HTTP: Hypertext Transfer Protocol

HTTP Operational Model and Client/Server Communication

The Hypertext Transfer Protocol is the application-layer protocol that implements the World Wide Web. While the Web itself has many different facets, HTTP is only concerned with one basic function: the transfer of hypertext documents and other files from Web servers to Web clients. In terms of actual communication, clients are chiefly concerned with making requests to servers, which respond to those requests.

Thus, even though HTTP includes a lot of functionality to meet the needs of clients and servers, when you boil it down, what see is a very simple, client/server, request/response protocol. In this respect, HTTP more closely resembles a rudimentary protocol like BOOTP or ARP than it does other application-layer protocols like FTP and SMTP, which all involve multiple communication steps and command/reply sequences.

Basic HTTP Client/Server Communication

In its simplest form, the operation of HTTP involves only an HTTP client, usually a *Web browser* on a client machine, and an HTTP server, more commonly known as a *Web server*. After a TCP connection is created, the two steps in communication are as follows:

1. **Client Request:** The HTTP client sends a request message formatted according to the rules of the HTTP standard—an *HTTP Request*. This message specifies the resource that the client wishes to retrieve, or includes information to be provided to the server.

2. **Server Response:** The server reads and interprets the request. It takes action relevant to the request and creates an *HTTP Response* message, which it sends back to the client. The response message indicates whether the request was successful, and may also contain the content of the resource that the client requested, if appropriate.

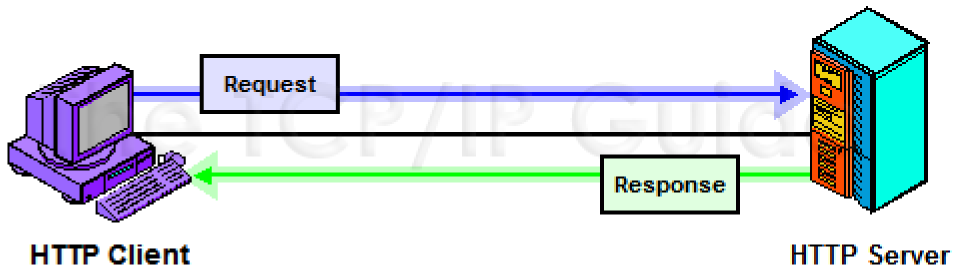


Figure 315: HTTP Client/Server Communication

In its simplest form, HTTP communication consists of an HTTP Request message sent by a client to a server, which replies with an HTTP Response.

In HTTP/1.0, each TCP connection involves only one such exchange, as shown in Figure 315; in HTTP/1.1, multiple exchanges are possible, as we'll see in the next topic. Note also that the server may in some cases respond with one or preliminary responses prior to sending the full response. This may occur if the server sends a preliminary response using the "100 Continue" status code prior to the "real" reply. See the topic on HTTP status codes for more information.

Key Concept: HTTP is a client/server-oriented, request/reply protocol. Basic communication consists of an *HTTP Request* message sent by an *HTTP client* to an *HTTP server*, which returns an *HTTP Response* message back to the client.

Intermediaries and The HTTP Request/Response Chain

The simple request/response pair between a client and server becomes more complex when *intermediaries* are placed in the virtual communication path between the client and server. These are devices such as *proxies*, *gateways* or *tunnels* that are used to improve performance, provide security or perform other necessary functions for particular clients or servers. Proxies are particularly commonly used on the Web, because they can greatly improve response time for groups of related client computers.

When an intermediary is involved in HTTP communication, it acts as a "middleman". Rather than the client speaking directly to the server and vice-versa, they each talk to the intermediary. This allows the intermediary to perform functions such as caching, translation, aggregation, or encapsulation. For example, consider an exchange through a single intermediary device. The two-step communication process above would become four steps:

1. **Client Request:** The HTTP client sends a request message to the intermediary device.

2. **Intermediary Request:** The intermediary processes the request, making changes to it if necessary. It then forwards the request to the actual server.

3. **Server Response:** The server reads and interprets the request, takes appropriate action and then sends a response. Since it received its request from the intermediary, its reply goes back to the intermediary.

4. **Intermediary Response:** The intermediary processes the request, again possibly making changes, and then forwards it back to the client.

As you can see, the intermediary acts as if it were a server from the client's perspective, and as a client from the server's viewpoint. Many intermediaries are designed to be able to "intercept" a variety of TCP/IP protocols, by "posing" as the server to a client and the client to a server. Most protocols are unaware of the existence of the interposition of an intermediary in this fashion. HTTP, however, includes special support for certain intermediaries such as proxy servers, providing headers that control how intermediaries handle HTTP requests and replies.

It is possible for two or more intermediaries to be linked together between the client and server. For example, the client might send a request to intermediary 1, which then forwards to intermediary 2, which then talks to the server; see Figure 316. The process is reversed for the reply. The HTTP standard uses the phrase *request/response chain* to refer collectively to the entire set of devices involved in an HTTP message exchange.

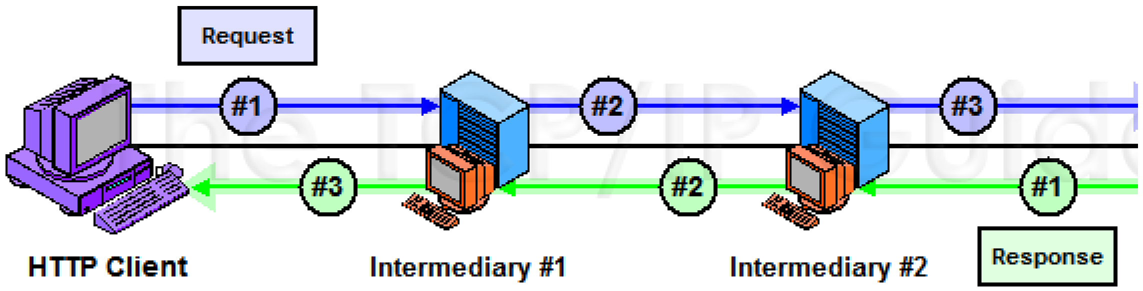


Figure 316: HTTP Request/Response Chain Using Intermediaries

Instead of being connected directly, an HTTP client and server may be linked using one or more intermediary devices such as proxies. In this example, two intermediaries are present. The *HTTP Request* sent by the client will actually be transferred three times: from the client to the first intermediary, then to the second, and finally to the server. The *HTTP Response* will likewise be created once but transmitted three distinct times. The full set of devices participating in the message exchange is called the request/response chain.

Key Concept: The simple client/server operational model of HTTP is complicated when *intermediary devices* such as proxies, tunnels or gateways are inserted in the communication path between the HTTP client and server. HTTP/1.1 is specifically designed with features to support the efficient conveyance of requests and responses through a series of steps from the client through the intermediaries to the server, and back again. The entire set of devices involved in such a communication is called the *request/response chain*.

The Impact of Caching on HTTP Communication

The normal HTTP communication model is changed through the application of *caching* to client requests. Caching is employed by various devices on the Web to store recently-retrieved resources so they can be quickly supplied in reply to a request. The client itself will cache recently-accessed Web documents so that if the user asks for them again they can be displayed without even making a request to a server. If a request is in fact required, any intermediary device can satisfy a request for a file if the file is in its cache.

When a cache is used, the device that has the cached resource requested returns it directly, "short-circuiting" the normal HTTP communication process. In the example above, if intermediary 1 has the file the client needs, it will supply it back to the client directly, and intermediary 2 and the real Web server that the client was trying to reach originally will not even be aware that a request was ever made; the topic on HTTP caching discusses the subject in much more detail.

Note: Most requests for Web resources are made using HTTP URLs based on a Domain Name System (DNS) host name. The first step in satisfying such requests is to resolve the DNS domain name into an IP address, but this process is separate from the HTTP communication itself.

HTTP Request Message Format

The client initiates an HTTP session by opening a TCP connection to the HTTP server with which it wishes to communicate. It then sends *request messages* to the server, each of which specifies a particular type of action that the user of the

HTTP client would like the server to take. Requests can be generated either by specific user action (such as clicking a hyperlink in a Web browser) or indirectly as a result of a prior action (such as a reference to an inline image in an HTML document leading to a request for that image.)

HTTP requests use a message format that is based on the generic message format described in the preceding topic, but specific to the needs of requests. The structure of this format is as follows (see Figure 317):

```
<request-line>
<general-headers>
<request-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

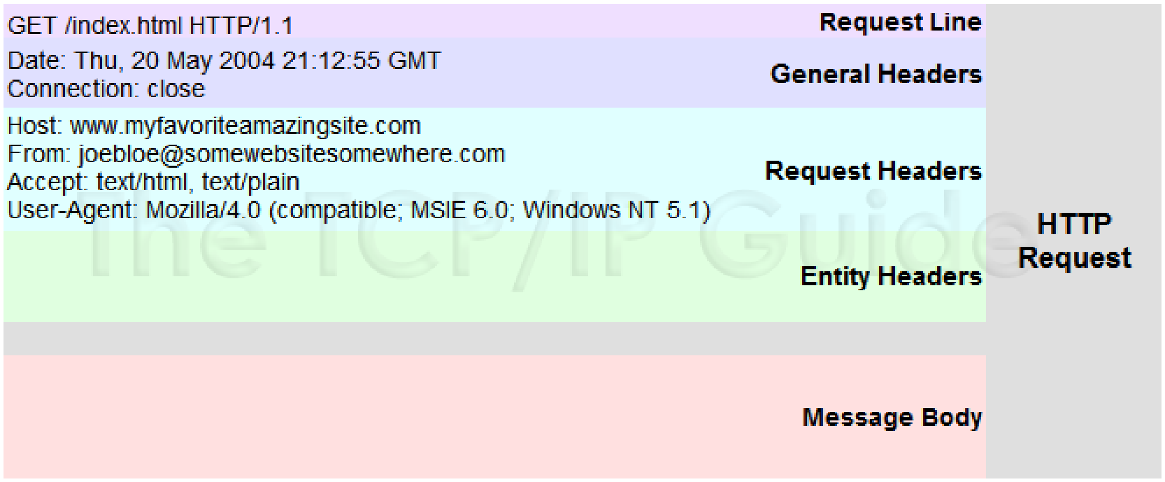

```
GET /index.html HTTP/1.1
Date: Thu, 20 May 2004 21:12:55 GMT
Connection: close
Host: www.myfavoriteamazingsite.com
From: joebloe@somewebsitesomewhere.com
Accept: text/html, text/plain
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

Figure 317: HTTP Request Message Format

This diagram shows the structural elements of an HTTP request and an example of the sorts of headers a request might contain. Like most HTTP requests, this one carries no entity, so there are no entity headers and the message body is empty. See Figure 318 for the HTTP response format.

Request Line

The generic *start line* that begins all HTTP messages is called a *request line* in request messages. Its has a three-fold purpose: to indicate the command or action that the client wants performed; to specify a resource upon which the action should be taken; and to indicate to the server what version of HTTP the client is using. The formal syntax for the request line is:

```
<METHOD> <request-uri> <HTTP-VERSION>
```

Method

The *method* is simply the type of action that the client wants the server to take; it is always specified in upper case letters. There are eight standard methods defined in HTTP/1.1, of which three are widely used: *GET*, *HEAD* and *POST*. They are called “methods” rather than “commands” because the HTTP standard uses terminology from object-oriented programming. I explain this and also describe the methods themselves in the topic describing HTTP methods.

Request URI

The *request URI* is the uniform resource identifier of the resource to which the request applies. While URIs can theoretically refer to either uniform resource locators (URLs) or uniform resource names (URNs), at the present time a URI is almost always an HTTP URL that follows the standard syntax rules of Web URLs.

Interestingly, the exact form of the URL used in the HTTP request line usually differs from that used in HTML documents or entered by users. This is because some of the information in a full URL is used to control HTTP itself. It is needed as part of the communication between the user and the HTTP client, but not in the request from the client to the server. The standard method of specifying a resource in a request is to include the path and file name in the request line (as well as any optional query information) while specifying the host in the special *Host* header that must be used in HTTP/1.1 requests.

For example, suppose the user enters a URL such as this:

```
http://www.myfavoritewebsite.com:8080/chatware/chatroom.php
```

We obviously don't need to send the “http:” to the server. The client would take the remaining information and split it so the URI was specified as “/chatware/chatroom.php” and the *Host* line would contain “www.myfavoritewebsite.com:8080”. Thus, the start of the request would look like this:

```
GET /chatware/chatroom.php HTTP/1.1
Host: www.myfavoritewebsite.com:8080
```

The exception to this rule is when a request is being made to a proxy server. In that case, the request is made using the full URL in its original form, so that it can be processed by the proxy just as the original client did. The request would be:

GET http://www.myfavoritewebsite.com:8080/chatware/chatroom.php HTTP/1.1

Finally, there is one special case where a single asterisk can be used instead of a real URL. This is for the *OPTIONS* method, which does not require the specification of a resource. (Nominally, the asterisk means the method refers to the server itself.)

HTTP Version

The *HTTP-VERSION* element tells the server what version the client is using so the server knows how to interpret the request, and what to send and not to send the client in its response. For example, a server receiving a request from a client using versions 0.9 or 1.0 will assume that a transitory connection is being used rather than a persistent one, and will avoid using version 1.1 headers in its reply. The version token is sent in upper case as “HTTP/0.9”, “HTTP/1.0” or “HTTP/1.1”—just the way I've been doing throughout my discussion of the protocol.

Headers

After the request line come any of the headers that the client wants to include in the message; it is in these headers that details are provided to the server about the request. The headers all use the same structure, but are organized into categories based on the functions they serve, and whether they are specific to one kind of message or not:

- **General Headers:** General headers refer mainly to the message itself, as opposed to its contents, and are used to control its processing or provide the recipient with extra information. They are not particular to either request or response messages, so they can appear in either. They are likewise not specifically relevant to any entity the message may be carrying.

- **Request Headers:** These headers convey to the server more details about the nature of the client's request, and give the client more control over how the request is handled. For example, special request headers can be used by the client to specify a conditional request—one that is only filled if certain criteria are met. Others can tell the server which formats or encodings the client is able to process in a response message.

- **Entity Headers:** These are headers that describe the entity contained in the body of the request, if any.

Request headers are obviously used only in request messages, but both general headers and entity headers can appear in either a request or a response message. Since there are so many headers and most are not particular to one message type, they are described in detail in their own complete section.

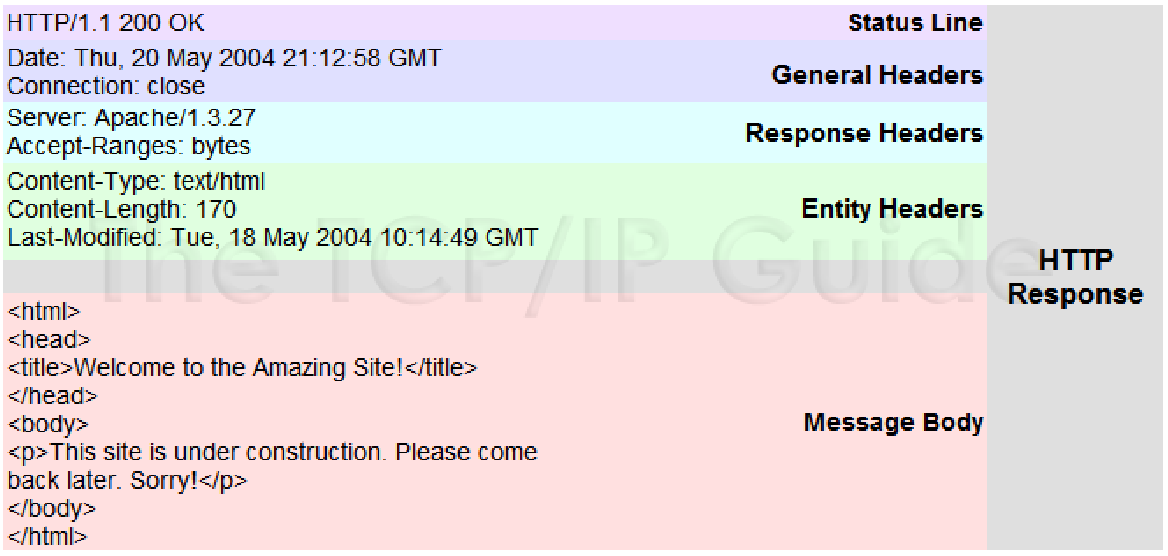
Key Concept: *HTTP requests* are the means by which HTTP clients ask servers to take a particular type of action, such as sending a file or processing user input. Each request message begins with a *request line*, which contains three critical pieces of information: the *method* (type of action) the client is requesting; the *URI* of the resource upon which the client wishes the action to be performed, and the version of HTTP that the client is using. After the request line come a set of message headers related to the request, followed by a blank line and then optionally, the message body

HTTP Response Message Format

Up and down; east and west; black and white; yin and yang. Well, you get the idea. Each request message sent by an HTTP client to a server prompts the server to send back a *response message*. Actually, in certain cases the server may in fact send two responses, a preliminary response followed by the real one. Usually though, one request yields one response, which indicates the results of the server's processing of the request, and often also carries an entity (file or resource) in the message body.

Like requests, responses use their own specific message format that is based on the HTTP generic message format. The format, shown in Figure 318, is:

```
<status-line>
<general-headers>
<response-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```



```
HTTP/1.1 200 OK
Date: Thu, 20 May 2004 21:12:58 GMT
Connection: close
Server: Apache/1.3.27
Accept-Ranges: bytes
Content-Type: text/html
Content-Length: 170
Last-Modified: Tue, 18 May 2004 10:14:49 GMT

<html>
<head>
<title>Welcome to the Amazing Site!</title>
</head>
<body>
<p>This site is under construction. Please come
back later. Sorry!</p>
</body>
</html>
```

Figure 318: HTTP Response Message Format

This figure illustrates the construction of an HTTP response, and includes an example of both message headers and body. The status code "200" indicates that this is a successful response to a request; it contains a brief text HTML entity in the message body. See Figure 317 for the HTTP request format.

Status Line

The *status line*—not "response line", note—is the start line used for response messages. It has two functions: to tell the client what version of the protocol the server is using, and to communicate a summary of the results of processing the client's request. The formal syntax for the status line is:

<HTTP-VERSION> <status-code> <reason-phrase>

HTTP Version

The *HTTP-VERSION* label in the status line serves the same purpose as it does in the request line of a request message; here, it tells the client the version number that the server is using for its response. It uses the same format as in the request line, in upper case as "HTTP/0.9", "HTTP/1.0" or "HTTP/1.1". The server is required to return an HTTP version number that is no greater than that the client sent in its request.

Status Code and Reason Phrase

The *status code* and *reason phrase* provide information about the results of processing the client's request in two different forms. The status code is a three-digit number that indicates the formal result that the server is communicating to the client; it is intended for the client HTTP implementation to process so the software can take appropriate action. The reason phrase is an additional, descriptive text string, which can be displayed to the human user of the HTTP client so he or she can see how the server responded. I describe status codes and reason phrases later in this section, and also list all of the standard codes.

Headers

The response message will always include a number of headers that provide extra information about it. Response message headers fall into these categories:

- **General Headers:** General headers that refer to the message itself and are not specific to response messages or the entity in the message body. These are the same as the generic headers that can appear in request messages (though certain headers appear more often in responses and others are more common in requests).

- **Response Headers:** These headers provide additional data that expands upon the summary result information in the status line. The server may also return extra result information in the body of the message, especially when an error occurs, as discussed below.

- **Entity Headers:** These are headers that describe the entity contained in the body of the response, if any. These are the same entity headers that can appear in a request message, but they are seen more often in response messages. The reason for this is simply that responses more often carry an entity than requests, because most requests are to retrieve a resource.

Note: Entity headers may appear in a response to describe the resource that is the subject of the request, even if the entity itself is not sent in the message. This occurs when the *HEAD* method is used to request only the headers associated with an entity.

Response headers are of course used only in response messages, while the others are general with respect to message type. See the section describing HTTP headers for more details.

Most response messages contain an entity in the message body. In the case of a successful request to retrieve a resource, this is the resource itself. Responses indicating unsuccessful requests usually contain detailed error information, often in the form of an HTML-formatted error message.

Key Concept: Each HTTP request sent by a client leads to a server returning one or more *HTTP responses*. Each response message starts with a *status line* that contains the server's HTTP version number, and a numeric *status code* and text *reason phrase* that indicate the result of processing the client's request. The message then contains headers related to the response, followed by a blank line and then the optional message body. Since most HTTP requests ask for a server to return a file or other resource, many HTTP responses carry an entity in the message body.

HTTP Methods

A request sent by a client to a server obviously… requests that the server do something. All client/server protocols provide a way for the client to prompt the server to take action, generally by having the client give the server a series of commands. HTTP, in contrast, does not have commands but rather *methods*. Each client request message begins with the specification of the method that is the subject of the request.

What is the difference between a method and a command? In practical terms, nothing; they are the same. So why does HTTP use the term "method" instead of "command"? Good question. ☺ The answer can be found in the abstract of the standard defining HTTP/1.0, RFC 1945. It states, in part, that HTTP is:

"a generic, stateless, object-oriented protocol which can be used for many tasks…"

In highly simplified terms, object-oriented programming is a technique in which software modules are described not as sets of procedures but as *objects* that possess attributes. These modules send messages to each other to communicate, and to cause actions to be performed, where the action taken depends on the nature of the object. In object-oriented programming, the procedures each object can perform are called *methods*.

HTTP is considered to be object-oriented because in many cases, the action taken by a server depends on the object that is the subject of the request. For example, if you ask a server to retrieve a text document, it will send that document, but if you ask for a directory, the server may instead return a default document for that directory. In contrast, a request that specifies the name of a program will result in the program being executed and its output returned (as opposed to the program's source code being returned.)

Common Methods

Each method allows the client to specify a particular type of action to be taken by the server. Method names are always in upper case letters. There are three methods that are commonly used in HTTP.

GET

The *GET* method requests that server retrieve the resource specified by the URL on the HTTP request line and send it in a response back to the client. This is the most basic type of request and the one that accounts for the majority of HTTP traffic. When you enter a conventional URL or click on a link to a document or other file, you are usually prompting your Web browser to send a *GET* request.

The handling of a *GET* request depends on a number of factors. If the URL is correct and the server can find the resource, it will of course send back the appropriate response to the client. As mentioned above, the exact resource returned depends on the nature of the object requested. If the request cannot be processed properly, an error message may result. Caching also comes into play, as a proxy server or even the client itself might satisfy the request before it gets to the server.

It's important to remember that the meaning of a *GET* request may change if certain headers, such as *If-Modified-Since* or *If-Match* are used—these and other similar headers tell the server to only send the resource if certain conditions are met. A request of this sort is sometimes called a *conditional GET*. Similarly, the *Range* header may be used by the client to request that the server send it only part of a resource; this is usually used for large files. When this header is included, the request may be called a *partial GET*.

HEAD

This is identical to the *GET* method, but tells the server to not send the actual body of the message. Thus, the response will contain all of the headers that would have accompanied a reply to the equivalent *GET* message, including entity headers describing the entity that the server would have sent had the method been *GET*. This method is often used by the client to check the existence, status or size of a file before deciding whether or not it wants the server to send the whole thing.

HEAD requests are processed in the same way as *GET* requests, except that only the headers are returned, not the actual resource.

POST

This method allows the client to send an entity containing arbitrary data to the server for processing. It is commonly used to enable a client to submit information such as an interactive HTML form to a program on the server, which then takes action based on that input and sends a response. This capability is now used for all sorts of online programs. The URL in the request specifies the name of the program on the server that is to accept the data.

Contrast this with the *PUT* method below.

Other, Less-Common Methods

The other methods defined by the HTTP standard are not used as often, but I will describe them briefly, as you may still encounter them from time to time.

OPTIONS

Allows the client to request that the server send it information about available communication options. A URI of a resource may be specified to request information relevant to accessing that resource, or an asterisk ("*") may be used to indicate that the query is about the server itself. The response includes headers that give the client more details about how the server may be accessed.

PUT

Requests that the server store the entity enclosed in the body of the request at the URL specified in the request line. The difference between a *PUT* and a *POST* is that in a *PUT*, the URI identifies the entity in the request, while in a *POST*, the URI identifies a program intended to ***process*** the entity in the request. Thus a *PUT* would be used to allow a file to be copied to a server, in the exact complement to how a *GET* requests that a file be copied to the client. A *POST* is used for interactive programs, as explained above.

Now, would you like people to be able to store files on your server in the same way that they request them? Neither would I. This is one primary reason why *PUT* is not often used. It has valid uses, such as uploading content to a Web site, and must be used with authentication in this case. However, generally speaking, storing files on a site is more often accomplished using other means, like FTP.

DELETE

Requests that the specified resource be deleted. This has the same issues as *PUT* and is not often used for similar reasons.

TRACE

Allows a client to receive back a copy of the request that it itself sent to the server, for diagnostic purposes.

Note: In addition to the methods described above, the HTTP standard reserves the method name *CONNECT* for future use. An earlier version of HTTP/1.1, RFC 2068, also defined the methods *PATCH*, *LINK* and *UNLINK*. These were removed in the final version but reference to them is still sometimes seen.

Key Concept: Each HTTP client request specifies a particular type of action that the server perform; in HTTP, these are called not commands but rather *methods*. The three most common HTTP methods are *GET*, which prompts a server to return a resource; *HEAD*, which returns just the headers associated with a resource; and *PUT*, which allows a client to submit data to a server for processing.

"Safe" and Idempotent Method Categorizations

As we've seen above, methods vary greatly in the type of behavior they cause the server to take. The HTTP standard defines two characteristics that can be used to differentiate methods based on the impact they have on a server:

- **"Safe" Methods:** These are methods that an administrator of a server can feel reasonably comfortable permitting a client to send because they are very unlikely to have any negative "side-effects". The methods usually put into this category are *GET*, *HEAD*, *OPTIONS* and *TRACE*. The methods that cause data to be accepted by the server for processing, or lead to changes on the server, are deemed "unsafe": *POST*, *PUT*, and *DELETE*. (The fact that they are "unsafe" doesn't mean a server never allows them, just that they require more care and detail in handling than the others.)

- **Idempotent Methods:** Woah, 10-dollar word time. ☺ A method is said to be *idempotent* if repeating the same method request numerous times causes the exact same results as if the method were issued only once.

> For example, if you load a Web page in your browser, and then type the same URL in again, you get the same result, at least most of the time. In general, all of the methods in HTTP have this property inherently except one: *POST*.

The *POST* method is not idempotent because each instance of a *POST* request causes the receiving server to process the data in the request's body. Submitting a POST request two or more times can often lead to undesirable results. The classic example is hitting the “submit” button on a form more than once, which can lead to annoyances such as a duplicate message on an Internet forum, or even a double order at an online store.

There are also situations where a method that is normally idempotent may not be. A *GET* request for a simple document is idempotent, but a *GET* for a script can change files on the server and therefore is not idempotent. Similarly, a sequence of idempotent methods can be non-idempotent. For example, consider a situation where a *PUT* request is followed by a *GET* for the same resource. This sequence is non-idempotent because the second request depends on the results of the first.

The significance of non-idempotence is that clients must handle such requests or sequences specially. The client must keep track of them, and make sure that they are filled in order and only once. The HTTP standard also specifies that non-idempotent methods should not be pipelined, to avoid problems if an HTTP session is unexpectedly terminated. For example, if two POST requests were pipelined and the server got hung up handling them, the client would need to reissue them but might not know how many of the originals had been successfully processed.

HTTP Status Code Format, Status Codes and Reason Phrases

Every request sent by an HTTP client causes one (or more) responses to be returned by the server that receives it. As we saw in the topic describing the response message format, the first line of the response is a status line that contains a summary of the results of processing the request. The purpose of this line is to communicate quickly whether or not the request was successful, and why.

HTTP status lines contain both a numeric *status code* and a text *reason phrase*. The idea behind this was taken directly from earlier application layer protocols such as FTP, SMTP and NNTP. The reason for having both a number and a text string is that computers can more easily “understand” the results of a request by looking at a number, and can then quickly respond accordingly. Humans, on the other hand, find numbers cryptic and text descriptions easier to comprehend. (The topic on FTP reply codes discusses more completely the reasons why numeric reply codes are used in addition to descriptive text.)

Status Code Format

HTTP status codes are three digits in length and follow a particular format where the first digit has particular significance. Unlike FTP and the others, the second digit does not stand for a functional grouping; the second and third digits together just make 100 different options for each of the categories indicated by the first digit. Thus, the general form of an HTTP status code is “xyy”, where the first digit, “x”, is specified as given in Table 274.

Table 274: HTTP Status Code Format: First Digit Interpretation

Status Code Format	Meaning	Description
1yy	**Informational Message**	Provides general information; does not indicate success or failure of a request.
2yy	**Success**	The method was received, understood and accepted by the server.
3yy	**Redirection**	The request did not fail outright, but additional action is needed before it can be successfully completed.
4yy	**Client Error**	The request was invalid, contained bad syntax or could not be completed for some other reason that the server believes was the client's fault.
5yy	**Server Error**	The request was valid but the server was unable to complete it due to a problem of its own.

In each of these five groups, the code where “yy” is “00” is defined as a “generic” status code for that group, while other two-digit combinations are more specific responses. For example, “404” is the well-known specific error message that means the requested resource was not found by the server, while “400” is the less specific “bad request” error. This system was set up to allow the definition of new status codes that certain clients might not comprehend. If a client receives a strange code, it just treats it as the equivalent of the generic response in the appropriate category. So if a server response starts with the code "491" and the client has no idea what this is, it just treats it as a 400 "bad request" reply.

Reason Phrases

The reason phrase is a text string that provides a more meaningful description of the error for people who are bad at remembering what cryptic codes stand for (which would be most of us!) The HTTP standard includes "sample" reason phrases for each status code, but these can be customized by the administrators of a server if desired. When a server returns a more detailed HTML error message in the body of its response message, the reason phrase is often used for the "title" tag in that message body.

Key Concept: Each HTTP response includes both a numeric *status code* and a text *reason phrase*, both of which indicate the disposition of the corresponding client request. The numeric code allows software programs to easily interpret the results of a request, while the text phrase provides more useful information to human users. HTTP status codes are three digits in length, with the first digit indicating the general class of the reply.

Status Codes and Reason Phrases

Table 275 lists in numerical order the status codes defined by the HTTP/1.1 standard, along with the "standard" reason phrase and a brief description of each:

Table 275: HTTP Status Codes

Status Code	Reason Phrase	Description
100	Continue	Client should continue sending its request. This is a special status code; see below for details.
101	Switching Protocols	The client has used the *Upgrade* header to request the use of an alternative protocol and the server has agreed.
200	OK	Generic successful request message response. This is the code sent most often when a request is filled normally.

201	Created	The request was successful and resulted in a resource being created. This would be a typical response to a *PUT* method.
202	Accepted	The request was accepted by the server but has not yet been processed. This is an intentionally "non-commital" response that does not tell the client whether or not the request will be carried out; the client determines the eventual disposition of the request in some unspecified way. It is used only in special circumstances.
203	Non-Authoritative Information	The request was successful, but some of the information returned by the server came not from the original server associated with the resource but from a third party.
204	No Content	The request was successful, but the server has determined that it does not need to return to the client an entity body.
205	Reset Content	The request was successful; the server is telling the client that it should reset the document from which the request was generated so that a duplicate request is not sent. This code is intended for use with forms.
206	Partial Content	The server has successfully fulfilled a partial GET request. See the topic on methods for more details on this, as well as the description of the Range header.
300	Multiple Choices	The resource is represented in more than one way on the server. The server is returning information describing these representations, so the client can pick the most appropriate one, a process called agent-driven negotiation.
301	Moved Permanently	The resource requested has been moved to a new URL permanently. Any future requests for this resource should use the new URL. This is the proper method of handling situations where a file on a server is renamed or moved to a new directory. Most people don't bother setting this

		up, which is why URLs “break” so often, resulting in 404 errors as discussed below.
302	Found	The resource requested is temporarily using a different URL. The client should continue to use the original URL. See code 307.
303	See Other	The response for the request can be found at a different URL, which the server specifies. The client must do a fresh *GET* on that URL to see the results of the prior request.
304	Not Modified	The client sent a conditional *GET* request, but the resource has not been modified since the specified date/time, so the server has not sent it.
305	Use Proxy	To access the requested resource, the client must use a proxy, whose URL is given by the server in its response.
306	(unused)	Defined in an earlier (draft?) version of HTTP and no longer used.
307	Temporary Redirect	The resource is temporarily located at a different URL than the one the client specified. Note that 302 and 307 are basically the same status code. 307 was created to clear up some confusion related to 302 that occurred in earlier versions of HTTP (which I'd rather not get into!)
400	Bad Request	Server says, “huh?” ☺ Generic response when the request cannot be understood or carried out due to a problem on the client's end.
401	Unauthorized	The client is not authorized to access the resource. Often returned if an attempt is made to access a resource protected by a password or some other means without the appropriate credentials.
402	Payment Required	This is reserved for future use. Its mere presence in the HTTP standard has caused a lot of people to scratch their chins and go “hmm…” ☺
403	Forbidden	The request has been disallowed by the server. This is a generic “no way” response that is not related to

		authorization. For example, if the maintainer of Web site blocks access to it from a particular client, any requests from that client will result in a 403 reply.
404	Not Found	The most common HTTP error message, returned when the server cannot locate the requested resource. Usually occurs due to either the server having moved/removed the resource, or the client giving an invalid URL (misspellings being the most common cause.)
405	Method Not Allowed	The requested method is not allowed for the specified resource. The response includes an *Allow* header that indicates what methods the server will permit.
406	Not Acceptable	The client sent a request that specifies limitations that the server cannot meet for the specified resource. This error may occur if an overly-restrictive list of conditions is placed into a request such that the server cannot return any part of the resource.
407	Proxy Authentication Required	Similar to 401, but the client must first authenticate itself with the proxy.
408	Request Timeout	The server was expecting the client to send a request within a particular time frame and the client didn't send it.
409	Conflict	The request could not be filled because of a conflict of some sort related to the resource. This most often occurs in response to a *PUT* method, such as if one user tries to *PUT* a resource that another user has open for editing, for example.
410	Gone	The resource is no longer available at the server, which does not know its new URL. This is a more specific version of the 404 code that is used only if the server knows that the resource was intentionally removed. It is seen rarely (if ever) compared to 404.
411	Length Required	The request requires a Content-Length header field and one was not included.
412	Precondition	Indicates that the client specified a precondition in its

	Failed	request, such as the use of an *If-Match* header, which evaluated to a false value. This indicates that the condition was not satisfied so the request is not being filled. This is used by clients in special cases to ensure that they do not accidentally receive the wrong resource.
413	Request Entity Too Large	The server has refused to fulfill the request because the entity that the client is requesting is too large.
414	Request-URI Too Long	The server has refused to fulfill the request because the URL specified is longer than the server can process. This rarely occurs with properly-formed URLs but may be seen if clients try to send gibberish to the server.
415	Unsupported Media Type	The request cannot be processed because it contains an entity using a media type the server does not support.
416	Requested Range Not Satisfiable	The client included a *Range* header specifying a range of values that is not valid for the resource. An example might be requesting bytes 3,000 through 4,000 of a 2,400-byte file.
417	Expectation Failed	The request included an *Expect* header that could not be satisfied by the server.
500	Internal Server Error	Generic error message indicating that the request could not be fulfilled due to a server problem.
501	Not Implemented	The server does not know how to carry out the request, so it cannot satisfy it.
502	Bad Gateway	The server, while acting as a gateway or proxy, received an invalid response from another server it tried to access on the client's behalf.
503	Service Unavailable	The server is temporarily unable to fulfill the request for internal reasons. This is often returned when a server is overloaded or down for maintenance.
504	Gateway Timeout	The server, while acting as a gateway or proxy, timed out while waiting for a response from another server it tried to access on the client's behalf.

505	HTTP Version Not Supported	The request used a version of HTTP that the server does not understand.

The 100 (Continue) Preliminary Reply

Phew. Now, let's go back to the top, status code 100. Normally, a client sends a complete request to the server, and waits for a response to it (while optionally pipelining additional requests). In certain circumstances, however, the client might wish to check in advance if the server is willing to accept the request before it bothers sending the whole message. This is not a common occurrence, because most requests are quite small, which makes it not worth the bother. However, in cases where a user wants to submit a very large amount of data to an online program, or use *PUT* to store a large file, for example, checking with the server first can be a useful optimization.

In this situation, the client sends a request containing the special header "*Expect: 100-continue*". Assuming that the server supports the feature, it will process the request's headers and immediately send back the "*100 Continue*" preliminary reply. This tells the client to continue sending the rest of the request. The server then processes it and responds normally. If the server doesn't send the 100 response after a certain amount of time, the client will typically just send the rest of the request anyway.

Note: In some cases, servers send these preliminary replies even when they are not supposed to, so clients must be prepared to deal with them (they are simply discarded, since they contain no information).

HTTP Request Headers

HTTP *request headers*, as you might imagine, are used only in HTTP request messages, and serve a number of functions in them. First, they allow the client to provide information about itself to the server. Second, they give additional details about the nature of the request that the client is making. Third, they allow the client to have greater control over how its request is processed and how (or even if) a response is returned by the server or intermediary.

This is the largest of the four categories of HTTP headers, comprising over a dozen different types.

Accept

Allows the client to tell the server what Internet media types it is willing to accept in a response. The header may list several different MIME media types and subtypes that the client knows how to deal with. Each may be prepended with a “quality value” (“q” parameter) to indicate the client's preference. If this header is not specified, the default is for the server to assume any media type may be sent to the client. See the topics on entity media types and content negotiation for more information on how this header is used.

Accept-Charset

Similar to *Accept*, but specifies what character sets the client is willing to accept in a response, rather than what media types. Again, the listed charsets may use a “q” value, and again, the default if the header is omitted is for the client to accept any character set.

Accept-Encoding

Similar to *Accept* and *Accept-Charset*, but specifies what content encodings the client is willing to accept. This is often used to control whether or not the server may send content in compressed form. (Remember that content codings are not the same as transfer encodings.)

Accept-Language

Similar to the preceding *Accept*-type headers, but provides a list of *language tags* that indicate what languages the client supports or expects the server to use in its response.

Authorization

Used by the client to present authentication information (called “credentials”) to the server to allow the client to be authenticated. This is required only when the server requests authentication, often by sending a 401 (“*Unauthorized*”) response to the client's initial request. This response will contain a *WWW-Authenticate* header providing the client with details on how to authenticate with the server. See the topic on security and privacy.

Expect

Indicates certain types of actions that the client is expecting the server to perform. Usually the server will accept the indicated parameters; if not, it will send back a 417 (“*Expectation Failed*”) response. The most common use of this field is to control when the server sends a 100 (“*Continue*”) response. The client indicates that it wants the server to send this preliminary reply by including the "*Expect: 100-continue*" header in its request. (See the end of the topic on status codes for details.)

From

Contains the e-mail address of the human user making the request. This is optional, and since it is easily spoofed, should be used only for informational purposes, and not for any type of access rights determination or authentication.

Host

Specifies the Internet host as a DNS domain name, and may also contain a port number specification as well (typically, only if a port other than the HTTP default of 80 is to be used). This header is used to allow multiple domains to be served by the same Web server on a particular IP host. It has the distinction of being the only ***mandatory*** header—it must be present in all HTTP/1.1 requests.

If-Match

Makes a method conditional by specifying the *entity tag* (or tags) corresponding to the specific entity that the client wishes to access. This is usually used in a *GET* method, and the server responds with the entity only if it matches the one specified in this header. Otherwise, a 412 ("*Precondition Failed*") reply is sent.

If-Modified-Since

Makes a method conditional by telling the server to return the requested entity only if it has been modified since the time specified in this header. Otherwise, the server sends a 304 ("*Not Modified*") response. This is used to check if a resource has changed since it was last accessed, to avoid unnecessary transfers.

If-None-Match

This is the opposite of *If-Match*; it creates a conditional request that is only filled if the specified tag(s) do ***not*** match the requested entity.

If-Range

This header is used in combination with the *Range* header to effectively allow a client to both check for whether an entity has changed and request that a portion of it be sent in a single request. (The alternative is to first issue a conditional request, which if it fails would then require a second request.) When present, *If-*

Range tells the server to send to the client the part of the entity indicated in the *Range* header if the entity has not changed. If the entity ***has*** changed, the entire entity is sent in response.

If-Unmodified-Since

The logical opposite of the *If-Modified-Since* header; the request is filled only if the resource has *not* been modified since the specified time; otherwise a 412 reply is sent.

Max-Forwards

Specifies a limit on the number of times a request can be forwarded to the next device in the request chain. This header is used with the *TRACE* or *OPTIONS* methods only, to permit diagnosis of forwarding failures or looping. When present in one of these methods, each time a device forwards the request, the number in this header is decremented. If a device receives a request with a *Max-Forwards* value of 0, it must not forward it but rather respond back to the client. (In a way, this is somewhat analogous to how the *Time To Live* field is used in the Internet Protocol datagram format.)

Proxy-Authorization

This is like the *Authorization* header, but is used to present credentials to a proxy server for authentication, rather than the end server. It is created using information sent by a proxy in a response containing a *Proxy-Authenticate* header. This is a hop-by-hop header, sent only to the first proxy that receives the request. If authentication is required with more than one proxy, multiple *Proxy-Authorization* headers may be put in a message, with each proxy in turn "consuming" one of the headers.

Range

Allows the client to request that the server send it only a portion of an entity, by specifying a range of bytes in the entity to be retrieved. If the requested range is valid, the server sends only the indicated part of the file, using a 206 ("*Partial Content*") status code; if the range requested cannot be filled, the reply is 416 ("*Requested Range Not Satisfiable*").

Referer [sic]

Tells the server the URL of the resource from which the URL of the current request was obtained. Typically, when a user clicks a link on one Web page to load another, the address of the original Web page is put into the *Referer* line when the request for the clicked link is sent. This allows tracking and logging of how the server is accessed. If a human user manually enters a URI into a Web browser, this header is not included in the request. Since this header provides information related to how Web pages are used, it has certain privacy implications.

The proper spelling of this word is "referrer". It was misspelled years ago in an earlier version of the HTTP standard, and before this was noticed and corrected, became incorporated into so much software that the IETF chose not to correct the spelling in HTTP/1.1.

TE

Provides information to the server about how the client wishes to deal with transfer encodings for entities sent by the server. If extensions to the standard HTTP transfer encodings are defined, the client can indicate its willingness to accept them in this header. The header "*TE: trailers*" can also be used by the client to indicate its ability to handle having headers sent as trailers following data when "chunking" of data is done. This is a hop-by-hop header and applies only to the immediate connection.

User-Agent

Provides information about the client software. This is normally the name and version number of the Web browser or other program sending the request. It is used for server access statistic logging and also may be used to tailor how the server responds to the needs of different clients. Note that proxies do not modify this field when forwarding a request; rather, they use the *Via* header.

Key Concept: HTTP *request headers* are used only in *HTTP Request* messages. They allow a client to provide information about itself to a server, and provide more details about a request and control over how it is carried out.

HTTP Response Headers

The counterpart to request headers, *response headers* appear only in HTTP responses sent by servers or intermediaries. They provide additional data that expands upon the summary information that is present in the status line at the beginning of each server reply. Many of the response headers are sent only in response to the receipt of specific types of requests, or even particular headers within certain requests.

There are nine response headers defined for HTTP/1.1.

Accept-Ranges

Tells the client whether or not the server accepts partial content requests using the *Range* request header, and if so, what type. Typical examples include "*Accept-Range: bytes*" if the server accepts byte ranges, or "*Accept-Range: none*" if range requests are not supported.

Note that this is header is different from the other "Accept-" headers, which are used in HTTP requests to perform content negotiation.

Age

Tells the client the approximate age of the resource, as calculated by the device sending the response.

ETag

Specifies the entity tag for the entity included in the response. This value can be used by the client in future requests to uniquely identify an entity, using the *If-Match* request header or similar.

Location

Indicates a new URL that the server is instructing the client to use in place of the one the client initially requested. This header is normally used when the server redirects a client request to a new location, using a 301, 302 or 307 reply. It is also used to indicate the location of a created resource in a 201 ("Created") response to a *PUT* request.

Note that this is not the same as the *Content-Location* entity header, which is used to indicate the location of the originally-requested resource.

Proxy-Authenticate

This is the proxy version of the *WWW-Authenticate* header (see below). It is included in a 407 ("*Proxy Authentication Required*") response, to indicate how the proxy is requiring the client to perform authentication. The header specifies an authentication method as well as any other parameters needed for authentication. The client will use this to generate a new request containing a *Proxy-Authorization* header. This is a hop-by-hop header.

Retry-After

This header is sometimes included in unsuccessful requests—such as those resulting in a 503 ("*Service Unavailable*") response—to tell the client when it should try its request again. It may also be used with a redirection response such as 301, 302 or 307, to indicate how long the client should wait before sending a request for the redirected URL. The *Retry-After* header may specify either a time interval to wait (in seconds) or a full date/time when the server suggests the client try again.

Server

This is the server's version of the *User-Agent* request header; it identifies the type and version of the server software generating the response. Note that proxies do not modify this field when forwarding a response; they put their identification information into a *Via* header instead.

Vary

Specifies which request header fields fully determine whether a cache is allowed to use this response to reply to subsequent requests for the same resource without revalidation. A caching device inspects the *Vary* header to determine which other headers it needs to examine to determine whether or not it can respond with a cached entry, when the client makes its next request for the resource in this reply. Yeah, this one's a bit confusing.

WWW-Authenticate

This header is included in a 401 ("*Unauthorized*") response, to indicate how the server wants the client to authenticate. The header specifies an authentication method as well as any other parameters needed for authentication. The client will use this to generate a new request containing an *Authorization* header.

Key Concept: HTTP *response headers* appear in *HTTP Response* messages, where they provide additional information about HTTP server capabilities and requirements, and the results of processing a client request.

HTTP Entity Headers

Last but not least, we come to the fourth group of HTTP headers: *entity headers*. These headers provide information about the resource carried in the body of an HTTP message, called an *entity* in the HTTP standards. They serve the overall purpose of conveying to the recipient of a message the information it needs to properly process and display the entity, such as its type and encoding method.

The most common type of entity is a file or other set of information that has been requested by a client, and for this reason, entity headers most often appear in HTTP responses. However, they can also appear in HTTP requests, especially those using the *PUT* and *POST* methods, which are the ones that transfer data from a client to a server.

At least one entity header should appear in any HTTP message that carries an entity. However, they may also be present in certain responses that do not have an actual entity in them. Most notably, a response to a *HEAD* request will contain all the entity headers associated with the resource specified in the request; these are the same headers that would have been included with the entity, had the *GET* method been used instead of *HEAD* on the same resource. Entity headers may also be present in certain error responses, to provide information to help the client make a successful follow-up request.

Note: Many of the entity headers have the same names as certain MIME headers, but they are often used in different ways. See the topic on HTTP Internet media types for a full discussion of the relationship between HTTP and MIME.

The following are the entity headers defined in HTTP/1.1:

Allow

Lists all the methods that are supported for a particular resource. This header may be provided in a server response as a guide to the client regarding what methods it may use on the resource in the future. The header ***must*** be included when a server returns a 405 ("*Method Not Allowed*") response to a request containing an unsupported method.

Content-Encoding

Describes any optional method that may have been used to encode the entity. This header is most often used when transferring entities that have been compressed; it tells the recipient what algorithm has been used so the entity can be uncompressed. Note that this header only describes transformations performed on the entity in a message; the *Transfer-Encoding* header describes encodings done on the message as a whole. See the topic on content codings and transfer codings for more details.

Content-Language

Specifies the natural (human) language intended for using the entity. This is an optional header, and may not be appropriate for all resource types. Multiple languages may be specified, if needed.

This header is intended to provide guidance so the entity can be presented to the correct audience; thus, the language should be selected based on who would best use the material, which may not necessarily include all of the languages used in the entity. For example, a German analysis of Italian operas would probably best tagged only with the language "de". (They do have German analyses of Italian operas, don't they? ☺)

Content-Length

Indicates the size of the entity in octets. This header is important, as it is used by the recipient to determine the end of a message. However, it may only be included in cases where the length of a message can be fully determined prior to transmitting the entity. This is not always possible in the case of dynamically-generated content, which complicates message length calculation; the discussion of data length and the "chunked" transfer encoding contains a full exploration of this issue.

Content-Location

Specifies the resource location of the entity, in the form of an absolute or relative uniform resource locator (URL). This is an optional header, and is normally included only in cases where the entity has been supplied from a location different from the one specified in the request. This may occur if a particular resource is stored in multiple places.

Content-MD5

Contains an MD5 digest for the entity, used for message integrity checking.

Content-Range

Sent when a message contains an entity that is only part of a complete resource; for example, a fragment of a file sent in response to an HTTP *GET* request containing the *Range* header. The *Content-Range* header Indicates what portion of the overall file this message contains, as well as the total size of the resource. This information is given as a byte range, with the first byte numbered 0; for example, if the entity contains the first 1,200 bytes of a 2,000-byte file, this header would have a value of “0-1199/2000”.

Content-Type

Specifies the media type and subtype of the entity, in a manner very similar to how this header is used in MIME. See the topic describing HTTP entities and Internet media types for a full discussion.

Expires

Specifies a date and time after which the entity in the message should be considered “stale”. This may be used to identify certain entities that should be held in HTTP caches for longer or shorter periods of time than usual.

This header is ignored if a *Cache-Control* header containing the “max-age” directive is present in the message.

Last-Modified

Indicates the date and time when the server believes the entity was last changed. This header is often used to determine if a resource has been modified since it was last retrieved. For example, suppose a client machine already contains a copy of a very large file that was obtained two months ago, and its user wants to check if an update to the file is available. The client can send a *HEAD* request for the file, and compare the value of the returned *Last-Modified* header to the date of the copy of the file it already has. Then it needs only to request the entire file if it has changed.

Note the use of the word “believe” in the description above. The reason for this wording is that the server cannot always be certain of the time that a resource was modified. With files this is fairly simple—it is usually the last-modified time stored for the file by the operating system. For other more complex resources such as database records or virtual objects, however, it may be more difficult to ascertain when the last change occurred to a particular piece of information. In the case of dynamically-generated content, the *Last-Modified* date/time may be the same as that of the message as a whole, as specified in the *Date* field.

Key Concept: HTTP *entity headers* appear in either request or response messages that carry an entity in the message body. They describe the nature of the entity, including its type, language and encoding, to facilitate the proper processing and presentation of the entity by the device receiving it.