HTTP Made Really Easy

A Practical Guide to Writing Clients and Servers

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 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

Top of Page

Appendix

Several related topics are discussed on a "footnotes" page:

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.

Return to Table of Contents

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 are 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.)

Return to Table of Contents

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.

Return to Table of Contents

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).

Return to Table of Contents

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
```

Return to Table of Contents

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.

Return to Table of Contents

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.

To familiarize yourself with requests and responses, manually experiment with HTTP using telnet.

Return to Table of Contents

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.

Return to Table of Contents

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.

Return to Table of Contents

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).

Return to Table of Contents

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.

Return to Table of Contents

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.

HTTP 1.1

Like many protocols, HTTP is constantly evolving. HTTP 1.1 has recently been defined, to address new needs and overcome shortcomings of HTTP 1.0. Generally speaking, it is a superset of HTTP 1.0. Improvements include:

- Faster response, by allowing multiple transactions to take place over a single *persistent connection*.
- Faster response and great bandwidth savings, by adding cache support.
- Faster response for dynamically-generated pages, by supporting *chunked encoding*, which allows a response to be sent before its total length is known.
- Efficient use of IP addresses, by allowing multiple domains to be served from a single IP address.

HTTP 1.1 requires a few extra things from both clients and servers. The next two sections detail how to make clients and servers comply with HTTP 1.1. If you're only writing clients, you can skip the section on servers. If you're writing servers, read both sections.

Only *requirements* for HTTP 1.1 compliance are described here. HTTP 1.1 has many optional features you may find useful; read the specification to learn more.

Return to Table of Contents

HTTP 1.1 Clients

To comply with HTTP 1.1, clients must

- include the **`Host:`** header with each request
- accept responses with *chunked* data
- either support *persistent connections*, or include the "**`Connection: close`**" header with each request
- handle the "**`100 Continue`**" response

Return to Table of Contents

Host: Header

Starting with HTTP 1.1, one server at one IP address can be *multi-homed*, i.e. the home of several Web domains. For example, "www.host1.com" and "www.host2.com" can live on the same server.

Several domains living on the same server is like several people sharing one phone: a caller knows who they're calling for, but whoever answers the phone doesn't. Thus, every HTTP request must specify which host name (and possibly port) the request is intended for, with the **`Host:`** header. A complete HTTP 1.1 request might be

```
GET /path/file.html HTTP/1.1
Host: www.host1.com:80
[blank line here]
```

except the "`:80`" isn't required, since that's the default HTTP port.

`Host:` is the only required header in an HTTP 1.1 request. *It's also the most urgently needed new feature in HTTP 1.1.* Without it, each host name requires a unique IP address, and we're quickly running out of IP addresses with the explosion of new domains.

Return to Table of Contents

Chunked Transfer-Encoding

If a server wants to start sending a response before knowing its total length (like with long script output), it might use the simple *chunked transfer-encoding*, which breaks the complete response into smaller chunks and sends them in series. You can identify such a response because it contains the "**`Transfer-Encoding: chunked`**" header. All HTTP 1.1 clients must be able to receive chunked messages.

A chunked message body contains a series of *chunks*, followed by a line with "0" (zero), followed by optional footers (just like headers), and a blank line. Each chunk consists of two parts:

- a line with the size of the chunk data, in hex, possibly followed by a semicolon and extra parameters you can ignore (none are currently standard), and ending with CRLF.
- the data itself, followed by CRLF.

So a chunked response might look like the following:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Transfer-Encoding: chunked

1a; ignore-stuff-here
abcdefghijklmnopqrstuvwxyz
10
1234567890abcdef
0
some-footer: some-value
another-footer: another-value
[blank line here]
```

Note the blank line after the last footer. The length of the text data is 42 bytes (1a + 10, in hex), and the data itself is **abcdefghijklmnopqrstuvwxyz1234567890abcdef**. The footers should be treated like headers, as if they were at the top of the response.

The chunks can contain any binary data, and may be much larger than the examples here. The size-line parameters are rarely used, but you should at least ignore them correctly. Footers are also rare, but might be appropriate for things like checksums or digital signatures.

For comparison, here's the equivalent to the above response, without using chunked encoding:

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 42
some-footer: some-value
another-footer: another-value

abcdefghijklmnopqrstuvwxyz1234567890abcdef
```

Return to Table of Contents

Persistent Connections and the "Connection: close" Header

In HTTP 1.0 and before, TCP connections are closed after each request and response, so each resource to be retrieved requires its own connection. Opening and closing TCP connections takes a substantial amount of CPU time, bandwidth, and memory. In practice, most Web pages consist of several files on the same server, so much can be saved by allowing several requests and responses to be sent through a single *persistent connection*.

Persistent connections are the default in HTTP 1.1, so nothing special is required to use them. Just open a connection and send several requests in series (called *pipelining*), and read the responses in the same order as the requests were sent. If you do this, be very careful to read the correct length of each response, to separate them correctly.

If a client includes the "**`Connection: close`**" header in the request, then the connection will be closed after the corresponding response. **Use this if you don't support persistent connections**, or if you know a request will be the last on its connection. Similarly, if a response contains this header, then the server will close the connection following that response, and the client shouldn't send any more requests through that connection.

A server might close the connection before all responses are sent, so a client must keep track of requests and resend them as needed. When resending, don't pipeline the requests until you know the connection is persistent. Don't pipeline at all if you know the server won't support persistent connections (like if it uses HTTP 1.0, based on a previous response).

Return to Table of Contents

The "100 Continue" Response

During the course of an HTTP 1.1 client sending a request to a server, the server might respond with an interim "**`100 Continue`**" response. This means the server has received the first part of the request, and can be used to aid communication over slow links. In any case, all HTTP 1.1 clients must handle the 100 response correctly (perhaps by just ignoring it).

The "**`100 Continue`**" response is structured like any HTTP response, i.e. consists of a status line, optional headers, and a blank line. Unlike other responses, it is always followed by another complete, final response.

So, further extending the last example, the full data that comes back from the server might consist of two responses in series, like

```
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 42
some-footer: some-value
another-footer: another-value

abcdefghijklmnoprstuvwxyz1234567890abcdef
```

To handle this, a simple HTTP 1.1 client might read one response from the socket; if the status code is 100, discard the first response and read the next one instead.

Return to Table of Contents

HTTP 1.1 Servers

To comply with HTTP 1.1, servers must:

- require the **`Host:`** header from HTTP 1.1 clients
- accept absolute URL's in a request

- accept requests with *chunked* data
- either support *persistent connections*, or include the "**`Connection: close`**" header with each response
- use the "**`100 Continue`**" response appropriately
- include the **`Date:`** header in each response
- handle requests with **`If-Modified-Since:`** or **`If-Unmodified-Since:`** headers
- support at least the GET and HEAD methods
- support HTTP 1.0 requests

Return to Table of Contents

Requiring the Host: Header

Because of the urgency of implementing the new **`Host:`** header, servers are not allowed to tolerate HTTP 1.1 requests without it. If a server receives such a request, it must return a "**`400 Bad Request`**" response, like

```
HTTP/1.1 400 Bad Request
Content-Type: text/html
Content-Length: 111

<html><body>
<h2>No Host: header received</h2>
HTTP 1.1 requests must include the Host: header.
</body></html>
```

This requirement applies *only* to clients using HTTP 1.1, not any future version of HTTP. If the request uses an HTTP version later than 1.1, the server can accept an absolute URL instead of a **`Host:`** header (see next section). If the request uses HTTP 1.0, the server may accept the request without any host identification.

Return to Table of Contents

Accepting Absolute URL's

The **`Host:`** header is actually an interim solution to the problem of host identification. In future versions of HTTP, requests will use an absolute URL instead of a pathname, like

```
GET http://www.somehost.com/path/file.html HTTP/1.2
```

To enable this protocol transition, HTTP 1.1 servers must accept this form of request, even though HTTP 1.1 clients won't send them. The server must still report an error if an HTTP 1.1 client leaves out the **`Host:`** header, as described in the previous section.

Return to Table of Contents

Chunked Transfer-Encoding

Just as HTTP 1.1 clients must accept chunked responses, servers must accept chunked requests (an unlikely scenario, but possible). See the earlier section on HTTP 1.1 Clients for details of the chunked data format.

Servers aren't required to generate chunked messages; they just have to be able to receive them.

Return to Table of Contents

Persistent Connections and the "Connection: close" Header

If an HTTP 1.1 client sends multiple requests through a single connection, the server should send responses back in the same order as the requests-- this is all it takes for a server to support persistent connections.

If a request includes the "**`Connection: close`**" header, that request is the final one for the connection and the server should close the connection after sending the response. Also, the server should close an idle connection after some timeout period (can be anything; 10 seconds is fine).

If you don't want to support persistent connections, include the "**`Connection: close`**" header in the response. Use this header whenever you want to close the connection, even if not all requests have been fulfilled. The header says that the connection will be closed after the current response, and a valid HTTP 1.1 client will handle it correctly.

Return to Table of Contents

Using the "100 Continue" Response

As described in the section on HTTP 1.1 Clients, this response exists to help deal with slow links.

When an HTTP 1.1 server receives the first line of an HTTP 1.1 (or later) request, it must respond with either "**`100 Continue`**" or an error. If it sends the "**`100 Continue`**" response, it must also send another, final response, once the request has been processed. The "**`100 Continue`**" response requires no headers, but must be followed by the usual blank line, like:

```
    HTTP/1.1 100 Continue
    [blank line here]
    [another HTTP response will go here]
```

Don't send "**`100 Continue`**" to HTTP 1.0 clients, since they don't know how to handle it.

Return to Table of Contents

The Date: Header

Caching is an important improvement in HTTP 1.1, and can't work without timestamped responses. So, servers must timestamp every response with a **`Date:`** header containing the current time, in the form

```
    Date: Fri, 31 Dec 1999 23:59:59 GMT
```

All responses except those with 100-level status (but including error responses) must include the **`Date:`** header.

All time values in HTTP use Greenwich Mean Time.

Return to Table of Contents

Handling Requests with If-Modified-Since: or If-Unmodified-Since: Headers

To avoid sending resources that don't need to be sent, thus saving bandwidth, HTTP 1.1 defines the **`If-Modified-Since:`** and **`If-Unmodified-Since:`** request headers. The former says "only send the resource if it has changed since this date"; the latter says the opposite. Clients aren't required to use them, but HTTP 1.1 servers are required to honor requests that do use them.

Unfortunately, due to earlier HTTP versions, the date value may be in any of three possible formats:

```
    If-Modified-Since:  Fri, 31 Dec 1999 23:59:59 GMT
    If-Modified-Since:  Friday, 31-Dec-99 23:59:59 GMT
    If-Modified-Since:  Fri Dec 31 23:59:59 1999
```

Again, all time values in HTTP use Greenwich Mean Time (though try to be tolerant of non-GMT times). If a date with a two-digit year seems to be more than 50 years in the future, treat it as being in the past--this helps with the millennium bug. In fact, do this with any date handling in HTTP 1.1.

Although servers must accept all three date formats, HTTP 1.1 clients and servers must only generate the first kind.

If the date in either of these headers is invalid, or is in the future, ignore the header.

If, without the header, the request would result in an unsuccessful (non-200-level) status code, ignore the header and send the non-200-level response. In other words, only apply these headers when you know the resource would otherwise be sent.

The **`If-Modified-Since:`** header is used with a GET request. If the requested resource has been modified since the given date, ignore the header and return the resource as you normally would. Otherwise, return a "**`304 Not Modified`**" response, including the **`Date:`** header and no message body, like

```
    HTTP/1.1 304 Not Modified
    Date: Fri, 31 Dec 1999 23:59:59 GMT
    [blank line here]
```

The **`If-Unmodified-Since:`** header is similar, but can be used with any method. If the requested resource has *not* been modified since the given date, ignore the header and return the resource as you normally would. Otherwise, return a "**`412 Precondition Failed`**" response, like

```
    HTTP/1.1 412 Precondition Failed
    [blank line here]
```

Return to Table of Contents

Supporting the GET and HEAD methods

To comply with HTTP 1.1, a server must support at least the GET and HEAD methods. If you're handling CGI scripts, you should probably support the POST method too.

Four other methods (PUT, DELETE, OPTIONS, and TRACE) are defined in HTTP 1.1, but are rarely used. If a client requests a method you don't support, respond with "**`501 Not Implemented`**", like

```
    HTTP/1.1 501 Not Implemented
    [blank line here]
```

Return to Table of Contents

Supporting HTTP 1.0 Requests

To be compatible with older browsers, HTTP 1.1 servers must support HTTP 1.0 requests. In particular, when a request uses HTTP 1.0 (as identified in the initial request line),

- don't require the **`Host:`** header, and
- don't send the "**`100 Continue`**" response.

Return to Table of Contents

The HTTP Specification

If you plan to do anything elaborate in HTTP, read the official specification. HTTP 1.0 was never made an official Internet standard, but the *de facto* standard is described in RFC 1945. HTTP 1.1 was developed by a working group of the IETF, openly gathering input from many sources before reaching an approximate consensus. In January 1997, RFC 2068 was created from draft 07 of the HTTP 1.1 spec; there will be more revisions before it becomes an Internet standard.

These documents are available in various formats:

- **HTTP 1.0 (RFC 1945)--** HTML, text, and gzip'd PostScript
- **HTTP 1.1**
    - **Latest (rev 06, 18-Nov-1998)--** text (gzip'd), Postscript (gzip'd), and MS Word (gzip'd)
    - **RFC 2068 (draft 07)--** currently only text.

Download and print the version you'll be using, for reference and bedtime reading.

The World Wide Web Consortium has a page devoted to HTTP including news and updates, and a page listing HTTP specifications, drafts, and reports.

Other RFC's you might find useful:

- RFC 822-- structure of Internet text messages, including header fields
- RFC 2396-- definition of URL/URI syntax (replaces RFC 1738 and RFC 1808)
- RFC 1521-- definition of MIME and of MIME types

Return to Table of Contents

Last (significantly) modified: August 15, 1997

Footnotes for "HTTP Made Really Easy"

Back to Main Page   Go to Other Tutorials

Sample HTTP Client

To see a working example of an HTTP client, download and view the source code for **geturl** (in Perl, with comments).

geturl downloads a resource at a given URL using GET, and saves it locally if desired. The first version was written to use HTTP 1.0. It was then modified to use HTTP 1.1, taking the steps detailed on the main page in the section HTTP 1.1 Clients.

- **HTTP 1.0**-- Download geturl10.pl, or just view it.
- **HTTP 1.1**-- Download geturl11.pl or just view it.

geturl runs from the command line. To download **`http://www.somehost.com/somefile`** into the local file **`myfile`**, use

```
    geturl http://www.somehost.com/somefile myfile
```

Return to Table of Contents

Using GET to Submit Query or Form Data

You can use GET to send small amounts of data to the server. The key is to understand just what the *request URI* is: It's not necessarily a file name, it's a string that *identifies a data resource* on the server. That may be a file name, but it may also be, for example, a specific query to a specific database. The result of that query may not live in a file, but it's a data resource all the same, identified by the search engine and the query data that together produce it.

So, to send data to a CGI script using a GET request, include that data after the question mark in the URL (read about URL syntax for more details). For example,

```
    GET /path/script.cgi?field1=value1&field2=value2 HTTP/1.0
```

This is an example of URL-encoded data, which is how HTML forms are submitted using the GET request. This is only appropriate if the amount of data is small, and there are no side effects on the server. Otherwise, use POST for form submission.

Return to Table of Contents

URL-encoding

HTML form data is usually URL-encoded to package it in a GET or POST submission. In a nutshell, here's how you URL-encode the name-value pairs of the form data:

1. Convert all "unsafe" characters in the names and values to "**%xx**", where "**xx**" is the ascii value of the character, in hex. "Unsafe" characters include **=**, **&**, **%**, **+**, non-printable characters, and any others you want to encode-- there's no danger in encoding too many characters. For simplicity, you might encode all non-alphanumeric characters.
2. Change all spaces to plusses.
3. String the names and values together with **=** and **&**, like
   ```
   name1=value1&name2=value2&name3=value3
   ```
4. This string is your message body for POST submissions, or the query string for GET submissions.

For example, if a form has a field called "name" that's set to "Lucy", and a field called "neighbors" that's set to "Fred & Ethel", the URL-encoded form data would be

```
    name=Lucy&neighbors=Fred+%26+Ethel
```

with a length of 34.

Technically, the term "URL-encoding" refers only to step 1 above; it's defined in RFC 2396, section 2.4 (previously in RFC 1738, section 2.2). Commonly, the term refers to this entire process of packaging form data into one long string.

Return to Table of Contents

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 the blank line, you'll see the raw response from the server, including the status line, headers, and message body.

Return to Table of Contents

Back to Main Page

Last modified: March 18, 1997