing CGI scripts.

server-side -- this kind of scripting uses the WWW server to process commands and perform the required activities. CGI scripts are server-side applications.

VBScript -- a scripting language similar to JavaScript but with a Visual Basic-like syntax. The major disadvantage of VBScript is that it is not understood by Netscape browsers unless a user purchases a special plug-in (and few do).

3. How do forms work?

A very common use for HTML forms is creating a feedback form, or some other user-response form. Usually, the feedback from the user is e-mailed to somebody. So how does this work? Data is passed from the HTML form to the CGI program in name-value pairs. The value is whatever the user enters, and the name is the label used to identify that input.

Here is an example. We have a form with three inputs: name, rank, and serial number. The inputs are labeled NAME, RANK, and SN, respectively. The user enters the data "Eric" for the name, "Very Low" for rank, and "123-45-6789" for the serial number. The data sent to the CGI program would look something like this:

NAME = Eric
RANK = Very Low
SN = 123-45-6789

The program would then do something with each piece of data. For those of you familiar with programming languages, you recognize what's going on here. The labels are variable names, and the data collected are the values of the variables. In other words, imagine that each input is a labelled box. In this example, the labels would be NAME, RANK, and SN. Whatever the user types in for a given box becomes the contents of that box. The boxes are then all loaded onto a truck (i.e., your connection to the Web server) which hauls them off to the CGI program. The program then unloads the contents of each box and does something with what it finds. What it does depends on how it's been programmed.

NOTE: No two inputs can share the same name. Each input has to have its own unique identifier, so that the CGI program can figure out which piece of data belongs where. So if I use the name "ssn" to refer to a Social Security Number input, and then later ask for the name of a nuclear submarine, I must use a different name. "sub" would work, as would "boat," but "ssn" is not allowed.

4. Client-Side vs. Server-Side

There are two distinct advantages to client-side scripting: you do not need a server configured for server-side scripts and the client computer -- not the server -- is burdened with the processing of the script. The distinct disadvantage is that even browsers which understand scripts (and some don't) don't all understand them the same way. What works in one browser may not work in another.

Server-side scripts, on the other hand, do not depend upon the capabilities of the browser and are thus very useful for sites accessed by a wide range of computer users. There are two possible disadvantages of server-side scripts: (1) if you are not running your own server, you will probably find your ability to use server-side scripts restricted; and (2) script processing may place too much of a burden on the server.

5. Creating a Form

There are two components in every form: the HTML that creates the appearance of the form and the script that makes it work. In other words, using HTML to create a form is less than half of the work involved in bringing interactivity to the web.

Using the `<form>` tag lets the browser know where a form begins and ends as well as where to send the data. In addition, any tag which is allowed inside of the `<body>` container is allowed inside a form. Headings, paragraphs, lists, tables, images, links-- anything and everything goes. Also, there are certain tags which are allowed to exist inside a form, and nowhere else. Finally, the `<form>` tag has

two attributes which must be used if the form is to work correctly with a CGI script. (Forms that use client-side processing may not need these attributes). The attributes are method and action.

Here's what an empty form would look like:

```
<form method="post" action="/cgi-bin/program1">
</form>
```

Method
Method has two possible values: GET and POST. If you want the data to go from the browser to a CGI program then use the method POST.

Action
This attribute contains the URL of the CGI program which processes the data sent by the browser. In the example above, the program (program1) resides in the cgi-bin directory of the server which contains the form itself. The value of action can be either a relative or a full URL.

Types of Input
The most commonly used form tag is input. There are several types of input, like open text inputs, radio buttons, and checkboxes; the type of input is specified using the type attribute. The most basic version of the tag is <input />, but like <img />, various attributes are required to make it work.

⇒ **Text**
A text input is simply a box in which anything can be typed-- letters, numbers, or anything else-- via the keyboard. In most browsers, the box is *twenty* characters wide, but this can vary. The markup and its effect:

Markup: <input type="text" name="socsec" />

Result:

While twenty characters is the default width, it is possible to specify the size of the input by using the size attribute. The value of size is a positive integer and translates into the width of the input box. This width is expressed as a number of characters. Therefore, size=″9″ means the input will be nine characters wide. The specified size does not, however, restrict the number of characters which can be input. The user can keep typing past nine characters, so this still isn't a very good way of getting a Social Security number. You can limit the number of characters which may be input by using the maxlength attribute. The maxlength value is expressed as a number of characters, just as size is.

⇒ **Password**
One obvious drawback of the text input is its openness, particularly if you need to ask for a password (or similarly sensitive piece of information, like the user's weight). If you use an <input type="text" /> tag, anyone sitting next to or behind the user will be able to read what the user types as it appears on the screen. The answer is the Password input.

Password inputs are strikingly similar to text inputs, in that they accept any input from the keyboard, they can have size and maxlength attributes, and (as always) they require names. The difference is that when the user types in a password field, the computer displays bullets or asterisks instead of the characters being typed. Thus, the user's password is kept safe from the prying eyes of his/her roommate, spouse, boss, or whomever.

A typical password input would look like this:

Markup: <input type="password" name="pwd" size=″15″ maxlength=″15″ />

Note: Be warned, however: this 'visual security' is the furthest extent of the security afforded by the password input. There is absolutely no encryption of any kind whatsoever.

⇒ Radio Buttons

Radio buttons are best used when you want the user to select one of a limited number of choices. For example, suppose you wanted to find out which computer operating system your users prefer. Of the six options provided, the user should only be able to pick one. The list will look something like this:

Figure 6.1

The markup will look like this:

```
<p>
Your favorite computer operating system:<br />
<input type="radio" name="fav_os" value="mac" />Macintosh<br />
<input type="radio" name="fav_os" value="dos" />DOS<br />
<input type="radio" name="fav_os" value="win" />Windows<br />
<input type="radio" name="fav_os" value="win95" />Windows95<br />
<input type="radio" name="fav_os" value="os2" />OS/2<br />
<input type="radio" name="fav_os" value="unix" />UNIX<br />
</P>
```

Radio buttons function so that *only one option* can be chosen. If an option is already selected, then choosing another option will de-select the previously chosen option and select the new option. Since there is nowhere for the user to enter a value, however, the value of each option must be specified in the HTML markup itself, using the `value` attribute.

⇒ Checkboxes

The HTML markup for a checkbox logical input looks very similar to that for radio buttons. The only structural difference is the use of `type="checkbox"` instead of `"radio"`. With checkboxes, however, the user can select more than one response by marking multiple checkboxes.

Using the SELECT Tag

Using a `select` list will create a pop-up list from which any one option may be selected. The advantages are that you still have a list to choose from, but it takes up very little screen space until the user interacts with the list. In this example the user is asked what kind of connection she/he's using to get her/his Internet feed. The result looks something like this:

Here's the markup:

```
<p>
How are you reaching this page?
<select name="access">
<option>No response</option>
<option>Compuserve</option>
<option>America On-Line</option>
<option>Local ISP</option>
<option>National ISP</option>
<option>Straight Internet connection</option>
<option>None o' yer beeswax</option>
<option>Other</option>
</select>
</p>
```

There are a few things to mention:

1. a name attribute must appear in the select tag; in this case, the name is access.
2. select and option are containers which require close tags (</select>, </option>)
3. the size of the list-box is as wide as the longest option in the list.
4. the browser will typically set the list to the first option
5. the select and option tags can be modified with attributes, e.g. size, multiple, and selected.

Size

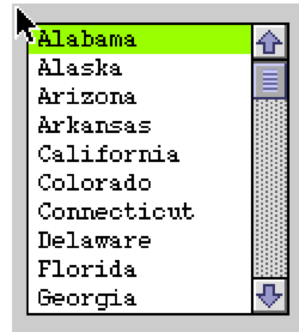
Suppose that we don't want to use a "pop-up" list, but instead want a scrollable list. Let's say you want to have your readers indicate which country they live in. Many browsers can't display a long pop-up list like this one would be. Instead, we want to use a list where we can't see the whole thing at once, but we can still scroll through it, and are able to see a portion of it at any time.

This is accomplished using the `size` attribute in the `select` tag. The number specified by `size` sets the number of lines of text which are displayed at once. I'll go with a list of the fifty United States of America, mostly because it's a shorter and much more stable list than the list of countries. The following example shows what the browser will display when restricting the size to 10 lines:

Markup:

```
<select name="state" size="10">
</select>
```

Result:



In this case, the browser has been told to show 10 lines, so we see twenty percent of the total list at any time (10 out of 50). `size` can be set to any positive integer, although going past twenty is generally discouraged.

Multiple

In order to create a list in which multiple selections can be made, simply add the `multiple` attribute to the `select` tag. This will let the user select more than one choice, and in some browsers they'll be able to do range-selections, and so on. We'll use the list of fifty states again, but this time we'll ask the user to indicate which ones s/he has visited.

Markup:

```
<select multiple name="state" size="10">
</select>
```

The result will be similar to the example above, but in this case the user can choose more than one state. If you're using a graphical browser, try scrolling around in the list and shift-clicking on different options. Then try control-clicking. Try clicking on already-selected options, using one or the other of the modifier keys. Just like checkboxes, you can select or de-select options at will.

Selected

With the `selected` attribute you can specify a default choice. Returning to the first example list, the one about access methods, let's assume that I want to make the default choice to be the last option, "Other." However, I don't want to move it to the top of the list for some reason. The markup would look like this:

```
How are you reaching this page?
   <select name="access">
   <option>No response</option>
   <option>Compuserve</option>
   <option>America On-Line</option>
   <option>Local ISP</option>
   <option>National ISP</option>
   <option>Straight Internet connection</option>
   <option>None o' yer beeswax</option>
   <option selected="selected">Other</option>
   </select>
```

Result:

How are you reaching this page?

Other

Obviously, only one `selected` should be used in a normal `select` list, whereas you could use `selected` on many different options in a `select multiple` list.

Other Useful Tags and Attributes

This last section will cover several miscellaneous tags that you can use in forms. They include: `textarea`, hidden `input`, submit, and reset.

`Textarea`

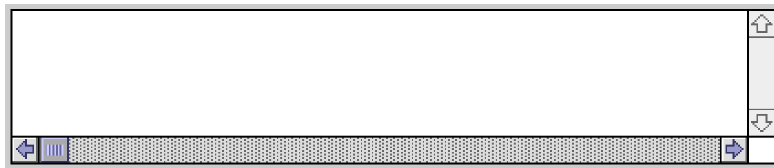
The `textarea` tag is used to create a box where the user may type large amounts of text at will. A typical use for `textarea` is to ask users to input general comments they may have about a Web site. In addition to having some special attributes, `textarea` is a container, so the close-tag is required.

The special attributes are, as you might have guessed from the above example, `rows` and `cols`. These specify the number of rows high and columns wide the textarea should be. The numbers are measured in characters, so `rows="5"` makes the box five lines high, and `cols="65"` means that the box will be 65 characters wide. This will cause the box to grow or shrink depending on the size of the monospace font set by each user.

Markup:

```
<textarea name="comments" rows="5" cols="65"></textarea>
```

Result:



Hidden input

Another type of `input` is "hidden". It does exactly what it sounds like: it allows for an input which is hidden from the user. With hidden inputs there is no way to affect the value of the `input`, which is why assigning an explicit value is important. Otherwise, the value of the input will be literally nothing. Hidden inputs are useful when you want to pass to the CGI program information which does not and should not change , but is for some reason important. Take a look:

Markup:
```
<input type="hidden" name="sendTo" value="pqs4@po.cwru.edu" />
```
Result:

Yes, the Result section does contain markup. It's just hidden from view.

Submit & Reset

What good are forms if the user can't tell the browser when to send the data off to the CGI program? And what if the user realizes s/he made several mistakes and just wants to start over instead of correcting each one of the mistakes s/he made?

That's what the `input` types "submit" and "reset" are for. Typically, these `inputs` do not need names, as they do not generate any data to be sent. Their functions are to affect the rest of the form.

First, submit inputs are the user's way of saying, "I'm done now, take this stuff I input and do that voodoo that you do so well!" The markup...

```
<input type="submit" />
```

...will create a button on the screen which says something like "Submit Query" (the actual label may vary). Selecting a Submit button triggers the posting of the input data to the CGI program. Similarly, the markup...

```
<input type="reset" />
```

...places a button which is labelled something like "Reset Form" (again, the actual label will vary). Selecting the button will cause the entire form to be reset to its default state, wiping out any and all changes made by the user.

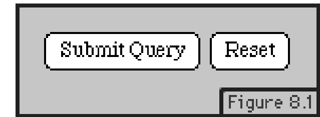


Figure 8.1

Usually, the two buttons are placed together at the bottom of a form, as in Figure 8.1, but there is nothing which says they have to go together, or at the bottom of the form. That's a matter of custom more than anything else.

Most people feel that the default labels which browsers give to these buttons are pretty boring. Fortunately, you can change those labels by using a `VALUE` attribute. For example:

```
<input type="submit" value="Send This Puppy" />
<input type="reset" value="No, Wait, It's All Wrong" />
```

will yield the following (the functions of the buttons are, of course, still the same):

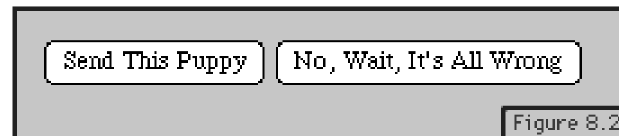


Figure 8.2

Changing Values
There is another way you can use the `VALUE` attribute with an `INPUT` tag. If you assign a value to a text or password input, then it will appear in the input box by default.

Markup: `<input type="text" name="greet" value="Hi there!" />`
Result:

The user is perfectly free to add to, alter, or completely replace the contents of the input box if she/he feels like doing so, of course. If s/he does nothing to the contents of the input, then its predefined value will be used when the form is submitted.

6. Using the Mailform CGI script

Aurora's *mailform cgi* script is used to mail data collected by a form to a particular e-mail address and to simultaneously display to the user a new page in the web browser. Complete information on using *mailform* is available at *http://www.cwru.edu/help/AuroraCGI/mailform.html*. The basic steps for using this script are:

1. In your Aurora home directory, create a cgi/ directory.
2. In the cgi directory, create a bin/ directory.
3. Copy the script from *http://www.cwru.edu/help/AuroraCGI/mailform* into the cgi/bin/ directory.
4. Create an HTML form. Make sure that your form includes fields named *mf_mailto* and *mf_fields*:
   mf_mailto—fill in the e-mail address which should receive responses from the form
   mf_fields—must contain the names of all the input fields you use

Examples:

```
<input type="hidden" name="mf_mailto" value="edtech@po.cwru.edu" />
<input type="hidden" name="mf_fields" value="dept institution phone
fax email mailaddress" />
```

5. The form tag should read: *<form method="post" action="/cgi-bin/AuroraCGI/*youraccountname/*mailform">* with the appropriate Aurora account name filled in.
6. If you want to test the form without loading it onto the server, change to *action* value to *http://www.cwru.edu/cgi-bin/AuroraCGI/*youraccountname/*mailform*.

Other Useful Fields:

Your form must include the *mf_mailto* and the *mf_fields* fields in order to work. The mailform script also recognizes a number of other field names, including the following:

mf_mailfrom— This is the e-mail address that will appear as the sender of the e-mail. This is often useful for allowing the responder to designate his/her own e-mail address. EXAMPLE: `<p>`Please enter your e-mail address: `<input type="text" name="mf_mailfrom" /></p>`

mf_subject—This is the subject line of the e-mail that will be sent. Again, you can set it to something constant, or let the responder specify it. EXAMPLES: `<p>`Please enter the subject: `<input type="text" name="mf_subject" /></p>` —or— `<input type="hidden" name="mf_subject" value="Your_important_ constant_subject" />`

mf_reqfields—This lists, by name, all the fields that must be filled in before the form can be submitted. If one is not filled out, then an error page is displayed. Each of the names of the fields should be separated by a single space, which means that none of the names can have spaces in them, or the program will think they are separate names. EXAMPLE: `<input type="hidden" name="mf_reqfields" value="field1 field3 field4 field8" />`

The Automatic Response Page:

When a form is submitted using the mailform script, a page is displayed which tells the person who submitted the form whether or not it has been successfully submitted. The maintainer of the form can control this response page by including hidden fields in the form itself.

mf_title—This sets the title of the page, both between the `<title>` and `</title>` tags and between the `<h1>` and `</h1>` tags. EXAMPLE: `<input type="hidden" name="mf_title" value="Title_of_displayed_page" />`

mf_success—This can do one of two things. First of all, it can display a message in the page that is displayed upon successful form submission. The VALUE of this should be the text of the message. EXAMPLE: `<input type="hidden" name="mf_success" value="You have successfully submitted this form. Thank you for your input." />` Secondly, this can specify the actual file that you want as the success message. This is accomplished by making the first symbol of the `value` a "/" and having the `value` be a directory structure and a filename. This directory starts from the base of your Aurora account. EXAMPLE: `<input type="hidden" name="mf_success" value="/results/forms/success1.html" />`

mf_failure—This can do one of two things. First of all, it can display a message in the page that is displayed upon a failed form submission. The `value` of this should be the text of the message. EXAMPLE: `<input type="hidden" name="mf_failure" value="Something was wrong in the submission of this form. Be sure to have all required fields filled out." />` Secondly, this can specify the actual file that you want as the failure message. This is accomplished by making the first symbol of the `VALUE` a "/" and having the `value` be a directory structure and a filename. This directory starts from the base of your Aurora account. EXAMPLE: `<input type="hidden" name="mf_failure" value="/results/forms/failure1.html" />`

mf_bodyattrib—This designates any special attributes to be specified in the `<body>` tag of the displayed page. This can be used to make the displayed page look like the rest of the pages on the site. EXAMPLE: `<input type="hidden" name="mf_bodyattrib" value=' background="pix/mybackground.gif" text="#ABC123" link="#321CBA"' />` **Note the use of single and double quotation marks!**

7. Finding and Using Scripts

For information about using CGI on Aurora, including extensive documentation on the SafePERL-CGI extension, go to *http://www.cwru.edu/help/AuroraCGI/AuroraCGI-docs.html*. Other information about PERL can be found at *http://www.perl.com/pace/pub*.

There are numerous sources of free JavaScripts on the World Wide Web. JavaScript is commonly used for client-side processing. Although you will probably need to understand how the language works to customize them, many can be used "as is" for specific purposes.

Half-Baked Software, based at the University of Victoria Language Centre, produces Hot Potatoes, a suite of six applications enabling the creation of interactive multiple-choice, short-answer, jumbled-sentence, crossword, matching/ordering, and gap-fill (Cloze) exercises for the World Wide Web. These applications produce web pages that use JavaScript for interactivity.
http://web.uvic.ca/hrd/halfbaked/

The JavaScript Source
http://javascript.internet.com/

Randy Bennett's JavaScripts
http://home.thezone.net/~rbennett/utility/javahead.htm

Knowledge Design Instructional Resources
contains information and instruction on scripting for foreign languages
http://www.auburn.edu/~mitrege/knowledge/index.html

8. For More Information

For more about forms in general see Eric Meyer's Intermediate HTML Tutorial at *http://www.cwru.edu/help/interHTML/toc.html,* from which portions of this documentation are adapted.