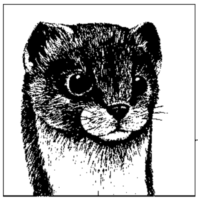


# Forms

Forms provide an interface allowing users to interact in some way with your site. In most cases, they are used to gather data, either for later use or to provide a customized response on the fly. Forms have a wide range of uses, from functions as simple as search boxes, mailing list signups, guestbooks, and surveys to as complex as online commerce systems.

Forms collect input via controls, such as buttons, text fields, or scrolling menus. Controls are placed on the page using special elements in the markup. These elements are merely an interface for collecting user information and do not actually process the data. The real work is done by forms-processing applications on the server, such as CGI scripts, ASP, ASP.NET, ColdFusion, PHP, or Java servlets.

The programming necessary for form processing is beyond the scope of this book. This chapter focuses on the frontend aspects of forms: the elements and attributes for building the form interface as well as the elements used to improve accessibility.

| | |
|---|---|
| `form` | Establishes the form |
| `input` | Creates a variety of controls |
| `button` | Generic input button |
| `textarea` | Multiline text entry control |
| `select` | Multiple-choice menu or scrolling list |
| `option` | An option within a `select` control |
| `optgroup` | Defines a group of options |
| `label` | Attaches information to controls |
| `fieldset` | Groups related controls and labels |
| `legend` | Assigns a caption to a fieldset |

## The Basic Form Element

The `form` element is used to designate an area of a web page as a form.

### form

```
<form> ... </form>
```

#### Attributes

*Core* (`id`, `class`, `style`, `title`), *Internationalization, Events,* `onsubmit`, `onreset`, `onblur`
`accept="`*`content-type-list`*`"`
`accept-charset="`*`charset list`*`"`
`action="`*`URL`*`"` *(Required)*
`enctype="`*`content type`*`"`
`method="get|post"`
`name="`*`text`*`"` *(Deprecated in XHTML in favor of* `id` *attribute)*
`target="`*`name`*`"`

The form may contain any web content (text, images, tables, and so on), but its function is to be a container for a number of controls (checkboxes, menus, text-entry fields, buttons, and the like) used for entering information. It also has the attributes necessary for interacting with the form-processing program. You can have several forms within a single document, but they cannot be nested, and you must be careful they do not overlap.

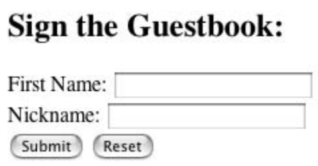
When the user completes the form and presses the “submit” button, the browser takes the information, arranges it into name/value pairs, encodes the information for transfer, and then sends it off to the server.

Figure 15-1 shows the form resulting from this simple form markup example.

```
<h2>Sign the Guestbook:</h2>
<form action="/cgi-bin/guestbook.pl" method="get">
<p>
First Name: <input type="text" name="first" /><br />
Nickname: <input type="text" name="nickname" /><br />
<input type="submit" /> <input type="reset" />
</p>
</form>
```

## The action Attribute

The `action` attribute in the `form` element provides the URL of the program to be used for processing the form. In the example in Figure 15-1, the form information is going to a Perl script called *guestbook.pl*, which resides in the *cgi-bin* directory of the current server (by convention, CGI programs are usually kept in a directory called *cgi-bin*).



```
<h2>Sign the Guestbook:</h2>

<form action="/cgi-bin/guestbook.pl" method="get">
<p>
First Name: <input type="text" name="first" /><br />
Nickname:   <input type="text" name="nickname" /><br />
<input type="submit" /> <input type="reset" />
</p>
</form>
```

*Figure 15-1. A simple form*

## The method Attribute

The `method` attribute specifies one of two methods, either `get` or `post`, for submitting the form information to the server. Form information is typically transferred in a series of variables with their respective content, separated by the ampersand (`&`), as shown here:

```
variable1=content1&variable2=content2&variable3=content3
```

The `name` attributes of form control elements provide the variable names. The content the user enters makes up the content assigned to the variable.

Using the form in Figure 15-1 as an example, if a user entered “Josephine” next to “First Name” and “Josie” next to “Nickname,” the form passes the variables on in this format:

```
name=Josephine&nickname=Josie
```

With the `get` method, the browser transfers the data from the form as part of the URL itself (appended to the end and separated by a question mark) in a single transmission. The information gathered from the nickname example would be transferred via the get method as follows:

```
get http://www.domainname.com/cgi-bin/guestbook.pl?name=
Josephine&nickname=Josie
```

The `post` method transmits the form input information separated from the URL, in essentially a two-part message. The first part of the message is simply the special header sent by the browser with each request. This header contains the URL from the `form` element, combined with a statement that this is a `post` request, plus some other headers we won’t discuss here. This is followed by the actual form data. When the server sees the word “`post`” at the beginning of the message, it stays tuned for the data. The information gathered with the name and nickname form would read as follows using the post method:

```
post http://www.domainname.com/cgi-bin/guestbook.pl HTTP1.0
... [more headers here]
name=Josephine&nickname=Josie
```

Whether you should use post or get may depend on the requirements of your server. In general, if you have a short form with a few short fields, use the get method. Conversely, long, complex forms are best sent via post. If security is an issue (such as when using the input type="password" element), use post, because it offers an opportunity for encryption rather than sending the form data straight away tacked onto the URL. One advantage of get is that the request can be bookmarked, because everything in the request is in the URL. This isn't true with post.

It is possible to send a *query string* via a URL in the document source, as shown here:

```
<a href="http://www.domainname.com/cgi-bin/guestbook.
pl?name=Josephine&amp;nickname=Josie">...</a>
```

Note that in XHTML documents, it is necessary to escape the ampersand character (that is, provide its character entity, &amp;) in the URL. It will be correctly parsed as an ampersand by the processing agent.

### Encoding

Another behind-the-scenes step that happens in the transaction is that the data gets encoded using standard URL encoding. This is a method for translating spaces and other characters not permitted in URLs (such as slashes) into their hexadecimal equivalents. For example, the space character translates to %20, and the slash character is transferred as %2F.

The default encoding format, the Internet Media Type (application/x-www-form-urlencoded), will suffice for most forms. If your form includes a file input type (for uploading documents to the server), you should use the enctype attribute to set the encoding to its alternate setting, multipart/form-data.

## Form Controls

A variety of form control elements (also sometimes called "widgets") are used for gathering information from a form. This section looks at each control and its specific attributes. Every form control (except submit and reset) requires that you give it a name (using the name attribute) so the form-processing application can sort the information. For easier processing of form data on the server, the value of name should not have any character spaces (use underscores or periods instead).

The name attribute works like a variable name. The value provided for name becomes the variable's name. The content entered by the user into the form control is then assigned to the variable. Of all the attributes, the name attribute is key in passing data from the HTML form to any other place in the page, another page, or out through middleware to a database.