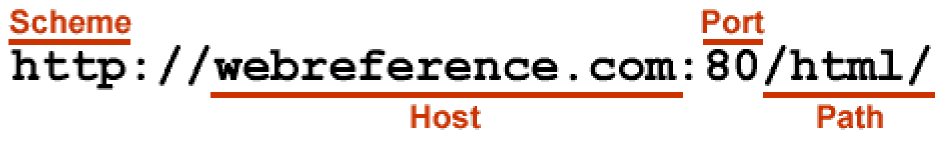
HTTP for HTML Authors, Part I

From: http://www.webreference.com/html/tutorial28/



Astute readers will note that the http bit means that this URL uses the http scheme. This governs how the rest of the URL is understood by a Web browser.

The server bit (webreference.com) and the port bit (80) tell the browser where to go looking for the Web site. The server denotes a computer connected to the Internet; the port denotes a sort of “socket” to which the browser plugs in to speak with the Web server.

The term “Web server” is often used to describe a computer that serves out Web pages, but is also used to describe a computer *program* that runs on a computer and serves out Web pages; this is how I'll use the term for the purposes of this tutorial. The person in charge of the computer runs this program, which then starts listening on a port for the first user to come a-browsing.

Using the, um, anatomically correct example above, a browser that would receive this URL (either because the user typed it into the Location field or because he clicked on a link) would run off to the computer called webreference.com, walk up to port 80 and knock politely. The Web server, which is running inside the computer, would open this port and look at the browser, with a look that suggests a strong “Whaddayawant”? kind of message.

The browser would then politely say something like the following:

```
GET /html/ HTTP/1.1
Host: webreference.com
```

This is HTTP-Speak for “Hello, I'd like to get the document called /html/ on your server. Oh, by the way, I'm fluent in version 1.1 of the HTTP protocol. And in case you're wondering, I came a-knocking for the host webreference.com; I hope this is it.” Most of this stuff is not of much importance to us HTML authors; the important bit is GET /html/. This is the main part of the HTTP *request*.

There are a few types of HTTP requests, but GET is by far the most common; it simply says “Give me this document.” We'll take a look at the second most common type of request, POST, later on.

The Host: webreference.com bit is an example of an HTTP *header* field. Headers are to HTTP requests what meta-information is to HTML; they're not critical, and most of the time they can just be omitted, but they can come in very, very handy. A header is always a name (in this case, Host), followed by a colon (:), followed by the header field's vallue (webreference.com). We'll talk more about headers later on.

Would Sir like some black pepper with his Web page?

After the server has listened attentively to the browser's request, it will usually reply with something like the following:

```
HTTP/1.1 200 OK
Server: Apache/1.3.12 (Unix)
Cache-Control: max-age=60
Content-Type: text/html
Transfer-Encoding: chunked
Date: Tue, 16 Jan 2001 00:39:46 GMT
Expires: Tue, 16 Jan 2001 00:46:07 GMT
d36
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">

( Rest of document omitted )
```

I've cut the response short since it goes on to list all of the HTML for the HTML with Style front page, which is quite lengthy, but you get the point. Translating this into English goes something like this:

HTTP/1.1 means “I too speak HTTP version 1.1”. The important bit, once more, is the bit on the same line with this that says 200 OK. This is an HTTP *response*, and in this case it means “I understood your request and managed to fulfill it.” Users usually don't see responses, but there is one noteable exception: If you've been browsing the Web for some time, you might have come across a page that reads “HTTP 404 Not Found” and then goes on to explain that the document you requested could not be found on the server; 404 is the HTTP response code that means exactly what it says, that the requested document was not found, just like 200 is the HTTP response code that says that everything went fine. There are a whole bunch of different responses defined in HTTP, but 200 is by far the most common and useful one.

As you can see, once again the response contains a whole bunch of headers that are sent back to the browser from the server. Many of these are technical and of little concern to authors; others are very useful. We'll have a look at some of these later on.

Now that you've seen an example of the simplest form of HTTP transaction, you have a good idea of how HTTP works; the browser issues a *request* followed by a few headers, and the server replies with a *response*, also followed by a few headers, and also followed by a document.

Mapping URLs to Files

The most basic set-up of any Web server usually works like this: the path bit in any URL is interpreted as a filename on the computer's hard disk. Usually, the Web server is configured to have a “document root” directory relative to which all URLs are resolved as filenames. Let me give you an example: Let's assume you're running a simple Web server on a Windows-based computer called arnie.acme.com, and the document root is D:\WWWFiles\. When a user types the URL http://arnie.acme.com/jokes/blondes.htm into his browser, the browser asks the server for the document /jokes/blondes.htm. The server will look in the directory D:\WWWFiles\jokes for a file called blondes.htm. If it's there, it will return a 200 OK response followed by the document; if it's not, it will return a 404 Not Found response followed by a helpful error message telling him to look elsewhere for bleached comic relief.

Practically every Web server in existence is set up to behave like this by default, and most hosting services offer only this kind of functionality; you put some files in a directory on the computer running the server (usually using a file transfer utility like FTP), and you can then access them using HTTP.

In this kind of set-up, when the user asks for a path that corresponds to a directory on the hard disk, for instance /jokes/, the server will probably return a list of files in that directory. Since this is rarely a nice way to organize a Web site, most servers are set up to look for a file called the *index* in the directory and display it instead.

Traditionally, index files are named index.html or index.htm. So, if you're running a Web server, and you have a directory called jokes filled with a bunch of un-politically correct documents, and you want people to see an HTML document that links to these documents when they ask for the /jokes/ document, you'd create a file called index.html and put it into this directory. Now, when a browser asks for the document /jokes/, he'll get the contents of the file D:\WWWFiles\jokes\index.html.

The above is only an example; the exact name of an index file, or the behavior of a server when it is asked to return a document that corresponds to a directory, varies from server to server and also according to the server's configuration. If you're running your own server, check the documentation that came with the software; if you're renting Web space from a hosting provider, check the documentation for the service.

However, index files serve to illustrate a basic point: Even though it is often the case, *a URL's path element does not always correspond to an identically named file*. In the example above, you effectively asked for the directory called /jokes/, but you got the file /jokes/index.html instead. This is because it's a lot more flexible to be able to "map" URLs to files in a more complicated manner.

Conclusion

This concludes our introduction to HTTP, and should be enough for those of you that want to build Web sites that consist only of a small number of static (unchanging) pages.

In this tutorial, you learned how a basic HTTP transaction works, and how you can use a simple server to make sure users can use HTTP to access your documents. You learned some basic guidelines about how most server software understands URLs and how it looks for documents in files on your hard disk.

In the following few tutorials, you'll learn why you might want more control over this process; you'll learn how to control the sending of some useful HTTP headers that allow you to specify how often a document should be updated, or which character encoding it uses. You'll also learn how to set things up so your users can interact with your Web server by using HTML forms, and retrieve documents that are not static, but change according to the information supplied in these forms. You'll also learn how to keep track of your users and how to redirect them from one page to another.

Complete HTTP Specification
http://www.w3.org/Protocols/rfc2616/rfc2616.html

W3C's HTTP Page
http://www.w3.org/Protocols/

Content Negotiation

Up until now, you've been used to the fact that a certain path in your server always corresponds to the same file on your server's hard disk. As I've said before, this is often not the case; the first example of such a situation which we'll examine is server-driven content negotiation.

The term *content negotiation* means that there exists a system which can be used to select one of several alternative documents for a given request. In this case, server-driven content negotiation is such a system.

The purpose of content negotiation is to automatically select the correct document for a user based on his abilities, preferences and set-up. The most commonly used example of content negotiation is in selecting documents in different languages. As we saw before, a Web server can specify the language of a document via the `Content-Language` HTTP header. This is useful if the reader doesn't speak the language, but in the end is very similar in functionality to that dumb look on your face when confronted with a language you've never seen before.

Another useful HTTP header is the `Accept-Language` header. In contrast to `Content-Language`, this header is sent along by the browser along with a request (instead of being sent by the server along with a response) and indicates which languages the browser would rather receive. Most modern browsers (including the Big Two) allow users to specify their preferred languages, usually in order of preference. These are then sent to Web servers via the `Accept-Language` header. A typical `Accept-Language` header might look like this:

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

This header specifies that Danish is the preferred language, followed by British English, followed by other English variants. The `q=0.8` bits are what are called *quality ratings* and merely specify the relative preference the user is expressing for each language; the details of how they work are beyond the scope of this tutorial.

A Web server, when confronted with such a header, is expected to look at the various alternatives it has on offer and select the one that can best satisfy the user's lack of linguistic skills. The precise mechanism through which this is accomplished varies from server to server, and once again is a matter of implementation. However, if you're going to offer a Web site in multiple languages, this is an excellent way to make sure the right people get the right version of your documents.

Content negotiation is not only done on the basis of language; the HTTP request headers `Accept` and `Accept-Charset` work in similar ways as `Accept-Language`, and offer means of selecting a version of the correct media type and character set (or encoding, in the case of HTML).

Software version info

Another two very useful HTTP headers that are used very often are `Server` and `User-Agent`. These two are used in the reponse and request, to specify the product name and version of the Web server and the user agent respectively. Neither of these fields is mandatory, and in fact they are often omitted, but they do sometimes give useful information.

As you might imagine, having experienced the chaos that is the Web today, a lot of people have thought of using the `User-Agent` header to dispense different documents depending on the browser in order to satisfy the numerous bugs and incompatibilities in popular browsers. Of course, before you know it, browser makers detested this attempt at setting right their wrongs and started imitating each other's `User-Agent` headers in order to convince Web servers that they where, in fact a different browser, so as to show that they are compatible with the latest and greatest. Unfortunately the browser makers forgot to make their browsers compatible with anything, and hence this nice and reliable way of working around browser bugs went the way of the Dodo. Although most modern browsers have stopped this despicable masquerading, looking at the `User-Agent` field is still a poor way of judging what a user agent is capable of.

The `Server` field is of less direct practical importance, and is mostly used by people like NetCraft to gather statistical information about software usage on the Web. Again, this is an optional header and many Web servers refuse to let the world know what version of which software they're running.

Referral Information

The last notable HTTP header we're going to examine today is the `Referer` field. Let me take a second to ensure those of you with a good knowledge of English that yes, that is *not* the right spelling for the word “referrer,” but people have been using the misspelled form for ages and whoever was responsible for the initial blunder has long been lost in oblivion. If this annoys you just think of all the British people out there who have to put up with the `color` property. If you fall into both categories, you can at least feel smug and superior about it.

But more to the point, what is the `Referer` header for, you ask. Well, the `Referer` header indicates which page *referred* the user (surprise) to this one. So, when a user is viewing a document at the URL `http://www.acme.com/foo.html` and then clicks on a link in this document and ends up in `http://www.slezcorp.com/bar.html`, the request for the second page may contain a header that looks like this:

```
Referer: http://www.acme.com/foo.html
```

This is the HTTP equivalent of saying “Vinnie sent me.” The Web server's reaction to this will depend entirely on the Web server and its relationship to Vinnie. It might then welcome you in and not only deliver the document you asked for, but also make you a nice cup of tea and let you in on a few secrets. It might slam the door in your face and tell you to get lost. Or, as is the case most of the time, it will just shrug and go about its business, not even bothering to ask who on Earth this Vinnie guy is.

There are very few sites that actually use the `Referer` header for anything useful, mainly because there's no way to prove that a user is coming from the stated URL, and you have to take him on his word, but also because this field is often optional. In general, you can't always depend on a browser setting the `Referer` header to what you'd expect it to.

What `Referer` *can* be useful for is tracking where your readers are coming from, by keeping logs of the value of the header. Most Web servers allow you to do this very easily. If you set this up, whenever someone links to your site and sends plenty of innocent viewers to an eternity of suffering, or at the very least a few seconds of wasted bandwidth, you'll know who to hold responsible, either to thank him for bringing your “Save the Completlius Uslessius Fungus of North Wahalamabad” campaign to the attention of so many motivated volunteers or destroy him for crashing your network just when you where winning the *Unreal Tournament* office final.

HTTP Made Really Easy

http://www.jmarshall.com/easy/http/

HTTP is the network protocol of the Web. It is both simple and powerful. Knowing HTTP enables you to write Web browsers, Web servers, automatic page downloaders, link-checkers, and other useful tools.

This tutorial explains the simple, English-based structure of HTTP communication, and teaches you the practical details of writing HTTP clients and servers. It assumes you know basic socket programming. HTTP is simple enough for a beginning sockets programmer, so this page might be a good followup to a sockets tutorial. This Sockets FAQ (hint: see "Categorized Questions" section at bottom) focuses on C, but the underlying concepts are language-independent.

Since you're reading this, you probably already use CGI. If not, it makes sense to learn that first.

The whole tutorial is about 15 printed pages long, including examples. The first half explains basic HTTP 1.0, and the second half explains the new requirements and features of HTTP 1.1. This tutorial doesn't cover everything about HTTP; it explains the basic framework, how to comply with the requirements, and where to find out more when you need it. If you plan to use HTTP extensively, you should read the specification as well-- see the end of this document for more details.

Before getting started, understand the following two paragraphs:

`<LECTURE>`

> ***Writing HTTP or other network programs requires more care than programming for a single machine.*** Of course, you have to follow standards, or no one will understand you. But even more important is the burden you place on other machines. Write a bad program for your own machine, and you waste your own resources (CPU time, bandwidth, memory). Write a bad network program, and you waste other people's resources. Write a *really* bad network program, and you waste many thousands of people's resources at the same time. Sloppy and malicious network programming forces network standards to be modified, made safer but less efficient. So be careful, respectful, and cooperative, for everyone's sake.
>
> ***In particular, don't be tempted to write programs that automatically follow Web links*** (called *robots* or *spiders*) before you really know what you're doing. They can be useful, but a badly-written robot is one of the worst kinds of programs on the Web, blindly following a rapidly increasing number of links and quickly draining server resources. If you plan to write anything like a robot, please read more about them. There may already be a working program to do what you want. If you really need to write your own, read these guidelines. Definitely support the current Standard for Robot Exclusion, and stay tuned for further developments.

`</LECTURE>`

OK, enough of that. Let's get started.

Table of Contents

Using HTTP 1.0

What is HTTP?

HTTP stands for **Hypertext Transfer Protocol**. It's the network protocol used to deliver virtually all files and other data (collectively called *resources*) on the World Wide Web, whether they're HTML files, image files, query results, or anything else. Usually, HTTP takes place through TCP/IP sockets (and this tutorial ignores other possibilities).

A browser is an *HTTP client* because it sends requests to an *HTTP server* (Web server), which then sends responses back to the client. The standard (and default) port for HTTP servers to listen on is 80, though they can use any port.

What are "Resources"?

HTTP is used to transmit *resources*, not just files. A resource is some chunk of information that can be identified by a URL (it's the **R** in **URL**). The most common kind of resource is a file, but a resource may also be a dynamically-generated query result, the output of a CGI script, a document that is available in several languages, or something else.

While learning HTTP, it may help to think of a resource as similar to a file, but more general. As a practical matter, almost all HTTP resources are currently either files or server-side script output.

Structure of HTTP Transactions

Like most network protocols, HTTP uses the client-server model: An *HTTP client* opens a connection and sends a *request message* to an *HTTP server*; the server then returns a *response message*, usually containing the resource that was requested. After delivering the response, the server closes the connection (making HTTP a *stateless* protocol, i.e. not maintaining any connection information between transactions).

The format of the request and response messages is similar, and English-oriented. Both kinds of messages consist of:

an initial line,

zero or more header lines,

a blank line (i.e. a CRLF by itself), and

an optional message body (e.g. a file, or query data, or query output).

Put another way, the format of an HTTP message is:

```
<initial line, different for request vs. response>
Header1: value1
Header2: value2
Header3: value3

<optional message body goes here, like file contents or query data;
 it can be many lines long, or even binary data $&*%@!^$@>
```

Initial lines and headers should end in CRLF, though you should gracefully handle lines ending in just LF. (More exactly, CR and LF here mean ASCII values 13 and 10, even though some platforms may use different characters.)

Initial Request Line

The initial line is different for the request than for the response. A request line has three parts, separated by spaces: a *method* name, the local path of the requested resource, and the version of HTTP being used. A typical request line is:

```
GET /path/to/file/index.html HTTP/1.0
```

Notes:

GET is the most common HTTP method; it says "give me this resource". Other methods include **POST** and **HEAD**-- more on those later. Method names are always uppercase.

The path is the part of the URL after the host name, also called the *request URI* (a URI is like a URL, but more general).

The HTTP version always takes the form "**HTTP/x.x**", uppercase.

Initial Response Line (Status Line)

The initial response line, called the *status line*, also has three parts separated by spaces: the HTTP version, a *response status code* that gives the result of the request, and an English *reason phrase* describing the status code. Typical status lines are:

```
HTTP/1.0 200 OK
```

or

```
HTTP/1.0 404 Not Found
```

Notes:

The HTTP version is in the same format as in the request line, "**HTTP/x.x**".

The status code is meant to be computer-readable; the reason phrase is meant to be human-readable, and may vary.

The status code is a three-digit integer, and the first digit identifies the general category of response:

1xx indicates an informational message only

2xx indicates success of some kind

3xx redirects the client to another URL

4xx indicates an error on the client's part

5xx indicates an error on the server's part

The most common status codes are:

`200 OK`

The request succeeded, and the resulting resource (e.g. file or script output) is returned in the message body.

`404 Not Found`

The requested resource doesn't exist.

`301 Moved Permanently`
`302 Moved Temporarily`
`303 See Other` *(HTTP 1.1 only)*

The resource has moved to another URL (given by the **`Location:`** response header), and should be automatically retrieved by the client. This is often used by a CGI script to redirect the browser to an existing file.

`500 Server Error`

An unexpected server error. The most common cause is a server-side script that has bad syntax, fails, or otherwise can't run correctly.

A complete list of status codes is in the HTTP specification (section 9 for HTTP 1.0, and section 10 for HTTP 1.1).

Header Lines

Header lines provide information about the request or response, or about the object sent in the message body.

The header lines are in the usual text header format, which is: one line per header, of the form "**`Header-Name: value`**", ending with CRLF. It's the same format used for email and news postings, defined in RFC 822, section 3. Details about RFC 822 header lines:

As noted above, they should end in CRLF, but you should handle LF correctly.

The header name is not case-sensitive (though the value may be).

Any number of spaces or tabs may be between the ":" and the value.

Header lines beginning with space or tab are actually part of the previous header line, folded into multiple lines for easy reading.

Thus, the following two headers are equivalent:

```
Header1: some-long-value-1a, some-long-value-1b
HEADER1:    some-long-value-1a,
            some-long-value-1b
```

HTTP 1.0 defines 16 headers, though none are required. HTTP 1.1 defines 46 headers, and one (**`Host:`**) is required in requests. For Net-politeness, consider including these headers in your requests:

The **`From:`** header gives the email address of whoever's making the request, or running the program doing so. (This *must* be user-configurable, for privacy concerns.)

The **`User-Agent:`** header identifies the program that's making the request, in the form "**Program-name/x.xx**", where **x.xx** is the (mostly) alphanumeric version of the program. For example, Netscape 3.0 sends the header "**`User-agent: Mozilla/3.0Gold`**".

These headers help webmasters troubleshoot problems. They also reveal information about the user. When you decide which headers to include, you must balance the webmasters' logging needs against your users' needs for privacy.

If you're writing servers, consider including these headers in your responses:

The **`Server:`** header is analogous to the **`User-Agent:`** header: it identifies the server software in the form "**Program-name/x.xx**". For example, one beta version of Apache's server returns "**`Server: Apache/1.2b3-dev`**".

The **`Last-Modified:`** header gives the modification date of the resource that's being returned. It's used in caching and other bandwidth-saving activities. Use Greenwich Mean Time, in the format

```
Last-Modified: Fri, 31 Dec 1999 23:59:59 GMT
```

The Message Body

An HTTP message may have a body of data sent after the header lines. In a response, this is where the requested resource is returned to the client (the most common use of the message body), or perhaps explanatory text if there's an error. In a request, this is where user-entered data or uploaded files are sent to the server.

If an HTTP message includes a body, there are usually header lines in the message that describe the body. In particular,

The **`Content-Type:`** header gives the MIME-type of the data in the body, such as **`text/html`** or **`image/gif`**.

The **`Content-Length:`** header gives the number of bytes in the body.

Sample HTTP Exchange

To retrieve the file at the URL

```
http://www.somehost.com/path/file.html
```

first open a socket to the host **www.somehost.com**, port 80 (use the default port of 80 because none is specified in the URL). Then, send something like the following through the socket:

```
GET /path/file.html HTTP/1.0
From: someuser@jmarshall.com
User-Agent: HTTPTool/1.0
[blank line here]
```

The server should respond with something like the following, sent back through the same socket:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
  .
  .
  .
</body>
</html>
```

After sending the response, the server closes the socket.

Other HTTP Methods, Like HEAD and POST

Besides GET, the two most commonly used methods are HEAD and POST.

The HEAD Method

A HEAD request is just like a GET request, except it asks the server to return the response headers only, and not the actual resource (i.e. no message body). This is useful to check characteristics of a resource without actually downloading it, thus saving bandwidth. Use HEAD when you don't actually need a file's contents.

The response to a HEAD request must *never* contain a message body, just the status line and headers.

The POST Method

A POST request is used to send data to the server to be processed in some way, like by a CGI script. A POST request is different from a GET request in the following ways:

There's a block of data sent with the request, in the message body. There are usually extra headers to describe this message body, like **`Content-Type:`** and **`Content-Length:`**.

The *request URI* is not a resource to retrieve; it's usually a program to handle the data you're sending.

The HTTP response is normally program output, not a static file.

The most common use of POST, by far, is to submit HTML form data to CGI scripts. In this case, the **`Content-Type:`** header is usually **`application/x-www-form-urlencoded`**, and the **`Content-Length:`** header gives the length of the URL-encoded form data (here's a note on URL-encoding). The CGI script receives the message body through STDIN, and decodes it. Here's a typical form submission, using POST:

```
POST /path/script.cgi HTTP/1.0
From: frog@jmarshall.com
User-Agent: HTTPTool/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

home=Cosby&favorite+flavor=flies
```

You can use a POST request to send whatever data you want, not just form submissions. Just make sure the sender and the receiving program agree on the format.

The GET method can also be used to submit forms. The form data is URL-encoded and appended to the request URI. Here are more details.

If you're writing HTTP servers that support CGI scripts, you should read the NCSA's CGI definition if you haven't already, especially which environment variables you need to pass to the scripts.

HTTP Proxies

An *HTTP proxy* is a program that acts as an intermediary between a client and a server. It receives requests from clients, and forwards those requests to the intended servers. The responses pass back through it in the same way. Thus, a proxy has functions of both a client and a server.

Proxies are commonly used in firewalls, for LAN-wide caches, or in other situations. If you're writing proxies, read the HTTP specification; it contains details about proxies not covered in this tutorial.

When a client uses a proxy, it typically sends all requests to that proxy, instead of to the servers in the URLs. Requests to a proxy differ from normal requests in one way: in the first line, they use the complete URL of the resource being requested, instead of just the path. For example,

```
GET http://www.somehost.com/path/file.html HTTP/1.0
```

That way, the proxy knows which server to forward the request to (though the proxy itself may use another proxy).

Being Tolerant of Others

As the saying goes (in network programming, anyway), "Be strict in what you send and tolerant in what you receive." Other clients and servers you interact with may have minor flaws in their messages, but you should try to work gracefully with them. In particular, the HTTP specification suggests the following:

Even though header lines should end with CRLF, someone might use a single LF instead. Accept either CRLF or LF.

The three fields in the initial message line should be separated by a single space, but might instead use several spaces, or tabs. Accept any number of spaces or tabs between these fields.

The specification has other suggestions too, like how to handle varying date formats. If your program interprets dates from other programs, read the "Tolerant Applications" section of the specification.

Manually Experimenting with HTTP

Using telnet, you can open an interactive socket to an HTTP server. This lets you manually enter a request, and see the response written to your screen. It's a great help when learning HTTP, to see exactly how a server responds to a particular request. It also helps when troubleshooting.

From a Unix prompt, open a connection to an HTTP server with something like

```
telnet www.somehost.com 80
```

Then enter your request line by line, like

```
GET /path/file.html HTTP/1.0
[headers here, if any]
[blank line here]
```

After you finish your request with

Conclusion

That's the basic structure of HTTP. If you understand everything so far, you have a good overview of HTTP communication, and should be able to write simple HTTP 1.0 programs. See this example to get started. Again, before you do anything heavy-duty, read the specification.

The rest of this document tells how to upgrade your clients and servers to use HTTP 1.1. There is a list of new client requirements, and a list of new server requirements. You can stop here if HTTP 1.0 satisfies your current needs (though you'll probably need HTTP 1.1 in the future).

Note: As of early 1997, the Web is moving from HTTP 1.0 to HTTP 1.1. Whenever practical, use HTTP 1.1. It's more efficient overall, and by using it, you'll help the Web perform better for everyone.