LESSON 3

Sending Requests Using HTTP

Various protocols are used for communication over the World Wide Web, perhaps the most important being HTTP, the protocol that is also fundamental to Ajax applications. This lesson introduces the HTTP protocol and shows how it is used to request and receive information.

Introducing HTTP

HTTP or *Hypertext Transfer Protocol* is the main protocol of the World Wide Web. When you request a web page by typing its address into your web browser, that request is sent using HTTP. The browser is an *HTTP client*, and the web page server is (unsurprisingly) an *HTTP server*.

In essence, HTTP defines a set of rules regarding how messages and other data should be formatted and exchanged between servers and browsers.

Why Do I Need To Know About This?

Ajax sends server requests using the HTTP protocol. It's important to recognize the different types of HTTP requests and the responses that the server may return. Ajax applications need to construct HTTP requests to query the server and will base decisions about what to do next on the content of HTTP responses from the server.

What Is (and Isn't) Covered in This Lesson

It would be possible to fill the whole book with information on the HTTP protocol, but here we simply discuss it in terms of its roles in requesting web pages and passing information between them.

In this lesson you'll look at the construction of HTTP requests and responses and see how HTML forms use such requests to transfer data between web pages.

Tip For a detailed account of HTTP, see Sams Publishing's *HTTP Developer's Handbook* by Chris Shiflett.

The HTTP Request and Response

The HTTP protocol can be likened to a conversation based on a series of questions and answers, which we refer to respectively as *HTTP requests* and *HTTP responses*.

The contents of HTTP requests and responses are easy to read and understand, being near to plain English in their syntax.

This section examines the structure of these requests and responses, along with a few examples of the sorts of data they may contain.

The HTTP Request

After opening a connection to the intended server, the HTTP client transmits a request in the following format:

- An opening line
- Optionally, a number of *header lines*
- A blank line
- Optionally, a message body

The opening line is generally split into three parts; the name of the *method*, the path to the required *server resource*, and the *HTTP version* being used. A typical opening line might read:

```
GET /sams/testpage.html HTTP/1.0
```

In this line we are telling the server that we are sending an HTTP request of type `GET` (explained more fully in the next section), we are sending this using HTTP version 1.0, and the server resource we require (including its local path) is

`/sams/testpage.html.`

Note In this example the server resource we seek is on our own server, so we have quoted a relative path. It could of course be on another server elsewhere, in which case the server resource would include the full URL.

Header lines are used to send information about the request, or about the data being sent in the message body. One parameter and value pair is sent per line, the parameter and value being separated by a colon. Here's an example:

```
User-Agent: [name of program sending request]
```

For instance, Internet Explorer v5.5 offers something like the following:

```
User-agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
```

A further example of a common request header is the `Accept:` header, which states what sort(s) of information will be found acceptable as a response from the server:

```
Accept: text/plain, text/html
```

By issuing the header in the preceding example, the request is informing the server that the sending application can accept either plain text or HTML responses (that is, it is not equipped to deal with, say, an audio or video file) .

Note HTTP request methods include `POST`, `GET`, `PUT`, `DELETE`, and `HEAD`. By far the most interesting in our pursuit of Ajax are the `GET` and `POST` requests. The `PUT`, `DELETE`, and `HEAD` requests are not covered here.

The HTTP Response

In answer to such a request, the server typically issues an HTTP response, the first line of which is often referred to as the *status line*. In that line the server echoes the HTTP version and gives a response status code (which is a three-digit integer) and a short message known as a *reason phrase*. Here's an example HTTP response:

```
HTTP/1.0 200 OK
```

The response status code and reason phrase are essentially intended as machine-and human-readable versions of the same message, though the reason phrase may actually vary a little from server to server. Table 3.1 lists some examples of common status codes and reason phrases. The first digit of the status code usually gives some clue about the nature of the message:

- 1**—Information
- 2**—Success
- 3**—Redirected
- 4**—Client error
- 5**—Server error

TABLE 3.1 Some Commonly Encountered HTTP Response Status Codes

Status Code	Explanation
`200 - OK`	The request succeeded.
`204 - No Content`	The document contains no data.
`301 - Moved Permanently`	The resource has permanently moved to a different URI.
`401 - Not Authorized`	The request needs user authentication.
`403 - Forbidden`	The server has refused to fulfill the request.
`404 - Not Found`	The requested resource does not exist on the server.

Status Code	Explanation
`408 - Request Timeout`	The client failed to send a request in the time allowed by the server.
`500 - Server Error`	Due to a malfunctioning script, server configuration error or similar.

Tip A detailed list of status codes is maintained by the World Wide Web Consortium, W3C, and is available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

The response may also contain header lines each containing a header and value pair similar to those of the HTTP request but generally containing information about the server and/or the resource being returned:

```
Server: Apache/1.3.22
Last-Modified: Fri, 24 Dec 1999 13:33:59 GMT
```

HTML Forms

Web pages often contain fields where you can enter information. Examples include select boxes, check boxes, and fields where you can type information. Table 3.2 lists some popular HTML form tags.

TABLE 3.2 Some Common HTML Form Tags

Tag	Description
`<form>...</form>`	Container for the entire form
`<input />`	Data entry element; includes text, password, check box and radio button fields, and submit and reset buttons
`<select>...</select>`	Drop-down select box
`<option>...</option>`	Selectable option within select box
`<textarea>...</textarea>`	Text entry field with multiple rows

After you have completed the form you are usually invited to submit it, using an appropriately labeled button or other page element.

At this point, the HTML form constructs and sends an HTTP request from the user-entered data. The form can use either the `GET` or `POST` request type, as specified in the `method` attribute of the `<form>` tag.

`GET` and `POST` Requests

Occasionally you may hear it said that the difference between `GET` and `POST` requests is that `GET` requests are just for `GET`ting (that is, retrieving) data, whereas `POST` requests can have many uses, such as uploading data, sending mail, and so on.

Although there may be some merit in this rule of thumb, it's instructive to consider the differences between these two HTTP requests in terms of how they are constructed.

A `GET` request encodes the message it sends into a *query string*, which is appended to the URL of the server resource. A `POST` request, on the other hand, sends its message in the *message body* of the request. What actually happens at this point is that the entered data is encoded and sent, via an HTTP request, to the URL declared in the `action` attribute of the form, where the submitted data will be processed in some way.

Whether the HTTP request is of type `GET` or `POST` and the URL to which the form is sent are both determined in the HTML markup of the form. Let's look at the HTML code of a typical form:

```
<form action="http://www.sometargetdomain.com/somepage.htm"
➥     method="post">
Your Surname: <input type="text" size="50" name="surname" />
<br />
<input type="submit" value="Send" />
</form>
```

This snippet of code, when embedded in a web page, produces the simple form shown in Figure 3.1.

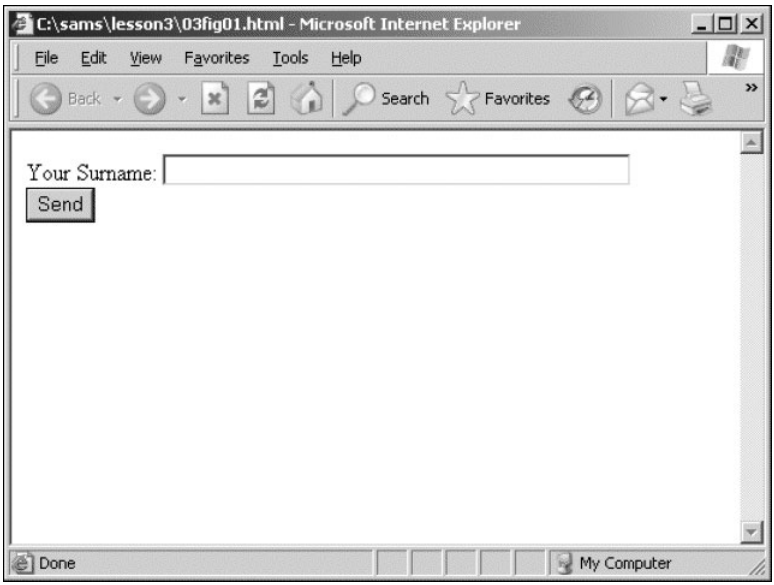

FIGURE 3.1 A simple HTML form.

Let's take a look at the code, line by line. First, we begin the form by using the `<form>` tag, and in this example we give the tag two attributes. The `action` attribute determines the URL to which the submitted form will be sent. This may be to another page on the same server and described by a relative path, or to a remote domain, as in the code behind the form in Figure 3.1.

Next we find the attribute method, which determines whether we want the data to be submitted with a `GET` or a `POST` request.

Now suppose that we completed the form by entering the value ***Ballard*** into the `surname` field. On submitting the form by clicking the Send button, we are taken to http://www.sometargetdomain.com/somepage.htm, where the submitted data will be processed—perhaps adding the surname to a database, for example.

The variable `surname` (the `name` attribute given to the `Your Surname` input field) and its value (the data we entered in that field) will also have been sent to this destination page, encoded into the body of the `POST` request and invisible to users.

Now suppose that the first line of the form code reads as follows:

```
<form action="http://www.sometargetdomain.com/somepage.htm"
➥      method="get">
```

On using the form, we would still be taken to the same destination, and the same variable and its value would also be transmitted. This time, however, the form would construct and send a `GET` request containing the data from the form. Looking at the address bar of the browser, after successfully submitting the form, we would find that it now contains:

```
http://www.example.com/page.htm?surname=Ballard
```

Here we can see how the parameter and its value have been appended to the URL. If the form had contained further input fields, the values entered in those fields would also have been appended to the URL as *`parameter=value`* pairs, with each pair separated by an `&` character. Here's an example in which we assume that the form has a further text input field called `firstname`:

```
http://www.example.com/page.htm?surname=Ballard&firstname=Phil
```

Some characters, such as spaces and various punctuation marks, are not allowed to be transmitted in their original form. The HTML form encodes these characters into a form that can be transmitted correctly. An equivalent process decodes these values at the receiving page before processing them, thus making the encoding/decoding operation essentially invisible to the user. We can, however, see what this encoding looks like by making a `GET` request and examining the URL constructed in doing so.

Suppose that instead of the `surname` field in our form we have a `fullname` field that asks for the full name of the user and encodes that information into a `GET` request. Then, after submitting the form, we might see the following URL in the browser:

```
http://www.example.com/page.htm?fullname=Phil+Ballard
```

Here the space in the name has been replaced by the + character; the decoding process at the receiving end removes this character and replaces the space.

Note In many cases, you may use either the `POST` or `GET` method for your form submissions and achieve essentially identical results. The difference becomes important, however, when you learn how to construct server calls in Ajax applications.

The `XMLHTTPRequest` object at the heart of all Ajax applications uses HTTP to make requests of the server and receive responses. The content of these HTTP requests are essentially identical to those generated when an HTML form is submitted.

Summary

This lesson covered some basics of server requests and responses using the HTTP protocol, the main communications protocol of the World Wide Web. In particular, we discussed how `GET` and `POST` requests are constructed, and how they are used in HTML forms. Additionally, we saw some examples of responses to these requests that we might receive from the server.