

The World of HTTP

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.)

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).

[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`

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](#)

HTTP Exchange

Here a sample HTTP 1.0 exchange:

```
GET / HTTP/1.0    >
                  >
                  < HTTP/1.0 200 OK
                  < Date: Wed, 18 Sep 1996 20:18:59 GMT
                  < Server: Apache/1.0.0
                  < Content-type: text/html
                  < Content-length: 1579
                  < Last-modified: Mon, 22 Jul 1996 22:23:34 GMT
                  <
                  < HTML document
```

The use of full headers is preferred for several reasons:

- The first line of a server header includes a response code indicating the success or failure of the operation.
- One of the server header fields will be `Content-type:`, which specifies a MIME type to describe how the document should be interpreted.
- If the document has moved, the server can specify its new location with a `Location:` field, allowing the client to transparently retry the request using the new URL.
- The `Authorization:` and `WWW-Authenticate:` fields allow access controls to be placed on Web documents.
- The `Referer:` field allows the client to tell the server the URL of the document that triggered this request, permitting savvy servers to trace clients through a series of requests.

In addition to `GET` requests, clients can also send `HEAD` and `POST` requests, of which `POSTS` are the most important. `POSTS` are used for HTML forms and other operations that require the client to transmit a block of data to the server. After sending the header and the blank line, the client transmits the data. The header must have included a `Content-Length:` field, which permits the server to determine when all the data has been received.

Uniform Resource Locators

Uniform Resource Locators (URLs) are strings that specify how to access network resources, such as HTML documents. They are part of the more general class of Universal Resource Identifiers (URIs). The most important use of URLs is in HTML documents to identify the targets of hyperlinks. When using a Web browser such as Netscape, every highlighted region has a URL associated with it, which is accessed when the link is activated by a mouse click. *Relative URLs* specify only a portion of the full URL - the missing information is inferred though the context of the source document.

URLs are documented in [RFC 1738](#). Relative URLs are documented in [RFC 1808](#). URIs are documented in [RFC 1630](#).

Here is a URL that describes the root page of the Internet Encyclopedia:

`http://www.FreeSoft.org/Connected/index.shtml`

The meaning of these fields is as follows:

`http`

The HyperText Transfer Protocol (HTTP) is to be used to retrieve the document. Other possible values for this field include `https` (use secure HTTP), `ftp` (use the File Transfer Protocol), and `gopher` (use the Gopher Protocol), among others.

`www.FreeSoft.org`

This is a hostname to be resolved using the Domain Name Service
`/Connected/index.shtml`

A directory and filename, to be passed along in the HTTP request to identify the document among many other on the server.

HTTP Tool

`http://www.httpviewer.net/`

Yahoo APIs

`http://developer.yahoo.com/`

Flickr API

`http://www.flickr.com/services/api/`

Google Maps

`http://code.google.com/apis/maps/`

SOAP

A SOAP request:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

The SOAP response:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

```
<?php
$homepage = file_get_contents('http://www.example.com/');
echo $homepage;
?>
```