WEBSERV

1. **"Message Syntax and Routing" [RFC7230]**

2. **"Semantics and Content" [RFC7231]**

3. **"Conditional Requests" [RFC7232]**

4. **"Range Requests" [RFC7233]**

5. **"Caching" [RFC7234]**

6. **"Authentication" [RFC7235]**

==========

RFC 7230

👉 **HEADER**

→ **Name of a part**

==========

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a stateless application- level request/response protocol that uses extensible semantics and self-descriptive message payloads for flexible interaction with network-based hypertext information systems. This document is the first in a series of documents that collectively form the HTTP/1.1 specification:

HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided. Likewise, servers do not need to be aware of each client's purpose: an HTTP request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. The result is a protocol that can be used effectively in many different contexts and for which implementations can evolve independently over time.

HTTP is also designed for use as an intermediation protocol for translating communication to and from non-HTTP information systems. HTTP proxies and gateways can provide access to alternative information services by translating their diverse protocols into a hypertext format that can be viewed and manipulated by clients in the same way as HTTP services.

One consequence of this flexibility is that the protocol cannot be defined in terms of what occurs behind the interface. Instead, we are limited to defining the syntax of communication, the intent of received communication, and the expected behavior of recipients. If the communication is considered in isolation, then successful actions ought to be reflected in corresponding changes to the observable interface provided by servers. However, since multiple clients might act in parallel and perhaps at cross-purposes, we cannot require that such changes be observable beyond the scope of a single response.

This document describes the architectural elements that are used or referred to in HTTP, defines the "http" and "https" URI schemes, describes overall network operation and connection management, and defines HTTP message framing and forwarding requirements. Our goal is to define all of the mechanisms necessary for HTTP message handling that are independent of message semantics, thereby defining the complete set of requirements for message parsers and message- forwarding intermediaries.

2. Architecture

HTTP was created for the World Wide Web (WWW) architecture and has evolved over time to support the scalability needs of a worldwide hypertext system. Much of that architecture is reflected in the terminology and syntax productions used to define HTTP.

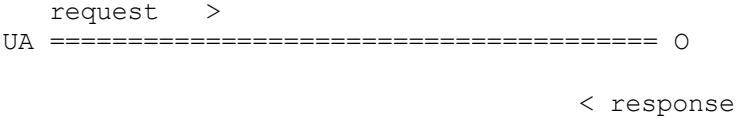
2.1. Client/Server Messaging

HTTP is a stateless request/response protocol that operates by exchanging messages across a reliable transport- or session-layer "connection". An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

The terms "client" and "server" refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others. The term "user agent" refers to any of the various client programs that initiate a request, including (but not limited to) browsers, spiders (web-based robots), command-line tools, custom applications, and mobile apps. The term "origin server" refers to the program that can originate authoritative responses for a given target resource. The terms "sender" and "recipient" refer to any implementation that sends or receives a given message, respectively.

HTTP relies upon the Uniform Resource Identifier (URI) standard to indicate the target resource and relationships between resources. Messages are passed in a format similar to that used by Internet mail and the Multipurpose Internet Mail Extensions (MIME) (see Appendix A of [RFC7231] for the differences between HTTP and MIME messages).

Most HTTP communication consists of a retrieval request (GET) for a representation of some resource identified by a URI. In the simplest case, this might be accomplished via a single bidirectional connection (===) between the user agent (UA) and the origin server (O).



A client sends an HTTP request to a server in the form of a request message, beginning with a request-line that includes a method, URI, and protocol version, followed by header fields containing request modifiers, client information, and representation metadata (Section 3.2), an empty line to indicate the end of the header section, and finally a message body containing the payload body.

A server responds to a client's request by sending one or more HTTP response messages, each beginning with a status line that includes the protocol version, a success or error code, and textual reason phrase, possibly followed by header fields containing server information, resource metadata, and representation metadata, an empty line to indicate the end of the header section, and finally a message body containing the payload body.

A connection might be used for multiple request/response exchanges.

The following example illustrates a typical message exchange for a GET request on the URI "http://www.example.com/hello.txt":

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

```
Hello World! My payload includes a trailing CRLF.
```

2.2. Implementation Diversity

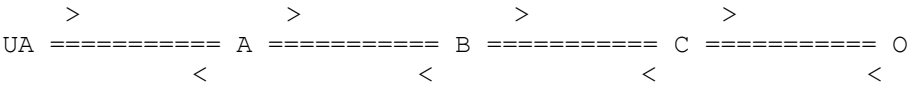
When considering the design of HTTP, it is easy to fall into a trap of thinking that all user agents are general-purpose browsers and all origin servers are large public websites. That is not the case in practice. Common HTTP user agents include household appliances, stereos, scales, firmware update scripts, command-line programs, mobile apps, and communication devices in a multitude of shapes and sizes. Likewise, common HTTP origin servers include home automation units, configurable networking components, office machines, autonomous robots, news feeds, traffic cameras, ad selectors, and video-delivery platforms.

The term "user agent" does not imply that there is a human user directly interacting with the software agent at the time of a request. In many cases, a user agent is installed or configured to run in the background and save its results for later inspection (or save only a subset of those results that might be interesting or erroneous). Spiders, for example, are typically given a start URI and configured to follow certain behavior while crawling the Web as a hypertext graph.

The implementation diversity of HTTP means that not all user agents can make interactive suggestions to their user or provide adequate warning for security or privacy concerns. In the few cases where this specification requires reporting of errors to the user, it is acceptable for such reporting to only be observable in an error console or log file. Likewise, requirements that an automated action be confirmed by the user before proceeding might be met via advance configuration choices, run-time options, or simple avoidance of the unsafe action; confirmation does not imply any specific user interface or interruption of normal processing if the user has already made that choice.

2.3. Intermediaries

HTTP enables the use of intermediaries to satisfy requests through a chain of connections. There are three common forms of HTTP intermediary: proxy, gateway, and tunnel. In some cases, a single intermediary might act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.



The figure above shows three intermediaries (A, B, and C) between the user agent and origin server. A request or response message that travels the whole chain will pass through four separate connections. Some HTTP communication options might apply only to the connection with the nearest, non-tunnel neighbor, only to the endpoints of the chain, or to all connections along the chain. Although the diagram is linear, each participant might be engaged in multiple, simultaneous communications. For example, B might be receiving requests from many clients other than A, and/or forwarding requests to servers other than C, at the same time that it is handling A’s

request. Likewise, later requests might be sent through a different path of connections, often based on dynamic configuration for load balancing.

The terms "upstream" and "downstream" are used to describe directional requirements in relation to the message flow: all messages flow from upstream to downstream. The terms "inbound" and "outbound" are used to describe directional requirements in relation to the request route: "inbound" means toward the origin server and "outbound" means toward the user agent.

A "proxy" is a message-forwarding agent that is selected by the client, usually via local configuration rules, to receive requests for some type(s) of absolute URI and attempt to satisfy those requests via translation through the HTTP interface. Some translations are minimal, such as for proxy requests for "http" URIs, whereas other requests might require translation to and from entirely different application-level protocols. Proxies are often used to group an organization’s HTTP requests through a common intermediary for the sake of security, annotation services, or shared caching. Some proxies are designed to apply transformations to selected messages or payloads while they are being forwarded.

A "gateway" (a.k.a. "reverse proxy") is an intermediary that acts as an origin server for the outbound connection but translates received requests and forwards them inbound to another server or servers. Gateways are often used to encapsulate legacy or untrusted information services, to improve server performance through "accelerator" caching, and to enable partitioning or load balancing of HTTP services across multiple machines.

All HTTP requirements applicable to an origin server also apply to the outbound communication of a gateway. A gateway communicates with inbound servers using any protocol that it desires, including private extensions to HTTP that are outside the scope of this specification. However, an HTTP-to-HTTP gateway that wishes to interoperate with third-party HTTP servers ought to conform to user agent requirements on the gateway's inbound connection.

A "tunnel" acts as a blind relay between two connections without changing the messages. Once active, a tunnel is not considered a party to the HTTP communication, though the tunnel might have been initiated by an HTTP request. A tunnel ceases to exist when both ends of the relayed connection are closed. Tunnels are used to extend a virtual connection through an intermediary, such as when Transport Layer Security (TLS) is used to establish confidential communication through a shared firewall proxy.

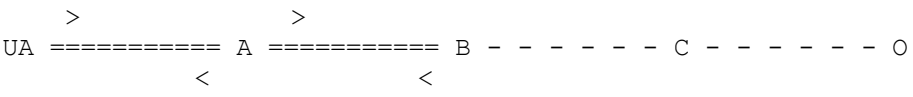
The above categories for intermediary only consider those acting as participants in the HTTP communication. There are also intermediaries that can act on lower layers of the network protocol stack, filtering or redirecting HTTP traffic without the knowledge or permission of message senders. Network intermediaries are indistinguishable (at a protocol level) from a man-in-the-middle attack, often introducing security flaws or interoperability problems due to mistakenly violating HTTP semantics.

HTTP is defined as a stateless protocol, meaning that each request message can be understood in isolation. Many implementations depend on HTTP's stateless design in order to reuse proxied connections or dynamically load balance requests across multiple servers. Hence, a server MUST NOT assume that two requests on the same connection are from the same user agent unless the connection is secured and specific to that agent.

2.4. Caches

A "cache" is a local store of previous response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Any client or server MAY employ a cache, though a cache cannot be used by a server while it is acting as a tunnel.

The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. The following illustrates the resulting chain if B has a cached copy of an earlier response from O (via C) for a request that has not been cached by UA or A.



A response is "cacheable" if a cache is allowed to store a copy of the response message for use in answering subsequent requests. Even when a response is cacheable, there might be additional constraints placed by the client or by the origin server on when that cached response can be used for a particular request. HTTP requirements for cache behavior and cacheable responses are defined in Section 2 of [RFC7234].

2.5. Conformance and Error Handling

This specification targets conformance criteria according to the role of a participant in HTTP communication. Hence, HTTP requirements are placed on senders, recipients, clients, servers, user agents, intermediaries, origin servers, proxies, gateways, or caches, depending on what behavior is being constrained by the requirement. Additional (social) requirements are placed on implementations, resource owners, and protocol element registrations when they apply beyond the scope of a single communication.

The verb "generate" is used instead of "send" where a requirement differentiates between creating a protocol element and merely forwarding a received element downstream.

An implementation is considered conformant if it complies with all of the requirements associated with the roles it partakes in HTTP.

Conformance includes both the syntax and semantics of protocol elements. A sender MUST NOT generate protocol elements that convey a meaning that is known by that sender to be false. A sender MUST NOT generate protocol elements that do not match the grammar defined by the corresponding ABNF rules. Within a given message, a sender MUST NOT generate protocol elements or syntax alternatives that are only allowed to be generated by participants in other roles (i.e., a role that the sender does not have for that message).

When a received protocol element is parsed, the recipient MUST be able to parse any value of reasonable length that is applicable to the recipient's role and that matches the grammar defined by the corresponding ABNF rules. Note, however, that some received protocol elements might not be parsed. For example, an intermediary

forwarding a message might parse a header-field into generic field-name and field-value components, but then forward the header field without further parsing inside the field-value.

A recipient MUST interpret a received protocol element according to the semantics defined for it by this specification, including extensions to this specification, unless the recipient has determined (through experience or configuration) that the sender incorrectly implements what is implied by those semantics. For example, an origin server might disregard the contents of a received Accept-Encoding header field if inspection of the User-Agent header field indicates a specific implementation version that is known to fail on receipt of certain content codings.

Unless noted otherwise, a recipient MAY attempt to recover a usable protocol element from an invalid construct. HTTP does not define specific error handling mechanisms except when they have a direct impact on security, since different applications of the protocol require different error handling strategies. For example, a Web browser might wish to transparently recover from a response where the Location header field doesn't parse according to the ABNF, whereas a systems control client might consider any form of error recovery to be dangerous.

2.6. Protocol Versioning

HTTP uses a "<major>.<minor>" numbering scheme to indicate versions of the protocol. This specification defines version "1.1". The protocol version as a whole indicates the sender's conformance with the set of requirements laid out in that version's corresponding specification of HTTP.

The version of an HTTP message is indicated by an HTTP-version field in the first line of the message. HTTP-version is case-sensitive.

```
HTTP-version = HTTP-name "/" DIGIT "." DIGIT

HTTP-name = %x48.54.54.50 ; "HTTP", case-sensitive
```

The HTTP version number consists of two decimal digits separated by a "." (period or decimal point). The first digit ("major version") indicates the HTTP messaging syntax, whereas the second digit ("minor version") indicates the highest minor version within that major version to which the sender is conformant and able to understand for future communication. The minor version advertises the sender's communication capabilities even when the sender is only using a backwards-compatible subset of the protocol, thereby letting the recipient know that more advanced features can be used in response (by servers) or in future requests (by clients).

The interpretation of a header field does not change between minor versions of the same major HTTP version, though the default behavior of a recipient in the absence of such a field can change. Unless specified otherwise, header fields defined in HTTP/1.1 are defined for all versions of HTTP/1.x. In particular, the Host and Connection header fields ought to be implemented by all HTTP/1.x implementations whether or not they advertise conformance with HTTP/1.1.

New header fields can be introduced without changing the protocol version if their defined semantics allow them to be safely ignored by recipients that do not recognize them.

Intermediaries that process HTTP messages (i.e., all intermediaries other than those acting as tunnels) MUST send their own HTTP-version in forwarded messages. In other words, they are not allowed to blindly forward the first line of an HTTP message without ensuring that the protocol version in that message matches a version to which that intermediary is conformant for both the receiving and sending of messages. Forwarding an HTTP message without rewriting the

HTTP-version might result in communication errors when downstream recipients use the message sender's version to determine what features are safe to use for later communication with that sender.

A client SHOULD send a request version equal to the highest version to which the client is conformant and whose major version is no higher than the highest version supported by the server, if this is known. A client MUST NOT send a version to which it is not conformant.

A client MAY send a lower request version if it is known that the server incorrectly implements the HTTP specification, but only after the client has attempted at least one normal request and determined from the response status code or header fields (e.g., Server) that the server improperly handles higher request versions.

A server SHOULD send a response version equal to the highest version to which the server is conformant that has a major version less than or equal to the one received in the request. A server MUST NOT send a version to which it is not conformant. A server can send a 505 (HTTP Version Not Supported) response if it wishes, for any reason, to refuse service of the client's major protocol version.

A server MAY send an HTTP/1.0 response to a request if it is known or suspected that the client incorrectly implements the HTTP specification and is incapable of correctly processing later version responses, such as when a client fails to parse the version number correctly or when an intermediary is known to blindly forward the HTTP-version even when it doesn't conform to the given minor version of the protocol. Such protocol downgrades SHOULD NOT be performed unless triggered by specific client attributes, such as when one or more of the request header fields (e.g., User-Agent) uniquely match the values sent by a client known to be in error.

The intention of HTTP's versioning design is that the major number will only be incremented if an incompatible message syntax is introduced, and that the minor number will only be incremented when changes made to the protocol have the effect of adding to the message semantics or implying additional capabilities of the sender.

When an HTTP message is received with a major version number that the recipient implements, but a higher minor version number than what the recipient implements, the recipient SHOULD process the message as if it were in the highest minor version within that major version to which the recipient is conformant. A recipient can assume that a message with a higher minor version, when sent to a recipient that has not yet indicated support for that higher version, is sufficiently backwards-compatible to be safely processed by any implementation of the same major version.

2.7. Uniform Resource Identifiers (URI)

Uniform Resource Identifiers (URIs)are used throughout HTTP as the means for identifying resources. URI references are used to target requests, indicate redirects, and define relationships.

The definitions of "URI-reference", "absolute-URI", "relative-part", "scheme", "authority", "port", "host", "path-abempty", "segment", "query", and "fragment" are adopted from the URI generic syntax. An "absolute-path" rule is defined for protocol elements that can contain a non-empty path component. (This rule differs slightly from the path-abempty rule of RFC 3986, which allows for an empty path to be used in references, and path-absolute rule, which does not allow paths that begin with "//".) A "partial-URI" rule is defined for protocol elements that can contain a relative URI but not a fragment component.

```
URI-reference = <URI-reference, see [RFC3986], Section 4.1>
absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
relative-part  = <relative-part, see [RFC3986], Section 4.2>
scheme = <scheme, see [RFC3986], Section 3.1>
authority = <authority, see [RFC3986], Section 3.2>
uri-host  = <host, see [RFC3986], Section 3.2.2>
port  = <port, see [RFC3986], Section 3.2.3>
path-abempty = <path-abempty, see [RFC3986], Section 3.3>
segment  = <segment, see [RFC3986], Section 3.3>
query = <query, see [RFC3986], Section 3.4>
fragment= <fragment, see [RFC3986], Section 3.5>

absolute-path = 1*( "/" segment )
partial-URI= relative-part [ "?" query ]
```

Each protocol element in HTTP that allows a URI reference will indicate in its ABNF production whether the element allows any form of reference (URI-reference), only a URI in absolute form (absolute-URI), only the path and optional query components, or some combination of the above. Unless otherwise indicated, URI references are parsed relative to the effective request URI.

2.7.1. http URI Scheme

The "http" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening for TCP connections on a given port.

```
http-URI = "http:" "//" authority path-abempty [ "?" query ] [ "#" fragment ]
```

The origin server for an "http" URI is identified by the authority component, which includes a host identifier and optional TCP port. The hierarchical path component and optional query component serve as an identifier for a potential target resource within that origin server's name space. The optional fragment component allows for indirect identification of a secondary resource, independent of the URI scheme.

A sender MUST NOT generate an "http" URI with an empty host identifier. A recipient that processes such a URI reference MUST reject it as invalid.

If the host identifier is provided as an IP address, the origin server is the listener (if any) on the indicated TCP port at that IP address. If host is a registered name, the registered name is an indirect identifier for use with a name resolution service, such as DNS, to find an address for that origin server. If the port subcomponent is empty or not given, TCP port 80 (the reserved port for WWW services) is the default.

Note that the presence of a URI with a given authority component does not imply that there is always an HTTP server listening for connections on that host and port. Anyone can mint a URI. What the authority component determines is who has the right to respond authoritatively to requests that target the identified resource. The delegated nature of registered names and IP addresses creates a federated namespace, based on control over the indicated host and port, whether or not an HTTP server is present. See Section 9.1 for security considerations related to establishing authority.

When an "http" URI is used within a context that calls for access to the indicated resource, a client MAY attempt access by resolving the host to an IP address, establishing a TCP connection to that address on the indicated port, and sending an HTTP request message containing the URI's identifying data to the server. If the server responds to that request with a non-interim HTTP response message, then that response is considered an authoritative answer to the client's request.

Although HTTP is independent of the transport protocol, the "http" scheme is specific to TCP-based services because the name delegation process depends on TCP for establishing authority. An HTTP service based on some other underlying connection protocol would presumably be identified using a different URI scheme, just as the "https" scheme (below) is used for resources that require an end-to-end secured connection. Other protocols might also be used to provide access to "http" identified resources -- it is only the authoritative interface that is specific to TCP.

The URI generic syntax for authority also includes a deprecated userinfo subcomponent for including user authentication information in the URI. Some implementations make use of the userinfo component for internal configuration of authentication information, such as within command invocation options, configuration files, or bookmark lists, even though such usage might expose a user identifier or password. A sender MUST NOT generate the userinfo subcomponent (and its "@" delimiter) when an "http" URI reference is generated within a message as a request target or header field value. Before making use of an "http" URI reference received from an untrusted source, a recipient SHOULD parse for userinfo and treat its presence as an error; it is likely being used to obscure the authority for the sake of phishing attacks.

2.7.2. https URI Scheme

The "https" URI scheme is hereby defined for the purpose of minting identifiers according to their association with the hierarchical namespace governed by a potential HTTP origin server listening to a given TCP port for TLS-secured connections.

All of the requirements listed above for the "http" scheme are also requirements for the "https" scheme, except that TCP port 443 is the default if the port subcomponent is empty or not given, and the user agent MUST ensure that its connection to the origin server is secured through the use of strong encryption, end-to-end, prior to sending the first HTTP request.

```
https-URI = "https:" "//" authority path-abempty [ "?" query ] [ "#" fragment ]
```

Note that the "https" URI scheme depends on both TLS and TCP for establishing authority. Resources made available via the "https" scheme have no shared identity with the "http" scheme even if their resource identifiers indicate the same authority (the same host listening to the same TCP port). They are distinct namespaces and are considered to be distinct origin servers. However, an extension to HTTP that is defined to apply to entire host domains, such as the Cookie protocol, can allow information set by one service to impact communication with other services within a matching group of host domains.

2.7.3. http and https URI Normalization and Comparison

Since the "http" and "https" schemes conform to the URI generic syntax, such URIs are normalized and compared according to the algorithm defined in Section 6 of [RFC3986], using the defaults described above for each scheme.

If the port is equal to the default port for a scheme, the normal form is to omit the port subcomponent. When not being used in absolute form as the request target of an OPTIONS request, an empty path component is equivalent to an absolute path of "/", so the normal form is to provide a path of "/" instead. The scheme and host are case-insensitive and normally provided in lowercase; all other components are compared in a case-sensitive manner. Characters other than those in the "reserved" set are equivalent to their percent-encoded octets: the normal form is to not encode them (see Sections 2.1 and 2.2 of [RFC3986]).

For example, the following three URIs are equivalent:

http://example.com:80/~smith/home.html
http://EXAMPLE.com/%7Esmith/home.html
http://EXAMPLE.com:/%7esmith/home.html

3. Message Format

All HTTP/1.1 messages consist of a start-line followed by a sequence of octets in a format similar to the Internet Message Format: zero or more header fields (collectively referred to as the "headers" or the "header section"), an empty line indicating the end of the header section, and an optional message body.

```
HTTP-message =    start-line
                  *( header-field CRLF )
                  CRLF
                  [ message-body ]
```

The normal procedure for parsing an HTTP message is to read the start-line into a structure, read each header field into a hash table by field name until the empty line,

and then use the parsed data to determine if a message body is expected. If a message body has been indicated, then it is read as a stream until an amount of octets equal to the message body length is read or the connection is closed.

A recipient MUST parse an HTTP message as a sequence of octets in an encoding that is a superset of US-ASCII. Parsing an HTTP message as a stream of Unicode characters, without regard for the specific encoding, creates security vulnerabilities due to the varying ways that string processing libraries handle invalid multibyte character sequences that contain the octet LF (%x0A). String-based parsers can only be safely used within protocol elements after the element has been extracted from the message, such as within a header field-value after message parsing has delineated the individual fields.

An HTTP message can be parsed as a stream for incremental processing or forwarding downstream. However, recipients cannot rely on incremental delivery of partial messages, since some implementations will buffer or delay message forwarding for the sake of network efficiency, security checks, or payload transformations.

A sender MUST NOT send whitespace between the start-line and the first header field. A recipient that receives whitespace between the start-line and the first header field MUST either reject the message as invalid or consume each whitespace-preceded line without further processing of it (i.e., ignore the entire line, along with any subsequent lines preceded by whitespace, until a properly formed header field is received or the header section is terminated).

The presence of such whitespace in a request might be an attempt to trick a server into ignoring that field or processing the line after it as a new request, either of which might result in a security vulnerability if other implementations within the request chain interpret the same message differently. Likewise, the presence of such whitespace in a response might be ignored by some clients or cause others to cease parsing.

=> Voir le cahier pour le reste de HTTP message

👉 Transfer-Encoding (RFC 7230, 3.3.1.)

-> lists the transfer coding names corresponding to the sequence of transfer codings that have been (or will be) applied to the payload body in order to form the message body.

Transfer-Encoding = 1#transfer-coding

Transfer-Encoding is primarily intended to accurately delimit a dynamically generated payload and to distinguish payload encodings that are only applied for transport efficiency or security from those that are characteristics of the selected resource.

A recipient MUST be able to parse the chunked transfer coding because it plays a crucial role in framing messages when the payload body size is not known in advance. A sender MUST NOT apply chunked more than once to a message body. If any transfer coding other than chunked is applied to a request payload body, the sender MUST apply chunked as the final transfer coding to ensure that the message is properly framed. If any transfer coding other than chunked is applied to a response payload body, the sender MUST either apply chunked as the final transfer coding or terminate the message by closing the connection.

For example,

Transfer-Encoding: gzip, chunked

indicates that the payload body has been compressed using the gzip coding and then chunked using the chunked coding while forming the message body.

Transfer-Encoding is a property of the message, not of the representation, and any recipient along the request/response chain MAY decode the received

transfer coding(s) or apply additional transfer coding(s) to the message body, assuming that corresponding changes are made to the Transfer-Encoding field-value. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Transfer-Encoding MAY be sent in a response to a HEAD request or in a 304 (Not Modified) response to a GET request, neither of which includes a message body, to indicate that the origin server would have applied a transfer coding to the message body if the request had been an unconditional GET. This indication is not required, however, because any recipient on the response chain (including the origin server) can remove transfer codings when they are not needed.

A server MUST NOT send a Transfer-Encoding header field in any response with a status code of 1xx (Informational) or 204 (No Content). A server MUST NOT send a Transfer-Encoding header field in any 2xx (Successful) response to a CONNECT request.

Transfer-Encoding was added in HTTP/1.1.

A server that receives a request message with a transfer coding it does not understand SHOULD respond with 501 (Not Implemented).

👉 Content-Length (RFC 7230, 3.3.2.)

When a message does not have a Transfer-Encoding header field, a Content-Length header field can provide the anticipated size, as a decimal number of octets, for a potential payload body. For messages that do include a payload body, the Content-Length field-value provides the framing information necessary for determining where the body (and message) ends. For messages that do not include a payload body, the Content-Length indicates the size of the selected representation.

Content-Length = 1*DIGIT

example:

Content-Length: 3495

A sender MUST NOT send a Content-Length header field in any message that contains a Transfer-Encoding header field.

A user agent SHOULD send a Content-Length in a request message when no Transfer-Encoding is sent and the request method defines a meaning for an enclosed payload body. For example, a Content-Length header field is normally sent in a POST request even when the value is 0. A user agent SHOULD NOT send a Content-Length header field when the request message does not contain a payload body and the method semantics do not anticipate such a body.

A server MAY send a Content-Length header field

-> in a response to a HEAD request; a server MUST NOT send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a response if the same request had used the GET method.

-> in a 304 (Not Modified) response to a conditional GET request; a server MUST NOT send Content-Length in such a response unless its field-value equals the decimal number of octets that would have been sent in the payload body of a 200 (OK) response to the same request.

A server MUST NOT send a Content-Length header field

-> in any response with a status code of 1xx (Informational) or 204 (No Content). A server MUST NOT send a Content-Length header field in any 2xx (Successful) response to a CONNECT request.

Aside from the cases defined above, in the absence of Transfer-Encoding, an origin server SHOULD send a Content-Length header field when the payload body size is known prior to sending the complete header section. This will allow downstream recipients to measure transfer progress, know when a received message is complete, and potentially reuse the connection for additional requests.

Any Content-Length field value greater than or equal to zero is valid. Since there is no predefined limit to the length of a payload, a recipient MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows.

If a message is received that has multiple Content-Length header fields with field-values consisting of the same decimal value, or a single Content-Length header field with a field value containing a list of identical decimal values (e.g., "Content-Length: 42, 42"), indicating that duplicate Content-Length header fields have been generated or combined by an upstream message processor, then the recipient MUST either reject the message as invalid or replace the duplicated field-values with a single valid Content-Length field containing that decimal value prior to determining the message body length or forwarding the message.

==========

→ Transfer Coding: (RFC 7230 4.)

Transfer coding names are used to indicate an encoding transformation that has been, can be, or might need to be applied to a payload body in order to ensure "safe transport" through the network. The transfer coding is a property of the message.

All transfer-coding names are case-insensitive and ought to be registered within the HTTP Transfer Coding registry, as defined in Section 8.4. They are used in the TE (Section 4.3) and Transfer-Encoding (Section 3.3.1) header fields

transfer-extension = token *(OWS ";" OWS transfer-parameter)

Parameters are in the form of a name or name=value pair.

transfer-parameter = token BWS "=" BWS (token / quoted-string)

→ Chunked Transfer Coding (RFC 7230 4.1.)

transfer-coding = "chunked" ;

The chunked transfer coding wraps the payload body in order to transfer it as a series of chunks, each with its own size indicator, followed by an OPTIONAL trailer containing header fields. Chunked enables content streams of unknown size to be transferred as a sequence of length-delimited buffers, which enables the sender to retain connection persistence and the recipient to know when it has received the entire message.

*chunked-body = *chunk last-chunk trailer-part CRLF*

chunk = chunk-size [chunk-ext] CRLF chunk-data CRLF

*chunk-size = 1*HEXDIG*

last-chunk = 1("0") [chunk-ext] CRLF*

*chunk-data = 1*OCTET ; a sequence of chunk-size octets*

The chunk-size field is a string of hex digits indicating the size of the chunk-data in octets. The chunked transfer coding is complete when a chunk with a chunk-size of zero is received, possibly followed by a trailer, and finally terminated by an empty line.

A recipient MUST be able to parse and decode the chunked transfer coding.

Chunk Extensions (RFC 7230 4.1.1.)

The chunked encoding allows each chunk to include zero or more chunk extensions, immediately following the chunk-size, for the sake of supplying per-chunk metadata (such as a signature or hash), mid-message control information, or randomization of message body size.

*chunk-ext = *(";" chunk-ext-name ["=" chunk-ext-val])*

chunk-ext-name = token

chunk-ext-val = token / quoted-string

The chunked encoding is specific to each connection and is likely to be removed or recoded by each recipient (including intermediaries) before any higher-level application would have a chance to inspect the extensions. Hence, use of chunk extensions is generally limited

to specialized HTTP services such as "long polling" (where client and server can have shared expectations regarding the use of chunk extensions) or for padding within an end-to-end secured connection.

A recipient MUST ignore unrecognized chunk extensions. A server ought to limit the total length of chunk extensions received in a request to an amount reasonable for the services provided, in the same way that it applies length limitations and timeouts for other parts of a message, and generate an appropriate 4xx (Client Error) response if that amount is exceeded.

Chunked Trailer Part (RFC 7230 4.1.2.)

A trailer allows the sender to include additional fields at the end of a chunked message in order to supply metadata that might be dynamically generated while the message body is sent, such as a message integrity check, digital signature, or post-processing status. The trailer fields are identical to header fields, except they are sent in a chunked trailer instead of the message's header section.

*trailer-part = *(header-field CRLF)*

A sender MUST NOT generate a trailer that contains a field necessary for message framing (e.g., Transfer-Encoding and Content-Length), routing (e.g., Host), request modifiers (e.g., controls and conditionals in Section 5 of [RFC7231]), authentication (e.g., see [RFC7235] and [RFC6265]), response control data (e.g., see Section 7.1 of [RFC7231]), or determining how to process the payload (e.g., Content-Encoding, Content-Type, Content-Range, and Trailer).

When a chunked message containing a non-empty trailer is received, the recipient MAY process the fields (aside from those forbidden above) as if they were appended to the message's header section. A recipient MUST ignore (or consider as an error) any fields that are forbidden to be sent in a trailer, since processing them as if they were present in the header section might bypass external security filters.

Unless the request includes a TE header field indicating "trailers" is acceptable, as described in Section 4.3, a server SHOULD NOT generate trailer fields that it believes are necessary for the user agent to receive. Without a TE containing "trailers", the server ought to assume that the trailer fields might be silently discarded along the path to the user agent. This requirement allows intermediaries to forward a de-chunked message to an HTTP/1.0 recipient without buffering the entire response.

Decoding Chunked (RFC 7230 4.1.3.)

A process for decoding the chunked transfer coding can be represented *in pseudo-code as:*

```
length := 0
read chunk-size, chunk-ext (if any), and CRLF
while (chunk-size > 0)
{
      read chunk-data and CRLF
      append chunk-data to decoded-body
      length := length + chunk-size
      read chunk-size, chunk-ext (if any), and CRLF
}
read trailer field
while (trailer field is not empty)
{
      if (trailer field is allowed to be sent in a trailer)
            append trailer field to existing header fields
      read trailer-field
}
Content-Length := length
Remove "chunked" from Transfer-Encoding
Remove Trailer from existing header fields
```

→ Compression Codings (RFC 7230 4.2.)

Compress Coding (RFC 7230 4.2.1.)

transfer-coding = "compress" ;

The "compress" coding is an adaptive Lempel-Ziv-Welch (LZW) coding [Welch] that is commonly produced by the UNIX file compression program "compress". A recipient SHOULD consider "x-compress" to be equivalent to "compress".

Deflate Coding (RFC 7230 4.2.2.)

transfer-coding = "deflate" ;

The "deflate" coding is a "zlib" data format [RFC1950] containing a "deflate" compressed data stream [RFC1951] that uses a combination of the Lempel-Ziv (LZ77) compression algorithm and Huffman coding.

Note: Some non-conformant implementations send the "deflate" compressed data without the zlib wrapper.

Gzip Coding (RFC 7230 4.2.3.)

transfer-coding = "gzip" ;

The "gzip" coding is an LZ77 coding with a 32-bit Cyclic Redundancy Check (CRC) that is commonly produced by the gzip file compression program [RFC1952]. A recipient SHOULD consider "x-gzip" to be equivalent to "gzip".

→ TE (RFC 7230 4.3.) // ❌ Header field not mandatory to implement

The "TE" header field in a request indicates what transfer codings, besides chunked, the client is willing to accept in response.

→ Trailer (RFC 7230 4.4.) // ❌ Header field not mandatory to implement

When a message includes a message body encoded with the chunked transfer coding and the sender desires to send metadata in the form of trailer fields at the end of the message, the sender SHOULD generate a Trailer header field before the message body to indicate which fields will be present in the trailers. This allows the recipient to prepare for receipt of that metadata before it starts processing the body, which is useful if the message is being streamed and the recipient wishes to confirm an integrity check on the fly.

Trailer = 1#field-name

→ Message Routing (RFC 7230 5.)

HTTP request message routing is determined by each client based on the target resource, the client's proxy configuration, and establishment or reuse of an inbound connection. The corresponding response routing follows the same connection chain back to the client.

5.1. Identifying a Target Resource //probably useless for us (for client)

5.2. Connecting Inbound //probably useless for us (for client)

5.3. Request Target //probably useless for us (for client)

👉 Host (RFC 7230, 5.4.)

The "Host" header field in a request provides the host and port information from the target URI, enabling the origin server to distinguish among resources while servicing requests for multiple host names on a single IP address.

Host = uri-host [":" port] ; Section 2.7.1

A client MUST send a Host header field in all HTTP/1.1 request messages. If the target URI includes an authority component, then a client MUST send a field-value for Host that is identical to that authority component, excluding any userinfo subcomponent and its "@" delimiter (Section 2.7.1). If the authority component is missing or undefined for the target URI, then a client MUST send a Host header field with an empty field-value.

Since the Host field-value is critical information for handling a request, a user agent SHOULD generate Host as the first header field following the request-line.

For example, a GET request to the origin server for <http://www.example.org/pub/WWW/> would begin with:

```
GET /pub/WWW/ HTTP/1.1
Host: www.example.org
```

A client MUST send a Host header field in an HTTP/1.1 request even if the request-target is in the absolute-form, since this allows the Host information to be forwarded through ancient HTTP/1.0 proxies that might not have implemented Host.

When a proxy receives a request with an absolute-form of request-target, the proxy MUST ignore the received Host header field (if any) and instead replace it with the host information of the request-target. A proxy that forwards such a request MUST generate a new Host field-value based on the received request-target rather than forward the received Host field-value.

Since the Host header field acts as an application-level routing mechanism, it is a frequent target for malware seeking to poison a shared cache or redirect a request to an unintended server. An interception proxy is particularly vulnerable if it relies on the Host field-value for redirecting requests to internal servers, or for use as a cache key in a shared cache, without first verifying that the intercepted connection is targeting a valid IP address for that host.

A server MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header field and to any request message that contains more than one Host header field or a Host header field with an invalid field-value.

→ Effective Request URI (RFC 7230 5.5.)

Since the request-target often contains only part of the user agent's target URI, a server reconstructs the intended target as an "effective request URI" to properly service the request. This reconstruction involves both the server's local configuration and information communicated in the request-target, Host header field, and connection context.

For a user agent, the effective request URI is the target URI.

If the request-target is in absolute-form, the effective request URI is the same as the request-target. Otherwise, the effective request URI is constructed as follows:

If the server's configuration (or outbound gateway) provides a fixed URI scheme, that scheme is used for the effective request URI. The scheme is "http".

If the server's configuration (or outbound gateway) provides a fixed URI authority component, that authority is used for the effective request URI. If not, then if the request-target is in authority-form, the effective request URI's authority component is the same as the request-target. If not, then if a Host header field is supplied with a non-empty field-value, the authority component is the same as the Host field-value. Otherwise, the authority component is assigned the default name configured for the server and, if the connection's incoming TCP port number differs from the default port for the effective request URI's scheme, then a colon (":") and the incoming port number (in decimal form) are appended to the authority component.

If the request-target is in authority-form or asterisk-form, the effective request URI's combined path and query component is empty. Otherwise, the combined path and query component is the same as the request-target.

The components of the effective request URI, once determined as above, can be combined into absolute-URI form by concatenating the scheme, "://", authority, and combined path and query component.

Example 1: the following message received over an insecure TCP connection

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.example.org:8080
```

has an effective request URI of

http://www.example.org:8080/pub/WWW/TheProject.html

Once the effective request URI has been constructed, an origin server needs to decide whether or not to provide service for that URI via the connection in which the request was received. For example, the request might have been misdirected, deliberately or accidentally, such that the information within a received request-target or Host header field differs from the host or port upon which the connection has been made. If the connection is from a trusted gateway, that inconsistency might be expected; otherwise, it might indicate an attempt to bypass security filters, trick the server into delivering non-public content, or poison a cache. See Section 9 for security considerations regarding message routing.

→ Associating a Response to a Request (RFC 7230 5.6.)

HTTP does not include a request identifier for associating a given request message with its corresponding one or more response messages. Hence, it relies on the order of response arrival to correspond exactly to the order in which requests are made on the same connection. More than one response message per request only occurs when one or more informational responses (1xx, see Section 6.2 of [RFC7231]) precede a final response to the same request.

A client that has more than one outstanding request on a connection MUST maintain a list of outstanding requests in the order sent and MUST associate each received response message on that connection to the highest ordered request that has not yet received a final (non-1xx) response.

→ Message Forwarding (RFC 7230 5.7.) //maybe useless (about intermediary)

As described in Section 2.3, intermediaries can serve a variety of roles in the processing of HTTP requests and responses. Some intermediaries are used to improve performance or availability. Others are used for access control or to filter content. Since an HTTP stream has characteristics similar to a pipe-and-filter architecture, there are no inherent limits to the extent an intermediary can enhance (or interfere) with either direction of the stream.

An intermediary not acting as a tunnel MUST implement the Connection header field, as specified in Section 6.1, and exclude fields from being forwarded that are only intended for the incoming connection.

An intermediary MUST NOT forward a message to itself unless it is protected from an infinite request loop. In general, an intermediary ought to recognize its own server names, including any aliases, local variations, or literal IP addresses, and respond to such requests directly.

→ Via (RFC 7230 5.7.1.) // ❌ Header field not mandatory to implement

→ Transformations (RFC 7230 5.7.2.) // ❌ probably useless

→ Connection Management (RFC 7230 6.)

HTTP messaging is independent of the underlying transport- or session-layer connection protocol(s). HTTP only presumes a reliable transport with in-order delivery of requests and the corresponding in-order delivery of responses. The mapping of HTTP request and response structures onto the data units of an underlying transport protocol is outside the scope of this specification.

As described in Section 5.2, the specific connection protocols to be used for an HTTP interaction are determined by client configuration and the target URI. For example, the "http" URI scheme

(Section 2.7.1) indicates a default connection of TCP over IP, with a default TCP port of 80, but the client might be configured to use a proxy via some other connection, port, or protocol.

HTTP implementations are expected to engage in connection management, which includes maintaining the state of current connections, establishing a new connection or reusing an existing connection, processing messages received on a connection, detecting connection failures, and closing each connection. Most clients maintain multiple connections in parallel, including more than one connection per server endpoint. Most servers are designed to maintain thousands of concurrent connections, while controlling request queues to enable fair use and detect denial-of-service attacks.

→ Connection (RFC 7230 6.1.) // ❌ Header field not mandatory to implement

→ Establishment (RFC 7230 6.2.)

Each connection applies to only one transport link.

→ Persistence (RFC 7230 6.3.)

HTTP/1.1 defaults to the use of "persistent connections", allowing multiple requests and responses to be carried over a single connection. The "close" connection option is used to signal that a connection will not persist after the current request/response. HTTP implementations SHOULD support persistent connections.

A recipient determines whether a connection is persistent or not based on the most recently received message's protocol version and Connection header field (if any):

o If the "close" connection option is present, the connection will not persist after the current response; else,

o If the received protocol is HTTP/1.1 (or later), the connection will persist after the current response; else,

o If the received protocol is HTTP/1.0, the "keep-alive" connection option is present, the recipient is not a proxy, and the recipient wishes to honor the HTTP/1.0 "keep-alive" mechanism, the connection will persist after the current response; otherwise,

o The connection will close after the current response.

A client MAY send additional requests on a persistent connection until it sends or receives a "close" connection option or receives an HTTP/1.0 response without a "keep-alive" connection option.

In order to remain persistent, all messages on a connection need to have a self-defined message length, as described in Section 3.3. A server MUST read the entire request message body or close the connection after sending its response, since otherwise the remaining data on a persistent connection would be misinterpreted as the next request. Likewise, a client MUST read the entire response message body if it intends to reuse the same connection for a subsequent request.

6.3.1.Retrying Requests

Connections can be closed at any time, with or without intention. Implementations ought to anticipate the need to recover from asynchronous close events.

When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods (Section 4.2.2 of [RFC7231]). A proxy MUST NOT automatically retry non-idempotent requests.

A user agent MUST NOT automatically retry a request with a non- idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

A client SHOULD NOT automatically retry a failed automatic retry.

6.3.2. Pipelining

A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MAY process a sequence of pipelined requests in parallel if they all have safe methods (Section 4.2.1 of [RFC7231]), but it MUST send the corresponding responses in the same order that the requests were received.

A client that pipelines requests SHOULD retry unanswered requests if the connection closes before it receives all of the corresponding responses. When retrying pipelined requests after a failed connection (a connection not explicitly closed by the server in its last complete response), a client MUST NOT pipeline immediately after connection establishment, since the first remaining request in the prior pipeline might have caused an error response that can be lost again if multiple requests are sent on a prematurely closed connection (see the TCP reset problem described in Section 6.6).

Idempotent methods (Section 4.2.2 of [RFC7231]) are significant to pipelining because they can be automatically retried after a connection failure. A user agent SHOULD NOT pipeline requests after a non-idempotent method, until the final response status code for that method has been received, unless the user agent has a means to detect and recover from partial failure conditions involving the pipelined sequence.

An intermediary that receives pipelined requests MAY pipeline those requests when forwarding them inbound, since it can rely on the outbound user agent(s) to determine what requests can be safely pipelined. If the inbound connection fails before receiving a response, the pipelining intermediary MAY attempt to retry a sequence of requests that have yet to receive a response if the requests all have idempotent methods; otherwise, the pipelining intermediary SHOULD forward any received responses and then close the corresponding outbound connection(s) so that the outbound user agent(s) can recover accordingly.

→ Concurrency (RFC 7230 6.4.)

A client ought to limit the number of simultaneous open connections that it maintains to a given server.

Multiple connections are typically used to avoid the "head-of-line blocking" problem, wherein a request that takes significant server-side processing and/or has a large payload blocks subsequent requests on the same connection. However, each connection consumes server resources. Furthermore, using multiple connections can cause undesirable side effects in congested networks.

Note that a server might reject traffic that it deems abusive or characteristic of a denial-of-service attack, such as an excessive number of open connections from a single client.

→ Failures and Timeouts (RFC 7230 6.5.)

Servers will usually have some timeout value beyond which they will no longer maintain an inactive connection. Proxy servers might make this a higher value since it is likely that the client will be making more connections through the same proxy server. The use of persistent connections places no requirements on the length (or existence) of this timeout for either the client or the server.

A client or server that wishes to time out SHOULD issue a graceful close on the connection. Implementations SHOULD constantly monitor open connections for a received closure signal and respond to it as appropriate, since prompt closure of both sides of a connection enables allocated system resources to be reclaimed.

A client, server, or proxy MAY close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

A server SHOULD sustain persistent connections, when possible, and allow the underlying transport's flow-control mechanisms to resolve temporary overloads, rather than terminate connections with the expectation that clients will retry. The latter technique can exacerbate network congestion.

A client sending a message body SHOULD monitor the network connection for an error response while it is transmitting the request. If the client sees a response that indicates the server does not wish to receive the message body and is closing the connection, the client SHOULD immediately cease transmitting the body and close its side of the connection.

→ Tear Down (RFC 7230 6.6.) //probably useless

→ Upgrade (RFC 7230 6.7.) // ❌ Header field not mandatory to implement

→ ABNF List Extension: #rule (RFC 7230 6.6.) //maybe useless ?

→ Security Considerations (RFC 7230 9.)

This section is meant to inform developers, information providers, and users of known security considerations relevant to HTTP message syntax, parsing, and routing. Security considerations about HTTP semantics and payloads are addressed in [RFC7231].

9.1. Establishing Authority

HTTP relies on the notion of an authoritative response: a response that has been determined by (or at the direction of) the authority identified within the target URI to be the most appropriate response for that request given the state of the target resource at the time of response message origination. Providing a response from a non-authoritative source, such as a shared cache, is often useful to improve performance and availability, but only to the extent that the source can be trusted or the distrusted response can be safely used.

Unfortunately, establishing authority can be difficult. For example, phishing is an attack on the user's perception of authority, where that perception can be misled by presenting similar branding in hypertext, possibly aided by userinfo obfuscating the authority component (see Section 2.7.1). User agents can reduce the impact of phishing attacks by enabling users to easily inspect a target URI prior to making an action, by prominently distinguishing (or rejecting) userinfo when present, and by not sending stored credentials and cookies when the referring document is from an unknown or untrusted source.

When a registered name is used in the authority component, the "http" URI scheme (Section 2.7.1) relies on the user's local name resolution service to determine where it can find authoritative responses. This means that any attack on a user's network host table, cached names, or name resolution libraries becomes an avenue for attack on establishing authority. Likewise, the user's choice of server for Domain Name Service (DNS), and the hierarchy of servers from which it obtains resolution results, could impact the authenticity of address mappings; DNS Security Extensions (DNSSEC, [RFC4033]) are one way to improve authenticity.

Furthermore, after an IP address is obtained, establishing authority for an "http" URI is vulnerable to attacks on Internet Protocol routing.

The "https" scheme (Section 2.7.2) is intended to prevent (or at least reveal) many of these potential attacks on establishing authority, provided that the negotiated TLS connection is secured and the client properly verifies that the communicating server's identity matches the target URI's authority component (see [RFC2818]). Correctly implementing such verification can be difficult (see [Georgiev]).

9.2. Risks of Intermediaries

By their very nature, HTTP intermediaries are men-in-the-middle and, thus, represent an opportunity for man-in-the-middle attacks. Compromise of the systems on which the intermediaries run can result in serious security and privacy problems. Intermediaries might have access to security-related information, personal information about individual users and organizations, and proprietary information belonging to users and content providers. A compromised intermediary, or an intermediary implemented or configured without regard to security and privacy considerations, might be used in the commission of a wide range of potential attacks.

Intermediaries that contain a shared cache are especially vulnerable to cache poisoning attacks, as described in Section 8 of [RFC7234].

Implementers need to consider the privacy and security implications of their design and coding decisions, and of the configuration options they provide to operators (especially the default configuration).

Users need to be aware that intermediaries are no more trustworthy than the people who run them; HTTP itself cannot solve this problem.

9.3. Attacks via Protocol Element Length

Because HTTP uses mostly textual, character-delimited fields, parsers are often vulnerable to attacks based on sending very long (or very slow) streams of data, particularly where an implementation is expecting a protocol element with no predefined length.

To promote interoperability, specific recommendations are made for minimum size limits on request-line (Section 3.1.1) and header fields (Section 3.2). These are minimum recommendations, chosen to be supportable even by implementations with limited resources; it is expected that most implementations will choose substantially higher limits.

A server can reject a message that has a request-target that is too long (Section 6.5.12 of [RFC7231]) or a request payload that is too large (Section 6.5.11 of [RFC7231]). Additional status codes related to capacity limits have been defined by extensions to HTTP [RFC6585].

Recipients ought to carefully limit the extent to which they process other protocol elements, including (but not limited to) request methods, response status phrases, header

field-names, numeric values, and body chunks. Failure to limit such processing can result in buffer overflows, arithmetic overflows, or increased vulnerability to denial-of-service attacks.

9.4. Response Splitting

Response splitting (a.k.a, CRLF injection) is a common technique, used in various attacks on Web usage, that exploits the line-based nature of HTTP message framing and the ordered association of requests to responses on persistent connections [Klein]. This technique can be particularly damaging when the requests pass through a shared cache.

Response splitting exploits a vulnerability in servers (usually within an application server) where an attacker can send encoded data within some parameter of the request that is later decoded and echoed within any of the response header fields of the response. If the decoded data is crafted to look like the response has ended and a

subsequent response has begun, the response has been split and the content within the apparent second response is controlled by the attacker. The attacker can then make any other request on the same persistent connection and trick the recipients (including intermediaries) into believing that the second half of the split is an authoritative answer to the second request.

For example, a parameter within the request-target might be read by an application server and reused within a redirect, resulting in the same parameter being echoed in the Location header field of the response. If the parameter is decoded by the application and not properly encoded when placed in the response field, the attacker can send encoded CRLF octets and other content that will make the application's single response look like two or more responses.

A common defense against response splitting is to filter requests for data that looks like encoded CR and LF (e.g., "%0D" and "%0A"). However, that assumes the application server is only performing URI decoding, rather than more obscure data transformations like charset transcoding, XML entity translation, base64 decoding, sprintf reformatting, etc. A more effective mitigation is to prevent anything other than the server's core protocol libraries from sending a CR or LF within the header section, which means restricting the output of header fields to APIs that filter for bad octets and not allowing application servers to write directly to the protocol stream.

9.5. Request Smuggling

Request smuggling ([Linhart]) is a technique that exploits differences in protocol parsing among various recipients to hide additional requests (which might otherwise be blocked or disabled by policy) within an apparently harmless request. Like response splitting, request smuggling can lead to a variety of attacks on HTTP usage.

This specification has introduced new requirements on request parsing, particularly with regard to message framing in

Section 3.3.3, to reduce the effectiveness of request smuggling.

9.6. Message Integrity

HTTP does not define a specific mechanism for ensuring message integrity, instead relying on the error-detection ability of underlying transport protocols and the use of length or chunk-delimited framing to detect completeness. Additional integrity mechanisms, such as hash functions or digital signatures applied to the content, can be selectively added to messages via extensible

metadata header fields. Historically, the lack of a single integrity mechanism has been justified by the informal nature of most HTTP communication. However, the prevalence of HTTP as an information access mechanism has resulted in its increasing use within environments where verification of message integrity is crucial.

User agents are encouraged to implement configurable means for detecting and reporting failures of message integrity such that those means can be enabled within environments for which integrity is necessary. For example, a browser being used to view medical history or drug interaction information needs to indicate to the user when such

information is detected by the protocol to be incomplete, expired, or corrupted during transfer. Such mechanisms might be selectively enabled via user agent extensions or the presence of message integrity metadata in a response. At a minimum, user agents ought to provide some indication that allows a user to distinguish between a complete and incomplete response message (Section 3.4) when such verification is desired.

9.7. Message Confidentiality

HTTP relies on underlying transport protocols to provide message confidentiality when that is desired. HTTP has been specifically designed to be independent of the transport protocol, such that it can be used over many different forms of encrypted connection, with the selection of such transports being identified by the choice of URI scheme or within user agent configuration.

The "https" scheme can be used to identify resources that require a confidential connection, as described in Section 2.7.2.

9.8. Privacy of Server Log Information

A server is in the position to save personal data about a user's requests over time, which might identify their reading patterns or subjects of interest. In particular, log information gathered at an intermediary often contains a history of user agent interaction, across a multitude of sites, that can be traced to individual users.

HTTP log information is confidential in nature; its handling is often constrained by laws and regulations. Log information needs to be securely stored and appropriate guidelines followed for its analysis. Anonymization of personal information within individual entries helps, but it is generally not sufficient to prevent real log traces from being re-identified based on correlation with other access characteristics. As such, access traces that are keyed to a specific client are unsafe to publish even if the key is pseudonymous.

To minimize the risk of theft or accidental publication, log information ought to be purged of personally identifiable information, including user identifiers, IP addresses, and user-provided query parameters, as soon as that information is no longer necessary to support operational needs for security, auditing, or fraud control.

==========

RFC 7231

👉 **HEADER**

→ **Name of a part**

==========

1. Introduction

Each Hypertext Transfer Protocol (HTTP) message is either a request or a response. A server listens on a connection for a request, parses each message received, interprets the message semantics in relation to the identified request target, and responds to that request with one or more response messages. A client constructs request messages to communicate specific intentions, examines received responses to see if the intentions were carried out, and determines how to interpret the results. This document defines HTTP/1.1 request and response semantics in terms of the architecture.

HTTP provides a uniform interface for interacting with a resource, regardless of its type, nature, or implementation, via the manipulation and transfer of representations.

HTTP semantics include the intentions defined by each request method, extensions to those semantics that might be described in request header fields, the meaning of status codes to indicate a machine-readable response, and the meaning of other control data and resource metadata that might be given in response header fields.

This document also defines representation metadata that describe how a payload is intended to be interpreted by a recipient, the request header fields that might influence

content selection, and the various selection algorithms that are collectively referred to as "content negotiation".

2. Resources

The target of an HTTP request is called a "resource". HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI).

When a client constructs an HTTP/1.1 request message, it sends the target URI in one of various forms. When a request is received, the server reconstructs an effective request URI for the target resource.

One design goal of HTTP is to separate resource identification from request semantics, which is made possible by vesting the request semantics in the request method and a few request-modifying header fields. If there is a conflict between the method semantics and any semantic implied by the URI itself, the method semantics take precedence.

3. Representations

Considering that a resource could be anything, and that the uniform interface provided by HTTP is similar to a window through which one can observe and act upon such a thing only through the communication of messages to some independent actor on the other side, an abstraction is needed to represent ("take the place of") the current or desired state of that thing in our communications. That abstraction is called a representation [REST].

For the purposes of HTTP, a "representation" is information that is intended to reflect a past, current, or desired state of a given resource, in a format that can be readily communicated via the protocol, and that consists of a set of representation metadata and a potentially unbounded stream of representation data.

An origin server might be provided with, or be capable of generating, multiple representations that are each intended to reflect the current state of a target resource. In such cases, some algorithm is used by the origin server to select one of those representations as most applicable to a given request, usually based on content negotiation. This "selected representation" is used to provide the data and metadata for evaluating conditional requests and constructing the payload for 200 (OK) and 304 (Not Modified) responses to GET.

3.1. Representation Metadata

Representation header fields provide metadata about the representation. When a message includes a payload body, the representation header fields describe how to interpret the representation data enclosed in the payload body. In a response to a HEAD request, the representation header fields describe the representation data that would have been enclosed in the payload body if the same request had been a GET.

The following header fields convey representation metadata:

Content-Type, Content-Encoding, Content-Language, Content-Location

3.1.1. Processing Representation Data

3.1.1.1. Media Type

HTTP uses Internet media types in the Content-Type and Accept header fields in order to provide open and extensible data typing and type negotiation. Media types define both a data format and various processing models: how to process that data in accordance with each context in which it is received.

```
media-type = type "/" subtype *( OWS ";" OWS parameter )
type = token
subtype = token
```

The type/subtype MAY be followed by parameters in the form of name=value pairs.

```
parameter = token "=" ( token / quoted-string )
```

The type, subtype, and parameter name tokens are case-insensitive. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name. The presence or absence of a parameter might be significant to the processing of a media-type, depending on its definition within the media type registry.

A parameter value that matches the token production can be transmitted either as a token or within a quoted-string. The quoted and unquoted values are equivalent. For example, the following examples are all equivalent, but the first is preferred for consistency:

```
text/html;charset=utf-8
text/html;charset=UTF-8
Text/HTML;Charset="utf-8"
text/html; charset="utf-8"
```

Internet media types ought to be registered with IANA according to the procedures defined in [BCP13].

Note: Unlike some similar constructs in other header fields, media type parameters do not allow whitespace (even "bad" whitespace) around the "=" character.

3.1.1.2. Charset

HTTP uses charset names to indicate or negotiate the character encoding scheme of a textual representation. A charset is identified by a case-insensitive token.

```
    charset = token
```

Charset names ought to be registered in the IANA "Character Sets" registry (<http://www.iana.org/assignments/character-sets>).

3.1.1.3. Canonicalization and Text Defaults

Internet media types are registered with a canonical form in order to be interoperable among systems with varying native encoding formats. Representations selected or transferred via HTTP ought to be in canonical form, for many of the same reasons described by the Multipurpose Internet Mail Extensions (MIME) [RFC2045].

MIME's canonical form requires that media subtypes of the "text" type use CRLF as the text line break. HTTP allows the transfer of text media with plain CR or LF alone representing a line break, when such line breaks are consistent for an entire representation. An HTTP sender MAY generate, and a recipient MUST be able to parse, line breaks in text media that consist of CRLF, bare CR, or bare LF. In addition, text media in HTTP is not limited to charsets that use octets 13 and 10 for CR and LF, respectively. This flexibility regarding line breaks applies only to text within a representation that has been assigned a "text" media type; it does not apply to "multipart" types or HTTP elements outside the payload body (e.g., header fields).

If a representation is encoded with a content-coding, the underlying data ought to be in a form defined above prior to being encoded.

3.1.1.4. Multipart Types

MIME provides for a number of "multipart" types -- encapsulations of one or more representations within a single message body. All multipart types share a common syntax, as defined in Section 5.1.1 of [RFC2046], and include a boundary parameter as part of the media type value. The message body is itself a protocol element; a sender MUST generate only CRLF to represent line breaks between body parts.

HTTP message framing does not use the multipart boundary as an indicator of message body length, though it might be used by implementations that generate or process the payload. For example, the "multipart/form-data" type is often used for carrying form data in a request, as described in [RFC2388], and the "multipart/ byteranges" type is defined by this specification for use in some 206 (Partial Content) responses [RFC7233].

👉 Content-Type (3.1.1.5.)

The "Content-Type" header field indicates the media type of the associated representation: either the representation enclosed in the message payload or the selected representation, as determined by the message semantics. The indicated media type defines both the data format and how that data is intended to be processed by a recipient, within the scope of the received message semantics, after any content codings indicated by Content-Encoding are decoded.

Content-Type = media-type

Media types are defined in Section 3.1.1.1. An example of the field is

Content-Type: text/html; charset=ISO-8859-4

A sender that generates a message containing a payload body SHOULD generate a Content-Type header field in that message unless the intended media type of the enclosed representation is unknown to the sender. If a Content-Type header field is not present, the recipient MAY either assume a media type of "application/octet-stream" ([RFC2046], Section 4.5.1) or examine the data to determine its type.

In practice, resource owners do not always properly configure their origin server to provide the correct Content-Type for a given representation, with the result that some clients will examine a payload's content and override the specified type. Clients that do so risk drawing incorrect conclusions, which might expose additional security risks (e.g., "privilege escalation"). Furthermore, it is impossible to determine the sender's intent by examining the data format: many data formats match multiple media types that differ only in processing semantics. Implementers are encouraged to provide a means of disabling such "content sniffing" when it is used.

3.1.2. Encoding for Compression or Integrity

3.1.2.1. Content Codings

Content coding values indicate an encoding transformation that has been or can be applied to a representation. Content codings are primarily used to allow a representation to be compressed or otherwise usefully transformed without losing the identity of its underlying media type and without loss of information. Frequently, the representation is stored in coded form, transmitted directly, and only decoded by the final recipient.

content-coding = token

All content-coding values are case-insensitive and ought to be registered within the "HTTP Content Coding Registry", as defined in Section 8.4. They are used in the Accept-Encoding (Section 5.3.4) and Content-Encoding (Section 3.1.2.2) header fields.

The following content-coding values are defined by this specification:

compress (and x-compress): See Section 4.2.1 of [RFC7230].

deflate: See Section 4.2.2 of [RFC7230].

gzip (and x-gzip): See Section 4.2.3 of [RFC7230].

3.1.2.2. Content-Encoding // ❌ Header not required

[...]

3.1.3. Audience Language

3.1.3.1. Language Tags

A language tag, identifies a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings. Computer languages are explicitly excluded.

HTTP uses language tags within the Accept-Language and Content-Language header fields. Accept-Language uses the broader language-range production, whereas Content-Language uses the language-tag production defined below.

language-tag = <Language-Tag, see [RFC5646], Section 2.1>

A language tag is a sequence of one or more case-insensitive subtags, each separated by a hyphen character ("-", %x2D). In most cases, a language tag consists of a primary language subtag that identifies a broad family of related languages (e.g., "en" = English), which is optionally followed by a series of subtags that refine or narrow that language's range (e.g., "en-CA" = the variety of English as communicated in Canada). Whitespace is not allowed within a language tag. Example tags include:

fr, en-US, es-419, az-Arab, x-pig-latin, man-Nkoo-GN See [RFC5646] for further information.

👉 **Content-Language (3.1.3.2.)**

The "Content-Language" header field describes the natural language(s) of the intended audience for the representation. Note that this might not be equivalent to all the languages used within the representation.

Content-Language = 1#language-tag

The primary purpose of Content-Language is to allow a user to identify and differentiate representations according to the users' own preferred language.

Thus, if the content is intended only for a Danish-literate audience, the appropriate field is

Content-Language: da

If no Content-Language is specified, the default is that the content is intended for all language audiences. This might mean that the sender does not consider it to be specific to any natural language, or that the sender does not know for which language it is intended.

Multiple languages MAY be listed for content that is intended for multiple audiences. For example, a rendition of the "Treaty of Waitangi", presented simultaneously in the original Maori and English versions, would call for

Content-Language: mi, en

However, just because multiple languages are present within a representation does not mean that it is intended for multiple linguistic audiences. An example would be a beginner's language primer, such as "A First Lesson in Latin", which is clearly intended to be used by an English-literate audience. In this case, the Content-Language would properly only include "en".

Content-Language MAY be applied to any media type -- it is not limited to textual documents.

3.1.4. Identification

3.1.4.1. Identifying a Representation

When a complete or partial representation is transferred in a message payload, it is often desirable for the sender to supply, or the recipient to determine, an identifier for a resource corresponding to that representation.

For a request message:

o If the request has a Content-Location header field, then the sender asserts that the payload is a representation of the resource identified by the Content-Location field-value. However, such an assertion cannot be trusted unless it can be verified by other means (not defined by this specification). The information might still be useful for revision history links.

o Otherwise, the payload is unidentified.

For a response message, the following rules are applied in order until a match is found:

1. If the request method is GET or HEAD and the response status code is 200 (OK), 204 (No Content), 206 (Partial Content), or 304 (Not Modified), the payload is a representation of the resource identified by the effective request URI.
2. If the request method is GET or HEAD and the response status code is 203 (Non-Authoritative Information), the payload is a potentially modified or enhanced representation of the target resource as provided by an intermediary.
3. If the response has a Content-Location header field and its field-value is a reference to the same URI as the effective request URI, the payload is a representation of the resource identified by the effective request URI.
4. If the response has a Content-Location header field and its field-value is a reference to a URI different from the effective request URI, then the sender asserts that the payload is a representation of the resource identified by the Content-Location field-value. However, such an assertion cannot be trusted unless it can be verified by other means (not defined by this specification).
5. Otherwise, the payload is unidentified.

3.1.4.2. Content-Location

The "Content-Location" header field references a URI that can be used as an identifier for a specific resource corresponding to the representation in this message's payload. In other words, if one were to perform a GET request on this URI at the time of this message's generation, then a 200 (OK) response would contain the same representation that is enclosed as payload in this message.

Content-Location = absolute-URI / partial-URI

The Content-Location value is not a replacement for the effective Request URI. It is representation metadata.

If Content-Location is included in a 2xx (Successful) response message and its value refers (after conversion to absolute form) to a URI that is the same as the effective request URI, then the recipient MAY consider the payload to be a current representation of that resource at the time indicated by the message origination date. For a GET or HEAD request, this is the same as the default semantics when no Content-Location is provided by the server. For a state-changing request like PUT or POST, it implies that the server's response contains the new representation of that resource, thereby distinguishing it from representations that might only report about the action (e.g., "It worked!"). This allows authoring applications to update their local copies without the need for a subsequent GET request.

If Content-Location is included in a 2xx (Successful) response message and its field-value refers to a URI that differs from the effective request URI, then the origin server claims that the URI is an identifier for a different resource corresponding to the enclosed representation. Such a claim can only be trusted if both identifiers share the same resource owner, which cannot be programmatically determined via HTTP.

o For a response to a GET or HEAD request, this is an indication that the effective request URI refers to a resource that is subject to content negotiation and the Content-Location field-value is a more specific identifier for the selected representation.

o For a 201 (Created) response to a state-changing method, a Content-Location field-value that is identical to the Location field-value indicates that this payload is a current representation of the newly created resource.

o Otherwise, such a Content-Location indicates that this payload is a representation reporting on the requested action's status and that the same report is available (for future access with GET) at the given URI. For example, a purchase transaction made via a POST request might include a receipt document as the payload of the 200 (OK) response; the Content-Location field-value provides an identifier for retrieving a copy of that same receipt in the future.

A user agent that sends Content-Location in a request message is stating that its value refers to where the user agent originally obtained the content of the enclosed representation (prior to any modifications made by that user agent). In other words, the user agent is providing a back link to the source of the original representation.

An origin server that receives a Content-Location field in a request message MUST treat the information as transitory request context rather than as metadata to be saved verbatim as part of the representation. An origin server MAY use that context to guide in processing the request or to save it for other uses, such as within source links or versioning metadata. However, an origin server MUST NOT use such context information to alter the request semantics.

For example, if a client makes a PUT request on a negotiated resource and the origin server accepts that PUT (without redirection), then the new state of that resource is expected to be consistent with the one representation supplied in that PUT; the Content-Location cannot be used as a form of reverse content selection identifier to update only one of the negotiated representations. If the user agent had wanted the latter semantics, it would have applied the PUT directly to the Content-Location URI.

3.2. Representation Data

The representation data associated with an HTTP message is either provided as the payload body of the message or referred to by the message semantics and the effective request URI. The representation data is in a format and encoding defined by the representation metadata header fields.

The data type of the representation data is determined via the header fields Content-Type and Content-Encoding. These define a two-layer, ordered encoding model:

```
representation-data := Content-Encoding( Content-Type( bits ) )
```

3.3. Payload Semantics

Some HTTP messages transfer a complete or partial representation as the message "payload". In some cases, a payload might contain only the associated representation's header fields (e.g., responses to HEAD) or only some part(s) of the representation data (e.g., the 206 (Partial Content) status code).

The purpose of a payload in a request is defined by the method semantics. For example, a representation in the payload of a PUT request represents the desired state of the target resource if the request is successfully applied, whereas a representation in the payload of a POST request represents information to be processed by the target resource.

In a response, the payload's purpose is defined by both the request method and the response status code. For example, the payload of a 200 (OK) response to GET represents the current state of the target resource, as observed at the time of the message origination date, whereas the payload of the same status code in a response to POST might represent either the processing result or the new state of the target resource after applying the processing. Response messages with an error status code usually contain a payload that represents the error condition, such that it describes the error state and what next steps are suggested for resolving it.

Header fields that specifically describe the payload, rather than the associated representation, are referred to as "payload header fields". Payload header fields are defined in other parts of this specification, due to their impact on message parsing.

Content-Length, Content-Range, Trailer, Transfer-Encoding => RFC 7230

3.4. Content Negotiation

When responses convey payload information, whether indicating a success or an error, the origin server often has different ways of representing that information; for example, in different formats, languages, or encodings. Likewise, different users or user agents might have differing capabilities, characteristics, or preferences that could influence which representation, among those available, would be best to deliver. For this reason, HTTP provides mechanisms for content negotiation.

This specification defines two patterns of content negotiation that can be made visible within the protocol: "proactive", where the server selects the representation based upon the user agent's stated preferences, and "reactive" negotiation, where the server provides a list of representations for the user agent to choose from. Other patterns of content negotiation include "conditional content", where the representation consists of

multiple parts that are selectively rendered based on user agent parameters, "active content", where the representation contains a script that makes additional (more specific) requests based on the user agent characteristics, and "Transparent Content Negotiation" ([RFC2295]), where content selection is performed by an intermediary. These patterns are not mutually exclusive, and each has trade-offs in applicability and practicality.

Note that, in all cases, HTTP is not aware of the resource semantics. The consistency with which an origin server responds to requests, over time and over the varying dimensions of content negotiation, and thus the "sameness" of a resource's observed representations over time, is determined entirely by whatever entity or algorithm selects or generates those responses. HTTP pays no attention to the man behind the curtain.

3.4.1. Proactive Negotiation

When content negotiation preferences are sent by the user agent in a request to encourage an algorithm located at the server to select the preferred representation, it is called proactive negotiation (a.k.a., server-driven negotiation). Selection is based on the available representations for a response (the dimensions over which it might vary, such as language, content-coding, etc.) compared to various information supplied in the request, including both the explicit negotiation fields of Section 5.3 and implicit characteristics, such as the client's network address or parts of the User-Agent field.

Proactive negotiation is advantageous when the algorithm for selecting from among the available representations is difficult to describe to a user agent, or when the server desires to send its "best guess" to the user agent along with the first response (hoping to avoid the round trip delay of a subsequent request if the "best guess" is good enough for the user). In order to improve the server's guess, a user agent MAY send request header fields that describe its preferences.

Proactive negotiation has serious disadvantages:

o It is impossible for the server to accurately determine what might be "best" for any given user, since that would require complete knowledge of both the capabilities of the user agent and the intended use for the response (e.g., does the user want to view it on screen or print it on paper?);

o Having the user agent describe its capabilities in every request can be both very inefficient (given that only a small percentage of responses have multiple representations) and a potential risk to the user's privacy;

o It complicates the implementation of an origin server and the algorithms for generating responses to a request; and,

o It limits the reusability of responses for shared caching.

A user agent cannot rely on proactive negotiation preferences being consistently honored, since the origin server might not implement proactive negotiation for the requested resource or might decide that sending a response that doesn't conform to the user agent's preferences is better than sending a 406 (Not Acceptable) response.

A Vary header field is often sent in a response subject to proactive negotiation to indicate what parts of the request information were used in the selection algorithm.

3.4.2. Reactive Negotiation

With reactive negotiation (a.k.a., agent-driven negotiation), selection of the best response representation (regardless of the status code) is performed by the user agent after receiving an initial response from the origin server that contains a list of resources for alternative representations. If the user agent is not satisfied by the initial response representation, it can perform a GET request on one or more of the alternative resources, selected based on metadata included in the list, to obtain a different form of representation for that response. Selection of alternatives might be performed automatically by the user agent or manually by the user selecting from a generated (possibly hypertext) menu.

Note that the above refers to representations of the response, in general, not representations of the resource. The alternative representations are only considered representations of the target resource if the response in which those alternatives are provided has the semantics of being a representation of the target resource (e.g., a 200 (OK) response to a GET request) or has the semantics of providing links to alternative representations for the target resource (e.g., a 300 (Multiple Choices) response to a GET request).

A server might choose not to send an initial representation, other than the list of alternatives, and thereby indicate that reactive negotiation by the user agent is preferred. For example, the alternatives listed in responses with the 300 (Multiple Choices) and 406 (Not Acceptable) status codes include information about the available representations so that the user or user agent can react by making a selection.

Reactive negotiation is advantageous when the response would vary over commonly used dimensions (such as type, language, or encoding), when the origin server is unable to determine a user agent's capabilities from examining the request, and generally when public caches are used to distribute server load and reduce network usage.

Reactive negotiation suffers from the disadvantages of transmitting a list of alternatives to the user agent, which degrades user-perceived latency if transmitted in the header section, and needing a second request to obtain an alternate representation. Furthermore, this specification does not define a mechanism for supporting automatic selection, though it does not prevent such a mechanism from being developed as an extension.

4. Request Methods

4.1. Overview

The request method token is the primary source of request semantics; it indicates the purpose for which the client has made this request and what is expected by the client as a successful result.

The request method's semantics might be further specialized by the semantics of some header fields when present in a request (Section 5) if those additional semantics do not conflict with the method. For example, a client can send conditional request header fields (Section 5.2) to make the requested action conditional on the current state of the target resource ([RFC7232]).

```
    method = token
```

HTTP was originally designed to be usable as an interface to distributed object systems. The request method was envisioned as applying semantics to a target resource in much the same way as invoking a defined method on an identified object would apply semantics. The method token is case-sensitive because it might be used as a gateway to object-based systems with case-sensitive method names.

Unlike distributed objects, the standardized request methods in HTTP are not resource-specific, since uniform interfaces provide for better visibility and reuse in network-based systems [REST]. Once defined, a standardized method ought to have the same semantics when applied to any resource, though each resource determines for itself whether those semantics are implemented or allowed.

This specification defines a number of standardized methods that are commonly used in HTTP, as outlined by the following table. By convention, standardized methods are defined in all-uppercase US-ASCII letters.

Method	Description	Sec.
GET	Transfer a current representation of the target resource.	4.3.1
HEAD	Same as GET, but only transfer the status line and header section.	4.3.2
POST	Perform resource-specific processing on the request payload.	4.3.3
PUT	Replace all current representations of the target resource with the request payload.	4.3.4
DELETE	Remove all current representations of the target resource.	4.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	4.3.6
OPTIONS	Describe the communication options for the target resource.	4.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	4.3.8

All general-purpose servers MUST support the methods GET and HEAD. All other methods are OPTIONAL.

Additional methods, outside the scope of this specification, have been standardized for use in HTTP. All such methods ought to be registered within the "Hypertext Transfer Protocol (HTTP) Method Registry" maintained by IANA, as defined in Section 8.1.

The set of methods allowed by a target resource can be listed in an Allow header field (Section 7.4.1). However, the set of allowed methods can change dynamically. When a request method is received that is unrecognized or not implemented by an origin server, the origin server SHOULD respond with the 501 (Not Implemented) status code. When a request method is received that is known by an origin server but not allowed for the target resource, the origin server SHOULD respond with the 405 (Method Not Allowed) status code.

4.2. Common Method Properties

4.2.1. Safe Methods

Request methods are considered "safe" if their defined semantics are essentially read-only; i.e., the client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource. Likewise, reasonable use of a safe method is not expected to cause any harm, loss of property, or unusual burden on the origin server.

This definition of safe methods does not prevent an implementation from including behavior that is potentially harmful, that is not entirely read-only, or that causes side effects while invoking a safe method. What is important, however, is that the client did not request that additional behavior and cannot be held accountable for it. For example, most servers append request information to access log files at the completion of every response, regardless of the method, and that is considered safe even though the log storage might become full and crash the server. Likewise, a safe request initiated by selecting an advertisement on the Web will often have the side effect of charging an advertising account.

Of the request methods defined by this specification, the GET, HEAD, OPTIONS, and TRACE methods are defined to be safe.

The purpose of distinguishing between safe and unsafe methods is to allow automated retrieval processes (spiders) and cache performance optimization (pre-fetching) to work without fear of causing harm. In addition, it allows a user agent to apply appropriate constraints on the automated use of unsafe methods when processing potentially untrusted content.

A user agent SHOULD distinguish between safe and unsafe methods when presenting potential actions to a user, such that the user can be made aware of an unsafe action before it is requested.

When a resource is constructed such that parameters within the effective request URI have the effect of selecting an action, it is the resource owner's responsibility to ensure that the action is consistent with the request method semantics. For example, it is common for Web-based content editing software to use actions within query parameters, such as "page?do=delete". If the purpose of such a resource is to perform an unsafe action, then the resource owner MUST disable or disallow that action when it is accessed using a safe request method. Failure to do so will result in unfortunate side effects when automated processes perform a GET on every URI reference for the sake of link maintenance, pre-fetching, building a search index, etc.

4.2.2. Idempotent Methods

A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. Of the request methods defined by this specification, PUT, DELETE, and safe request methods are idempotent.

Like the definition of safe, the idempotent property only applies to what has been requested by the user; a server is free to log each request separately, retain a revision control history, or implement other non-idempotent side effects for each idempotent request.

Idempotent methods are distinguished because the request can be repeated automatically if a communication failure occurs before the client is able to read the server's response. For example, if a client sends a PUT request and the underlying connection is closed before any response is received, then the client can establish a new connection and retry the idempotent request. It knows that repeating the request will have the same intended effect, even if the original request succeeded, though the response might differ.

4.2.3. Cacheable Methods

Request methods can be defined as "cacheable" to indicate that responses to them are allowed to be stored for future reuse; for specific requirements see [RFC7234]. In general, safe methods that do not depend on a current or authoritative response are defined as cacheable; this specification defines GET, HEAD, and POST as cacheable, although the overwhelming majority of cache implementations only support GET and HEAD.

4.3. Method Definitions

4.3.1. GET

The GET method requests transfer of a current selected representation for the target resource. GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Hence, when people speak of retrieving some identifiable information via HTTP, they are generally referring to making a GET request.

It is tempting to think of resource identifiers as remote file system pathnames and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented. However, there are no such limitations in practice. The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation rather than transfer the files directly. Regardless, only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a response to GET.

A client can alter the semantics of GET to be a "range request", requesting transfer of only some part(s) of the selected representation, by sending a Range header field in the request ([RFC7233]).

A payload within a GET request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request.

The response to a GET request is cacheable; a cache MAY use it to satisfy subsequent GET and HEAD requests unless otherwise indicated by the Cache-Control header field (Section 5.2 of [RFC7234]).

4.3.2. HEAD

The HEAD method is identical to GET except that the server MUST NOT send a message body in the response (i.e., the response terminates at the end of the header section). The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request had been a GET, except that the payload header fields (Section 3.3) MAY be omitted. This method can be used for obtaining metadata about the selected representation without transferring the representation data and is often used for testing hypertext links for validity, accessibility, and recent modification.

A payload within a HEAD request message has no defined semantics; sending a payload body on a HEAD request might cause some existing implementations to reject the request.

The response to a HEAD request is cacheable; a cache MAY use it to satisfy subsequent HEAD requests unless otherwise indicated by the Cache-Control header field (Section 5.2 of [RFC7234]). A HEAD response might also have an effect on previously cached responses to GET; see Section 4.3.5 of [RFC7234].

4.3.3. POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics. For example, POST is used for the following functions (among others):

o Providing a block of data, such as the fields entered into an HTML form, to a data-handling process;

o Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;

o Creating a new resource that has yet to be identified by the origin server; and

o Appending data to a resource's existing representation(s).

An origin server indicates response semantics by choosing an appropriate status code depending on the result of processing the POST request; almost all of the status codes defined by this specification might be received in a response to POST (the exceptions being 206 (Partial Content), 304 (Not Modified), and 416 (Range Not Satisfiable)).

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server SHOULD send a 201 (Created) response containing a Location header field that provides an identifier for the primary resource created (Section 7.1.2) and a representation that describes the status of the request while referring to the new resource(s).

Responses to POST requests are only cacheable when they include explicit freshness information (see Section 4.2.1 of [RFC7234]). However, POST caching is not widely implemented. For cases where an origin server wishes the client to be able to cache the result of a POST in a way that can be reused by a later GET, the origin server MAY send a 200 (OK) response containing the result and a Content-Location header field that has the same value as the POST's effective request URI (Section 3.1.4.2).

If the result of processing a POST would be equivalent to a representation of an existing resource, an origin server MAY redirect the user agent to that resource by sending a 303 (See Other) response with the existing resource's identifier in the Location field. This has the benefits of providing the user agent a resource identifier and transferring the

representation via a method more amenable to shared caching, though at the cost of an extra request if the user agent does not already have the representation cached.

4.3.4. PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a 200 (OK) response. However, there is no guarantee that such a state change will be observable, since the target resource might be acted upon by other user agents in parallel, or might be subject to dynamic processing by the origin server, before any subsequent GET is received. A successful response only implies that the user agent's intent was achieved at the time of its processing by the origin server.

If the target resource does not have a current representation and the PUT successfully creates one, then the origin server MUST inform the user agent by sending a 201 (Created) response. If the target resource does have a current representation and that representation is successfully modified in accordance with the state of the enclosed representation, then the origin server MUST send either a 200 (OK) or a 204 (No Content) response to indicate successful completion of the request.

An origin server SHOULD ignore unrecognized header fields received in a PUT request (i.e., do not save them as part of the resource state).

An origin server SHOULD verify that the PUT representation is consistent with any constraints the server has for the target resource that cannot or will not be changed by the PUT. This is particularly important when the origin server uses internal configuration information related to the URI in order to set the values for representation metadata on GET responses. When a PUT representation is inconsistent with the target resource, the origin server SHOULD either make them consistent, by transforming the representation or changing the resource configuration, or respond with an appropriate error message containing sufficient information to explain why the representation is unsuitable. The 409 (Conflict) or 415 (Unsupported Media Type) status codes are suggested, with the latter being specific to constraints on Content-Type values.

For example, if the target resource is configured to always have a Content-Type of "text/html" and the representation being PUT has a Content-Type of "image/jpeg", the origin server ought to do one of:

- reconfigure the target resource to reflect the new media type;
- transform the PUT representation to a format consistent with that of the resource before saving it as the new resource state; or,
- reject the request with a 415 (Unsupported Media Type) response indicating that the target resource is limited to "text/html", perhaps including a link to a different resource that would be a suitable target for the new representation.

HTTP does not define exactly how a PUT method affects the state of an origin server beyond what can be expressed by the intent of the user agent request and the semantics of the origin server response. It does not define what a resource might be, in any sense of that word, beyond the interface provided via HTTP. It does not define how resource state is "stored", nor how such storage might change as a result of a change in resource state, nor how the origin server translates resource state into representations. Generally speaking, all implementation details behind the resource interface are intentionally hidden by the server.

An origin server MUST NOT send a validator header field (Section 7.2), such as an ETag or Last-Modified field, in a successful response to PUT unless the request's representation data was saved without any transformation applied to the body (i.e., the resource's new representation data is identical to the representation data received in the PUT request) and the validator field value reflects the new representation. This requirement allows a user agent to know when the representation body it has in memory remains current as a result of the PUT, thus not in need of being retrieved again from the origin server, and that the new validator(s) received in the response can be used for future conditional requests in order to prevent accidental overwrites (Section 5.2).

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

Proper interpretation of a PUT request presumes that the user agent knows which target resource is desired. A service that selects a proper URI on behalf of the client, after receiving a state-changing request, SHOULD be implemented using the POST method rather than PUT. If the origin server will not make the requested PUT state change to the target resource and instead wishes to have it applied to a different resource, such as when the resource has been moved to a different URI, then the origin server MUST send an appropriate 3xx (Redirection) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

A PUT request applied to the target resource can have side effects on other resources. For example, an article might have a URI for identifying "the current version" (a resource) that is separate from the URIs identifying each particular version (different resources

that at one point shared the same state as the current version resource). A successful PUT request on "the current version" URI might therefore create a new version resource in addition to changing the state of the target resource, and might also cause links to be added between the related resources.

An origin server that allows PUT on a given target resource MUST send a 400 (Bad Request) response to a PUT request that contains a Content-Range header field (Section 4.2 of [RFC7233]), since the payload is likely to be partial content that has been mistakenly PUT as a full representation. Partial content updates are possible by targeting a separately identified resource with state that overlaps a portion of the larger resource, or by using a different method that has been specifically defined for partial updates (for example, the PATCH method defined in [RFC5789]).

Responses to the PUT method are not cacheable. If a successful PUT request passes through a cache that has one or more stored responses for the effective request URI, those stored responses will be invalidated (see Section 4.4 of [RFC7234]).

4.3.5. DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality. In effect, this method is similar to the rm command in UNIX: it expresses a deletion operation on the URI mapping of the origin server rather than an expectation that the previously associated information be deleted.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server (which are beyond the scope of this specification). Likewise, other implementation aspects of a resource might need to be deactivated or archived as a result of a DELETE, such as database or gateway connections. In general, it is assumed that the origin server will only allow DELETE on resources for which it has a prescribed mechanism for accomplishing the deletion.

Relatively few resources allow the DELETE method -- its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location header field after a 201 (Created) response to a POST request, might allow a corresponding DELETE request to undo those actions. Similarly, custom user agent implementations that implement

an authoring function, such as revision control clients using HTTP for remote operations, might use DELETE based on an assumption that the server's URI space has been crafted to correspond to a version repository.

If a DELETE method is successfully applied, the origin server SHOULD send a 202 (Accepted) status code if the action will likely succeed but has not yet been enacted, a 204 (No Content) status code if the action has been enacted and no further information is

to be supplied, or a 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status.

A payload within a DELETE request message has no defined semantics; sending a payload body on a DELETE request might cause some existing implementations to reject the request.

Responses to the DELETE method are not cacheable. If a DELETE request passes through a cache that has one or more stored responses for the effective request URI, those stored responses will be invalidated (see Section 4.4 of [RFC7234]).

4.3.6. CONNECT

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target and, if successful, thereafter restrict its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security, [RFC5246]).

CONNECT is intended only for use in requests to a proxy. An origin server that receives a CONNECT request for itself MAY respond with a 2xx (Successful) status code to indicate that a connection is established. However, most origin servers do not implement CONNECT.

A client sending a CONNECT request MUST send the authority form of request-target (Section 5.3 of [RFC7230]); i.e., the request-target consists of only the host name and port number of the tunnel destination, separated by a colon. For example,

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
```

The recipient proxy can establish a tunnel either by directly connecting to the request-target or, if configured to use another proxy, by forwarding the CONNECT request to the next inbound proxy. Any 2xx (Successful) response indicates that the sender (and all inbound proxies) will switch to tunnel mode immediately after the blank line that concludes the successful response's header section; data received after that blank line is from the server identified by the request-target. Any response other than a successful response indicates that the tunnel has not yet been formed and that the connection remains governed by HTTP.

A tunnel is closed when a tunnel intermediary detects that either side has closed its connection: the intermediary MUST attempt to send any outstanding data that came from the closed side to the other side, close both connections, and then discard any remaining data left undelivered.

Proxy authentication might be used to establish the authority to create a tunnel. For example,

```
CONNECT server.example.com:80 HTTP/1.1
Host: server.example.com:80
Proxy-Authorization: basic aGVsbG86d29ybGQ=
```

There are significant risks in establishing a tunnel to arbitrary servers, particularly when the destination is a well-known or reserved TCP port that is not intended for Web traffic. For example, a CONNECT to a request-target of "example.com:25" would suggest that the proxy connect to the reserved port for SMTP traffic; if allowed, that could trick the proxy into relaying spam email. Proxies that support CONNECT SHOULD restrict its use to a limited set of known ports or a configurable whitelist of safe request targets.

A server MUST NOT send any Transfer-Encoding or Content-Length header fields in a 2xx (Successful) response to CONNECT. A client MUST ignore any Content-Length or Transfer-Encoding header fields received in a successful response to CONNECT.

A payload within a CONNECT request message has no defined semantics; sending a payload body on a CONNECT request might cause some existing implementations to reject the request.

Responses to the CONNECT method are not cacheable.

4.3.7. OPTIONS

The OPTIONS method requests information about the communication options available for the target resource, at either the origin server or an intervening intermediary. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.

An OPTIONS request with an asterisk ("*") as the request-target applies to the server in general rather than to a specific resource. Since a server's communication options typically depend on the resource, the "*" request is only useful as a "ping" or "no-op" type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 conformance (or lack thereof).

If the request-target is not an asterisk, the OPTIONS request applies to the options that are available when communicating with the target resource.

A server generating a successful response to OPTIONS SHOULD send any header fields that might indicate optional features implemented by the server and applicable to the target resource (e.g., Allow), including potential extensions not defined by this specification.

The response payload, if any, might also describe the communication options in a machine or human-readable representation. A standard format for such a representation is not defined by this specification, but might be defined by future extensions to HTTP. A server MUST generate a Content-Length field with a value of "0" if no payload body is to be sent in the response.

A client MAY send a Max-Forwards header field in an OPTIONS request to target a specific recipient in the request chain (see Section 5.1.2). A proxy MUST NOT generate a Max-Forwards header field while forwarding a request unless that request was received with a Max-Forwards field.

A client that generates an OPTIONS request containing a payload body MUST send a valid Content-Type header field describing the representation media type. Although this specification does not define any use for such a payload, future extensions to HTTP might use the OPTIONS body to make more detailed queries about the target resource.

Responses to the OPTIONS method are not cacheable.

4.3.8. TRACE

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the message body of a 200 (OK) response with a Content-Type of "message/http" (Section 8.3.1 of [RFC7230]). The final recipient is either the origin server or the first server to receive a Max-Forwards value of zero (0) in the request

(Section 5.1.2).

A client MUST NOT generate header fields in a TRACE request containing sensitive data that might be disclosed by the response. For example, it would be foolish for a user agent to send stored user credentials [RFC7235] or cookies [RFC6265] in a TRACE request. The final recipient of the request SHOULD exclude any request header fields that are likely to contain sensitive data when that recipient generates the response body.

TRACE allows the client to see what is being received at the other end of the request chain and use that data for testing or diagnostic information. The value of the Via header field (Section 5.7.1 of [RFC7230]) is of particular interest, since it acts as a trace of the request chain. Use of the Max-Forwards header field allows the client to limit the length of the request chain, which is useful for testing a chain of proxies forwarding messages in an infinite loop.

A client MUST NOT send a message body in a TRACE request.

Responses to the TRACE method are not cacheable.

5. Request Header Fields

A client sends request header fields to provide more information about the request context, make the request conditional based on the target resource state, suggest preferred formats for the response, supply authentication credentials, or modify the expected request processing. These fields act as request modifiers, similar to the parameters on a programming language method invocation.

5.1. Controls

Controls are request header fields that direct specific handling of the request.

Header Field Name	Defined in...
Cache-Control	Section 5.2 of [RFC7234]
Expect	Section 5.1.1
Host	Section 5.4 of [RFC7230]
Max-Forwards	Section 5.1.2
Pragma	Section 5.4 of [RFC7234]
Range	Section 3.1 of [RFC7233]
TE	Section 4.3 of [RFC7230]

5.1.1. Expect //❌ Header not mandatory

5.1.2. Max-Forwards //❌ Header not mandatory

5.2. Conditionals //❌ Headers not mandatory

5.3. Content Negotiation

The following request header fields are sent by a user agent to engage in proactive negotiation of the response content, as defined in Section 3.4.1. The preferences sent in these fields apply to any content in the response, including representations of the target resource, representations of error or processing status, and potentially even the miscellaneous text strings that might appear within the protocol.

Header Field Name	Defined in...
Accept	Section 5.3.2
Accept-Charset	Section 5.3.3
Accept-Encoding	Section 5.3.4
Accept-Language	Section 5.3.5

5.3. Quality Values

Many of the request header fields for proactive negotiation use a common parameter, named "q" (case-insensitive), to assign a relative "weight" to the preference for that associated kind of content. This weight is referred to as a "quality value" (or "qvalue") because the same parameter name is often used within server configurations to assign a weight to the relative quality of the various representations that can be selected for a resource.

The weight is normalized to a real number in the range 0 through 1, where 0.001 is the least preferred value and 1 is the most preferred; a value of 0 means "not acceptable". If no "q" parameter is present, the default weight is 1.

```
weight = OWS ";" OWS "q=" qvalue
qvalue = ( "0" [ "." 0*3DIGIT ] )
       / ( "1" [ "." 0*3("0") ] )
```

A sender of qvalue MUST NOT generate more than three digits after the decimal point. User configuration of these values ought to be limited in the same fashion.

👉 Accept (5.3.2.)

The "Accept" header field can be used by user agents to specify response media types that are acceptable. Accept header fields can be used to indicate that the request is specifically limited to a small set of desired types, as in the case of a request for an in-line image.

```
Accept = #( media-range [ accept-params ] )
media-range =( "*/*"
            / ( type "/" "*" )
            / ( type "/" subtype )
            ) *( OWS ";" OWS parameter )
accept-params = weight *( accept-ext )
accept-ext = OWS ";" OWS token [ "=" ( token / quoted-string ) ]
```

The asterisk "*" character is used to group media types into ranges, with "*/*" indicating all media types and "type/*" indicating all subtypes of that type. The media-range can include media type parameters that are applicable to that range.

Each media-range might be followed by zero or more applicable media type parameters (e.g., charset), an optional "q" parameter for indicating a relative weight (Section 5.3.1), and then zero or more extension parameters. The "q" parameter is necessary if any extensions (accept-ext) are present, since it acts as a separator between the two parameter sets.

Note: Use of the "q" parameter name to separate media type parameters from Accept extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA

media type registry and the rare usage of any media type parameters in Accept. Future media types are discouraged from registering any parameter named "q".

The example

```
Accept: audio/*; q=0.2, audio/basic
```

is interpreted as "I prefer audio/basic, but send me any audio type if it is the best available after an 80% markdown in quality".

A request without any Accept header field implies that the user agent will accept any media type in response. If the header field is present in a request and none of the available representations for the response have a media type that is listed as acceptable, the origin server can either honor the header field by sending a 406 (Not Acceptable) response or disregard the header field by treating the response as if it is not subject to content negotiation.

A more elaborate example is

```
Accept: text/plain; q=0.5, text/html,
        text/x-dvi; q=0.8, text/x-c
```

Verbally, this would be interpreted as "text/html and text/x-c are the equally preferred media types, but if they do not exist, then send the text/x-dvi representation, and if that does not exist, send the text/plain representation".

Media ranges can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence. For example,

```
Accept: text/*, text/plain, text/plain;format=flowed, */*
```

have the following precedence:

1. text/plain;format=flowed
2. text/plain
3. text/* 4. */*

The media type quality factor associated with a given type is determined by finding the media range with the highest precedence that matches the type. For example,

```
Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1,
        text/html;level=2;q=0.4, */*;q=0.5
```

would cause the following values to be associated:

Media Type	Quality Value
text/html;level=1	1
text/html	0.7
text/plain	0.3
image/jpeg	0.5
text/html;level=2	0.4
text/html;level=3	0.7

Note: A user agent might be provided with a default set of quality values for certain media ranges. However, unless the user agent is a closed system that cannot interact with other rendering agents, this default set ought to be configurable by the user.

👉 Accept-Charset (5.3.3.)

The "Accept-Charset" header field can be sent by a user agent to indicate what charsets are acceptable in textual response content. This field allows user agents capable of understanding more comprehensive or special-purpose charsets to signal that capability to an origin server that is capable of representing information in those charsets.

```
Accept-Charset = 1#( ( charset / "*" ) [ weight ] )
```

Charset names are defined in Section 3.1.1.2. A user agent MAY associate a quality value with each charset to indicate the user's relative preference for that charset, as defined in Section 5.3.1. An example is

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

The special value "*", if present in the Accept-Charset field, matches every charset that is not mentioned elsewhere in the Accept-Charset field. If no "*" is present in an Accept-Charset field, then any charsets not explicitly mentioned in the field are considered "not acceptable" to the client.

A request without any Accept-Charset header field implies that the user agent will accept any charset in response. Most general-purpose user agents do not send Accept-Charset, unless specifically configured to do so, because a detailed list of supported charsets makes it easier for a server to identify an individual by virtue of the user agent's request characteristics (Section 9.7).

If an Accept-Charset header field is present in a request and none of the available representations for the response has a charset that is listed as acceptable, the origin server can either honor the header field, by sending a 406 (Not Acceptable) response, or disregard the header field by treating the resource as if it is not subject to content negotiation.

5.3.4. Accept-Encoding //❌ Header field not mandatory

5.3.5. Accept-Language

The "Accept-Language" header field can be used by user agents to indicate the set of natural languages that are preferred in the response. Language tags are defined in Section 3.1.3.1.

```
Accept-Language = 1#( language-range [ weight ] )

language-range = <language-range, see [RFC4647], Section 2.1>
```

Each language-range can be given an associated quality value representing an estimate of the user's preference for the languages specified by that range, as defined in Section 5.3.1. For example,

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

would mean: "I prefer Danish, but will accept British English and other types of English".

A request without any Accept-Language header field implies that the user agent will accept any language in response. If the header field is present in a request and none of the available representations for the response have a matching language tag, the origin server can either disregard the header field by treating the response as if it is not subject to content negotiation or honor the header field by sending a 406 (Not Acceptable) response. However, the latter is not encouraged, as doing so can prevent users from accessing content that they might be able to use (with translation software, for example).

Note that some recipients treat the order in which language tags are listed as an indication of descending priority, particularly for tags that are assigned equal quality values (no value is the same as q=1). However, this behavior cannot be relied upon. For consistency and to maximize interoperability, many user agents assign each language tag a unique quality value while also listing them in order of decreasing quality. Additional discussion of language priority lists can be found in Section 2.3 of [RFC4647].

For matching, Section 3 of [RFC4647] defines several matching schemes. Implementations can offer the most appropriate matching scheme for their requirements. The "Basic Filtering" scheme ([RFC4647], Section 3.3.1) is identical to the matching scheme that was previously defined for HTTP in Section 14.4 of [RFC2616].

It might be contrary to the privacy expectations of the user to send an Accept-Language header field with the complete linguistic preferences of the user in every request (Section 9.7).

Since intelligibility is highly dependent on the individual user, user agents need to allow user control over the linguistic preference (either through configuration of the user agent itself or by defaulting to a user controllable system setting). A user agent that does not provide such control to the user MUST NOT send an Accept-Language header field.

Note: User agents ought to provide guidance to users when setting a preference, since users are rarely familiar with the details of language matching as described above. For example, users might assume that on selecting "en-gb", they will be served any kind of English document if British English is not available. A user agent might suggest, in such a case, to add "en" to the list for better matching behavior.

5.4. Authentication Credentials

Two header fields are used for carrying authentication credentials, as defined in [RFC7235]. Note that various custom mechanisms for user authentication use the Cookie header field for this purpose, as defined in [RFC6265].

Header Field Name	Defined in...
Authorization	Section 4.2 of [RFC7235]
Proxy-Authorization	Section 4.4 of [RFC7235]

5.5. Request Context

The following request header fields provide additional information about the request context, including information about the user, user agent, and resource behind the request.

Header Field Name	Defined in...
From	Section 5.5.1
Referer	Section 5.5.2
User-Agent	Section 5.5.3

5.5.1. From // ❌ Header field not mandatory

The "From" header field contains an Internet email address for a human user who controls the requesting user agent.

👉 Referer (5.5.2.)

The "Referer" [sic] header field allows the user agent to specify a URI reference for the resource from which the target URI was obtained (i.e., the "referrer", though the field name is misspelled). A user agent MUST NOT include the fragment and userinfo components of the URI reference [RFC3986], if any, when generating the Referer field value.

Referer = absolute-URI / partial-URI

The Referer header field allows servers to generate back-links to other resources for simple analytics, logging, optimized caching, etc. It also allows obsolete or mistyped links to be found for maintenance. Some servers use the Referer header field as a means of denying links from other sites (so-called "deep linking") or restricting cross-site request forgery (CSRF), but not all requests contain it.

Example:

Referer: http://www.example.org/hypertext/Overview.html

If the target URI was obtained from a source that does not have its own URI (e.g., input from the user keyboard, or an entry within the user's bookmarks/favorites), the user agent MUST either exclude the Referer field or send it with a value of "about:blank".

The Referer field has the potential to reveal information about the request context or browsing history of the user, which is a privacy concern if the referring resource's identifier reveals personal information (such as an account name) or a resource that is supposed to be confidential (such as behind a firewall or internal to a secured service). Most general-purpose user agents do not send the Referer header field when the referring resource is a local "file" or "data" URI. A user agent MUST NOT send a Referer header field in an unsecured HTTP request if the referring page was received with a secure protocol. See Section 9.4 for additional security considerations.

Some intermediaries have been known to indiscriminately remove Referer header fields from outgoing requests. This has the unfortunate side effect of interfering with protection against CSRF attacks, which can be far more harmful to their users. Intermediaries and user agent extensions that wish to limit information disclosure in Referer ought to restrict their changes to specific edits, such as replacing internal domain names with pseudonyms or truncating the query and/or path components. An intermediary SHOULD NOT modify or delete the Referer header field when the field value shares the same scheme and host as the request target.

👉 User-Agent (5.5.3.)

The "User-Agent" header field contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor responses to avoid particular user agent limitations, and for analytics regarding browser or operating system use. A user agent SHOULD send a User-Agent field in each request unless specifically configured not to do so.

User-Agent = product *(RWS (product / comment))

The User-Agent field-value consists of one or more product identifiers, each followed by zero or more comments (Section 3.2 of [RFC7230]), which together identify the user agent software and its significant subproducts. By convention, the product identifiers are listed in decreasing order of their significance for identifying the user agent software. Each product identifier consists of a name and optional version.

product = token ["/" product-version] product-version = token

A sender SHOULD limit generated product identifiers to what is necessary to identify the product; a sender MUST NOT generate advertising or other nonessential information within the product identifier. A sender SHOULD NOT generate information in product-version that is not a version identifier (i.e., successive versions of the same product name ought to differ only in the product-version portion of the product identifier).

Example:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

A user agent SHOULD NOT generate a User-Agent field containing needlessly fine-grained detail and SHOULD limit the addition of subproducts by third parties. Overly long and detailed User-Agent field values increase request latency and the risk of a user being identified against their wishes ("fingerprinting").

Likewise, implementations are encouraged not to use the product tokens of other implementations in order to declare compatibility with them, as this circumvents the purpose of the field. If a user agent masquerades as a different user agent, recipients can assume that the user intentionally desires to see responses tailored for that identified user agent, even if they might not work as well for the actual user agent being used.

6. Response Status Codes

The status-code element is a three-digit integer code giving the result of the attempt to understand and satisfy the request.

HTTP status codes are extensible. HTTP clients are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, a client MUST understand the class of any status code, as indicated by the first digit, and treat an unrecognized status code as being equivalent to the x00 status code of that class, with the exception that a recipient MUST NOT cache a response with an unrecognized status code.

For example, if an unrecognized status code of 471 is received by a client, the client can assume that there was something wrong with its request and treat the response as if it had received a 400 (Bad Request) status code. The response message will usually contain a representation that explains the status.

The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:

- 1xx (Informational): The request was received, continuing process
- 2xx (Successful): The request was successfully received, understood, and accepted
- 3xx (Redirection): Further action needs to be taken in order to complete the request
- 4xx (Client Error): The request contains bad syntax or cannot be fulfilled
- 5xx (Server Error): The server failed to fulfill an apparently valid request

6.1. Overview of Status Codes

The status codes listed below are defined in this specification, Section 4 of [RFC7232], Section 4 of [RFC7233], and Section 3 of [RFC7235]. The reason phrases listed here are only recommendations -- they can be replaced by local equivalents without affecting the protocol.

Responses with status codes that are defined as cacheable by default (e.g., 200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501 in this specification) can be reused by a cache with heuristic expiration unless otherwise indicated by the method definition or explicit cache controls [RFC7234]; all other status codes are not cacheable by default.

Code	Reason-Phrase	Defined in...
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5
205	Reset Content	Section 6.3.6
206	Partial Content	Section 4.1 of [RFC7233]
300	Multiple Choices	Section 6.4.1
301	Moved Permanently	Section 6.4.2
302	Found	Section 6.4.3
303	See Other	Section 6.4.4
304	Not Modified	Section 4.1 of [RFC7232]
305	Use Proxy	Section 6.4.5
307	Temporary Redirect	Section 6.4.7
400	Bad Request	Section 6.5.1
401	Unauthorized	Section 3.1 of [RFC7235]
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
407	Proxy Authentication Required	Section 3.2 of [RFC7235]
408	Request Timeout	Section 6.5.7
409	Conflict	Section 6.5.8
410	Gone	Section 6.5.9
411	Length Required	Section 6.5.10
412	Precondition Failed	Section 4.2 of [RFC7232]
413	Payload Too Large	Section 6.5.11
414	URI Too Long	Section 6.5.12
415	Unsupported Media Type	Section 6.5.13
416	Range Not Satisfiable	Section 4.4 of [RFC7233]
417	Expectation Failed	Section 6.5.14
426	Upgrade Required	Section 6.5.15
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

Note that this list is not exhaustive -- it does not include extension status codes defined in other specifications. The complete list of status codes is maintained by IANA. See Section 8.2 for details.

6.2. Informational 1xx

The 1xx (Informational) class of status code indicates an interim response for communicating connection status or request progress prior to completing the requested action and sending a final response. 1xx responses are terminated by the first empty line after the status-line (the empty line signaling the end of the header section). Since HTTP/1.0 did not define any 1xx status codes, a server MUST NOT send a 1xx response to an HTTP/1.0 client.

A client MUST be able to parse one or more 1xx responses received prior to a final response, even if the client does not expect one. A user agent MAY ignore unexpected 1xx responses.

A proxy MUST forward 1xx responses unless the proxy itself requested the generation of the 1xx response. For example, if a proxy adds an "Expect: 100-continue" field when it

forwards a request, then it need not forward the corresponding 100 (Continue) response(s).

6.2.1. 100 Continue

The 100 (Continue) status code indicates that the initial part of a request has been received and has not yet been rejected by the server. The server intends to send a final response after the request has been fully received and acted upon.

When the request contains an Expect header field that includes a 100-continue expectation, the 100 response indicates that the server wishes to receive the request payload body, as described in

Section 5.1.1. The client ought to continue sending the request and discard the 100 response.

If the request did not contain an Expect header field containing the 100-continue expectation, the client can simply discard this interim response.

6.2.2. 101 Switching Protocols

The 101 (Switching Protocols) status code indicates that the server understands and is willing to comply with the client's request, via the Upgrade header field (Section 6.7 of [RFC7230]), for a change in the application protocol being used on this connection. The server

MUST generate an Upgrade header field in the response that indicates which protocol(s) will be switched to immediately after the empty line that terminates the 101 response.

It is assumed that the server will only agree to switch protocols when it is advantageous to do so. For example, switching to a newer version of HTTP might be advantageous over older versions, and switching to a real-time, synchronous protocol might be advantageous when delivering resources that use such features.

6.3. Successful 2xx

The 2xx (Successful) class of status code indicates that the client's request was successfully received, understood, and accepted.

6.3.1. 200 OK

The 200 (OK) status code indicates that the request has succeeded. The payload sent in a 200 response depends on the request method. For the methods defined by this specification, the intended meaning of the payload can be summarized as:

GET a representation of the target resource;

HEAD the same representation as GET, but without the representation data;

POST a representation of the status of, or results obtained from, the action;

PUT, DELETE a representation of the status of the action;

OPTIONS a representation of the communications options;

TRACE a representation of the request message as received by the end server.

Aside from responses to CONNECT, a 200 response always has a payload, though an origin server MAY generate a payload body of zero length. If no payload is desired, an origin server ought to send 204 (No Content) instead. For CONNECT, no payload is allowed because the successful result is a tunnel, which begins immediately after the 200 response header section.

A 200 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.3.2. 201 Created

The 201 (Created) status code indicates that the request has been fulfilled and has resulted in one or more new resources being created. The primary resource created by the request is identified by either a Location header field in the response or, if no Location field is received, by the effective request URI.

The 201 response payload typically describes and links to the resource(s) created. See Section 7.2 for a discussion of the meaning and purpose of validator header fields, such as ETag and Last-Modified, in a 201 response.

6.3.3. 202 Accepted

The 202 (Accepted) status code indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility in HTTP for re-sending a status code from an asynchronous operation.

The 202 response is intentionally noncommittal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The representation sent with this response ought to describe the request's current status and point to (or embed) a status monitor that can provide the user with an estimate of when the request will be fulfilled.

6.3.4. 203 Non-Authoritative Information

The 203 (Non-Authoritative Information) status code indicates that the request was successful but the enclosed payload has been modified from that of the origin server's 200 (OK) response by a transforming proxy (Section 5.7.2 of [RFC7230]). This status code allows the proxy to notify recipients when a transformation has been applied, since that knowledge might impact later decisions regarding the content. For example, future cache validation requests for the content might only be applicable along the same request path (through the same proxies).

The 203 response is similar to the Warning code of 214 Transformation Applied (Section 5.5 of [RFC7234]), which has the advantage of being applicable to responses with any status code.

A 203 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.3.5. 204 No Content

The 204 (No Content) status code indicates that the server has successfully fulfilled the request and that there is no additional content to send in the response payload body. Metadata in the response header fields refer to the target resource and its selected representation after the requested action was applied.

For example, if a 204 status code is received in response to a PUT request and the response contains an ETag header field, then the PUT was successful and the ETag field-value contains the entity-tag for the new representation of that target resource.

The 204 response allows a server to indicate that the action has been successfully applied to the target resource, while implying that the user agent does not need to traverse away from its current "document view" (if any). The server assumes that the user agent will provide some indication of the success to its user, in accord with its own interface, and apply any new or updated metadata in the response to its active representation.

For example, a 204 status code is commonly used with document editing interfaces corresponding to a "save" action, such that the document being saved remains available to the user for editing. It is also frequently used with interfaces that expect automated data transfers to be prevalent, such as within distributed version control systems.

A 204 response is terminated by the first empty line after the header fields because it cannot contain a message body.

A 204 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.3.6. 205 Reset Content

The 205 (Reset Content) status code indicates that the server has fulfilled the request and desires that the user agent reset the "document view", which caused the request to be sent, to its original state as received from the origin server.

This response is intended to support a common data entry use case where the user receives content that supports data entry (a form, notepad, canvas, etc.), enters or manipulates data in that space, causes the entered data to be submitted in a request, and then the data entry mechanism is reset for the next entry so that the user can easily initiate another input action.

Since the 205 status code implies that no additional content will be provided, a server MUST NOT generate a payload in a 205 response. In other words, a server MUST do one of the following for a 205 response: a) indicate a zero-length body for the response by including a Content-Length header field with a value of 0; b) indicate a zero-length payload for the response by including a Transfer-Encoding header field with a value of chunked and a message body consisting of a single chunk of zero-length; or, c) close the connection immediately after sending the blank line terminating the header section.

6.4. Redirection 3xx

The 3xx (Redirection) class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. If a Location header field (Section 7.1.2) is provided, the user agent MAY automatically redirect its request to the URI referenced by the Location field value, even if the specific status code is not understood. Automatic redirection needs to done with care for methods not known to be safe, as defined in Section 4.2.1, since the user might not wish to redirect an unsafe request.

There are several types of redirects:

1. Redirects that indicate the resource might be available at a different URI, as provided by the Location field, as in the status codes 301 (Moved Permanently), 302 (Found), and 307 (Temporary Redirect).
2. Redirection that offers a choice of matching resources, each capable of representing the original request target, as in the 300 (Multiple Choices) status code.
3. Redirection to a different resource, identified by the Location field, that can represent an indirect response to the request, as in the 303 (See Other) status code.
4. Redirection to a previously cached result, as in the 304 (Not Modified) status code.

   Note: In HTTP/1.0, the status codes 301 (Moved Permanently) and 302 (Found) were defined for the first type of redirect ([RFC1945], Section 9.3). Early user agents split on whether the method applied to the redirect target would be the same as the original request or would be rewritten as GET. Although HTTP originally defined the former semantics for 301 and 302 (to match its original implementation at CERN), and defined 303 (See Other) to match the latter semantics, prevailing practice gradually converged on the latter semantics for 301 and 302 as well. The first revision of HTTP/1.1 added 307 (Temporary Redirect) to indicate the former semantics without being impacted by divergent practice. Over 10 years later, most user agents still do method rewriting for 301 and 302; therefore, this specification makes that behavior conformant when the original request is POST.

A client SHOULD detect and intervene in cyclical redirections (i.e., "infinite" redirection loops).

Note: An earlier version of this specification recommended a maximum of five redirections ([RFC2068], Section 10.3). Content developers need to be aware that some clients might implement such a fixed limitation.

6.4.1. 300 Multiple Choices

The 300 (Multiple Choices) status code indicates that the target resource has more than one representation, each with its own more specific identifier, and information about the alternatives is being provided so that the user (or user agent) can select a preferred representation by redirecting its request to one or more of those identifiers. In other words, the server desires that the user agent engage in reactive negotiation to select the most appropriate representation(s) for its needs (Section 3.4).

If the server has a preferred choice, the server SHOULD generate a Location header field containing a preferred choice's URI reference. The user agent MAY use the Location field value for automatic redirection.

For request methods other than HEAD, the server SHOULD generate a payload in the 300 response containing a list of representation metadata and URI reference(s) from which the user or user agent can choose the one most preferred. The user agent MAY make a selection from that list automatically if it understands the provided media type. A specific format for automatic selection is not defined by this specification because HTTP tries to remain orthogonal to the definition of its payloads. In practice, the representation is provided in some easily parsed format believed to be acceptable to the user agent, as determined by shared design or content negotiation, or in some commonly accepted hypertext format.

A 300 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

Note: The original proposal for the 300 status code defined the URI header field as providing a list of alternative representations, such that it would be usable for 200, 300, and 406 responses and be transferred in responses to the HEAD method. However, lack of deployment and disagreement over syntax led to both URI and Alternates (a subsequent proposal) being dropped from this specification. It is possible to communicate the list using a set of Link header fields [RFC5988], each with a relationship of "alternate", though deployment is a chicken-and-egg problem.

6.4.2. 301 Moved Permanently

The 301 (Moved Permanently) status code indicates that the target resource has been assigned a new permanent URI and any future references to this resource ought to use one of the enclosed URIs. Clients with link-editing capabilities ought to automatically re-link references to the effective request URI to one or more of the new references sent by the server, where possible.

The server SHOULD generate a Location header field in the response containing a preferred URI reference for the new permanent URI. The user agent MAY use the Location field value for automatic redirection. The server's response payload usually contains a short hypertext note with a hyperlink to the new URI(s).

Note: For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

A 301 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.4.3. 302 Found

The 302 (Found) status code indicates that the target resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client ought to continue to use the effective request URI for future requests.

The server SHOULD generate a Location header field in the response containing a URI reference for the different URI. The user agent MAY use the Location field value for automatic redirection. The server's response payload usually contains a short hypertext note with a hyperlink to the different URI(s).

Note: For historical reasons, a user agent MAY change the request method from POST to GET for the subsequent request. If this behavior is undesired, the 307 (Temporary Redirect) status code can be used instead.

6.4.4. 303 See Other

The 303 (See Other) status code indicates that the server is redirecting the user agent to a different resource, as indicated by a URI in the Location header field, which is intended to provide an indirect response to the original request. A user agent can perform a retrieval request targeting that URI (a GET or HEAD request if using HTTP), which might also be redirected, and present the eventual result as an answer to the original request. Note that the new URI in the Location header field is not considered equivalent to the effective request URI.

This status code is applicable to any HTTP method. It is primarily used to allow the output of a POST action to redirect the user agent to a selected resource, since doing so provides the information corresponding to the POST response in a form that can be separately identified, bookmarked, and cached, independent of the original request.

A 303 response to a GET request indicates that the origin server does not have a representation of the target resource that can be transferred by the server over HTTP. However, the Location field value refers to a resource that is descriptive of the target resource, such that making a retrieval request on that other resource might result in a representation that is useful to recipients without implying that it represents the original target resource. Note that answers to the questions of what can be represented, what representations are adequate, and what might be a useful description are outside the scope of HTTP.

Except for responses to a HEAD request, the representation of a 303 response ought to contain a short hypertext note with a hyperlink to the same URI reference provided in the Location header field.

6.4.5. 305 Use Proxy

The 305 (Use Proxy) status code was defined in a previous version of this specification and is now deprecated (Appendix B).

6.4.6. 306 (Unused)

The 306 status code was defined in a previous version of this specification, is no longer used, and the code is reserved.

6.4.7. 307 Temporary Redirect

The 307 (Temporary Redirect) status code indicates that the target resource resides temporarily under a different URI and the user agent MUST NOT change the request method if it performs an automatic redirection to that URI. Since the redirection can change over time, the client ought to continue using the original effective request URI for future requests.

The server SHOULD generate a Location header field in the response containing a URI reference for the different URI. The user agent MAY use the Location field value for automatic redirection. The server’s response payload usually contains a short hypertext note with a hyperlink to the different URI(s).

Note: This status code is similar to 302 (Found), except that it does not allow changing the request method from POST to GET. This specification defines no equivalent counterpart for 301 (Moved Permanently) ([RFC7238], however, defines the status code 308 (Permanent Redirect) for this purpose).

6.5. Client Error 4xx

The 4xx (Client Error) class of status code indicates that the client seems to have erred. Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents SHOULD display any included representation to the user.

6.5.1. 400 Bad Request

The 400 (Bad Request) status code indicates that the server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

6.5.2. 402 Payment Required

The 402 (Payment Required) status code is reserved for future use.

6.5.3. 403 Forbidden

The 403 (Forbidden) status code indicates that the server understood the request but refuses to authorize it. A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

An origin server that wishes to "hide" the current existence of a forbidden target resource MAY instead respond with a status code of 404 (Not Found).

6.5.4. 404 Not Found

The 404 (Not Found) status code indicates that the origin server did not find a current representation for the target resource or is not willing to disclose that one exists. A 404 status code does not indicate whether this lack of representation is temporary or permanent; the 410 (Gone) status code is preferred over 404 if the origin server knows, presumably through some configurable means, that the condition is likely to be permanent.

A 404 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.5.5. 405 Method Not Allowed

The 405 (Method Not Allowed) status code indicates that the method received in the request-line is known by the origin server but not supported by the target resource. The origin server MUST generate an Allow header field in a 405 response containing a list of the target resource's currently supported methods.

A 405 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.5.6. 406 Not Acceptable

The 406 (Not Acceptable) status code indicates that the target resource does not have a current representation that would be acceptable to the user agent, according to the proactive negotiation header fields received in the request (Section 5.3), and the server is unwilling to supply a default representation.

The server SHOULD generate a payload containing a list of available representation characteristics and corresponding resource identifiers from which the user or user agent can choose the one most appropriate. A user agent MAY automatically select the most appropriate choice from that list. However, this specification does not define any standard for such automatic selection, as described in Section 6.4.1.

6.5.7. 408 Request Timeout

The 408 (Request Timeout) status code indicates that the server did not receive a complete request message within the time that it was prepared to wait. A server SHOULD send the "close" connection option (Section 6.1 of [RFC7230]) in the response, since 408 implies that the server has decided to close the connection rather than continue waiting. If the client has an outstanding request in transit, the client MAY repeat that request on a new connection.

6.5.8. 409 Conflict

The 409 (Conflict) status code indicates that the request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request. The server

SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the representation being PUT included changes to a resource that conflict with those made by an earlier (third-party) request, the origin server might use a 409 response to indicate that it can't complete the request. In this case, the response representation would likely contain information useful for merging the differences based on the revision history.

6.5.9. 410 Gone

The 410 (Gone) status code indicates that access to the target resource is no longer available at the origin server and that this condition is likely to be permanent. If the origin server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) ought to be used instead.

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer associated with the origin server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time -- that is left to the discretion of the server owner.

A 410 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.5.10. 411 Length Required

The 411 (Length Required) status code indicates that the server refuses to accept the request without a defined Content-Length (Section 3.3.2 of [RFC7230]). The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message body in the request message.

6.5.11. 413 Payload Too Large

The 413 (Payload Too Large) status code indicates that the server is refusing to process a request because the request payload is larger than the server is willing or able to process. The server MAY close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD generate a Retry-After header field to indicate that it is temporary and after what time the client MAY try again.

6.5.12. 414 URI Too Long

The 414 (URI Too Long) status code indicates that the server is refusing to service the request because the request-target (Section 5.3 of [RFC7230]) is longer than the server is willing to interpret. This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a "black hole" of redirection (e.g., a redirected URI prefix that points to a suffix of itself) or when the server is under attack by a client attempting to exploit potential security holes.

A 414 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.5.13. 415 Unsupported Media Type

The 415 (Unsupported Media Type) status code indicates that the origin server is refusing to service the request because the payload is in a format not supported by this method on the target resource. The format problem might be due to the request's indicated Content-Type or Content-Encoding, or as a result of inspecting the data directly.

6.5.14. 417 Expectation Failed

The 417 (Expectation Failed) status code indicates that the expectation given in the request's Expect header field (Section 5.1.1) could not be met by at least one of the inbound servers.

6.5.15. 426 Upgrade Required

The 426 (Upgrade Required) status code indicates that the server refuses to perform the request using the current protocol but might be willing to do so after the client upgrades to a different protocol. The server MUST send an Upgrade header field in a 426 response to indicate the required protocol(s) (Section 6.7 of [RFC7230]).

Example:

HTTP/1.1 426 Upgrade Required Upgrade: HTTP/3.0 Connection: Upgrade Content-Length: 53 Content-Type: text/plain

This service requires use of the HTTP/3.0 protocol.

6.6. Server Error 5xx

The 5xx (Server Error) class of status code indicates that the server is aware that it has erred or is incapable of performing the requested method. Except when responding to a HEAD request, the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition. A user agent SHOULD display any included representation to the user. These response codes are applicable to any request method.

6.6.1. 500 Internal Server Error

The 500 (Internal Server Error) status code indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.

6.6.2. 501 Not Implemented

The 501 (Not Implemented) status code indicates that the server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

A 501 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

6.6.3. 502 Bad Gateway

The 502 (Bad Gateway) status code indicates that the server, while acting as a gateway or proxy, received an invalid response from an inbound server it accessed while attempting to fulfill the request.

6.6.4. 503 Service Unavailable

The 503 (Service Unavailable) status code indicates that the server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay. The server MAY send a Retry-After header field

(Section 7.1.3) to suggest an appropriate amount of time for the client to wait before retrying the request.

Note: The existence of the 503 status code does not imply that a server has to use it when becoming overloaded. Some servers might simply refuse the connection.

6.6.5. 504 Gateway Timeout

The 504 (Gateway Timeout) status code indicates that the server, while acting as a gateway or proxy, did not receive a timely response from an upstream server it needed to access in order to complete the request.

6.6.6. 505 HTTP Version Not Supported

The 505 (HTTP Version Not Supported) status code indicates that the server does not support, or refuses to support, the major version of HTTP that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, as described in Section 2.6 of [RFC7230], other than with this error message. The server SHOULD generate a representation for the 505 response that describes why that version is not supported and what other protocols are supported by that server.

7. Response Header Fields

The response header fields allow the server to pass additional information about the response beyond what is placed in the status-line. These header fields give information about the server, about further access to the target resource, or about related resources.

Although each response header field has a defined meaning, in general, the precise semantics might be further refined by the semantics of the request method and/or response status code.

7.1. Control Data

Response header fields can supply control data that supplements the status code, directs caching, or instructs the client where to go next.

Header Field Name	Defined in...
Age	Section 5.1 of [RFC7234]
Cache-Control	Section 5.2 of [RFC7234]
Expires	Section 5.3 of [RFC7234]
Date	Section 7.1.1.2
Location	Section 7.1.2
Retry-After	Section 7.1.3
Vary	Section 7.1.4
Warning	Section 5.5 of [RFC7234]

7.1.1. Origination Date

7.1.1.1. Date/Time Formats

Prior to 1995, there were three different formats commonly used by servers to communicate timestamps. For compatibility with old implementations, all three are defined here. The preferred format is a fixed-length and single-zone subset of the date and time specification used by the Internet Message Format [RFC5322].

```
HTTP-date = IMF-fixdate / obs-date
```

An example of the preferred format is

```
      Sun, 06 Nov 1994 08:49:37 GMT  ;     IMF-fixdate
```

Examples of the two obsolete formats are

```
Sunday, 06-Nov-94 08:49:37 GMT       ; obsolete RFC 850 format

Sun Nov 6 08:49:37 1994              ; ANSI C's asctime() format
```

A recipient that parses a timestamp value in an HTTP header field MUST accept all three HTTP-date formats. When a sender generates a header field that contains one or more timestamps defined as HTTP-date, the sender MUST generate those timestamps in the IMF-fixdate format.

An HTTP-date value represents time as an instance of Coordinated Universal Time (UTC). The first two formats indicate UTC by the three-letter abbreviation for Greenwich Mean Time, "GMT", a predecessor of the UTC name; values in the asctime format are assumed to

be in UTC. A sender that generates HTTP-date values from a local clock ought to use NTP ([RFC5905]) or some similar protocol to synchronize its clock to UTC.

  Preferred format:

```
      IMF-fixdate = day-name "," SP date1 SP time-of-day SP GMT
      ; fixed length/zone/capitalization subset of the format
      ; see Section 3.3 of [RFC5322]

day-name = %x4D.6F.6E ; "Mon", case-sensitive
            / %x54.75.65 ; "Tue", case-sensitive
            / %x57.65.64 ; "Wed", case-sensitive
            / %x54.68.75 ; "Thu", case-sensitive
            / %x46.72.69 ; "Fri", case-sensitive
            / %x53.61.74 ; "Sat", case-sensitive
            / %x53.75.6E ; "Sun", case-sensitive
      date1       = day SP month SP year
                    ; e.g., 02 Jun 1982

      day         = 2DIGIT


      month       = %x4A.61.6E ; "Jan", case-sensitive
                  / %x46.65.62 ; "Feb", case-sensitive
                  / %x4D.61.72 ; "Mar", case-sensitive
                  / %x41.70.72 ; "Apr", case-sensitive
                  / %x4D.61.79 ; "May", case-sensitive
                  / %x4A.75.6E ; "Jun", case-sensitive
                  / %x4A.75.6C ; "Jul", case-sensitive
                  / %x41.75.67 ; "Aug", case-sensitive
                  / %x53.65.70 ; "Sep", case-sensitive
                  / %x4F.63.74 ; "Oct", case-sensitive
                  / %x4E.6F.76 ; "Nov", case-sensitive
                  / %x44.65.63 ; "Dec", case-sensitive
      year        = 4DIGIT

      GMT         = %x47.4D.54 ; "GMT", case-sensitive

      time-of-day = hour ":" minute ":" second
                    ; 00:00:00 - 23:59:60 (leap second)
      hour        = 2DIGIT

      minute      = 2DIGIT

      second      = 2DIGIT
```

Obsolete formats:

```
      obs-date    = rfc850-date / asctime-date

      rfc850-date = day-name-l "," SP date2 SP time-of-day SP GMT
      date2       = day "-" month "-" 2DIGIT
                    ; e.g., 02-Jun-82
```

```
day-name-l    = %x4D.6F.6E.64.61.79    ; "Monday", case-sensitive
       / %x54.75.65.73.64.61.79       ; "Tuesday", case-sensitive
       / %x57.65.64.6E.65.73.64.61.79 ; "Wednesday", case-sensitive
       / %x54.68.75.72.73.64.61.79    ; "Thursday", case-sensitive
       / %x46.72.69.64.61.79          ; "Friday", case-sensitive
       / %x53.61.74.75.72.64.61.79    ; "Saturday", case-sensitive
       / %x53.75.6E.64.61.79          ; "Sunday", case-sensitive


asctime-date = day-name SP date3 SP time-of-day SP year
date3        = month SP ( 2DIGIT / ( SP 1DIGIT ))
             ; e.g., Jun  2
```

HTTP-date is case sensitive. A sender MUST NOT generate additional whitespace in an HTTP-date beyond that specifically included as SP in the grammar. The semantics of day-name, day, month, year, and time-of-day are the same as those defined for the Internet Message Format constructs with the corresponding name ([RFC5322], Section 3.3).

Recipients of a timestamp value in rfc850-date format, which uses a two-digit year, MUST interpret a timestamp that appears to be more than 50 years in the future as representing the most recent year in the past that had the same last two digits.

Recipients of timestamp values are encouraged to be robust in parsing timestamps unless otherwise restricted by the field definition. For example, messages are occasionally forwarded over HTTP from a non-HTTP source that might generate any of the date and time specifications defined by the Internet Message Format.

Note: HTTP requirements for the date/time stamp format apply only to their usage within the protocol stream. Implementations are not required to use these formats for user presentation, request logging, etc.

👉 Date (7.1.1.2.)

The "Date" header field represents the date and time at which the message was originated. The field value is an HTTP-date, as defined in Section 7.1.1.1.

```
Date = HTTP-date
```

An example is

```
Date: Tue, 15 Nov 1994 08:12:31 GMT
```

When a Date header field is generated, the sender SHOULD generate its field value as the best available approximation of the date and time of message generation. In theory, the date ought to represent the moment just before the payload is generated. In practice, the date can be generated at any time during message origination.

An origin server MUST NOT send a Date header field if it does not have a clock capable of providing a reasonable approximation of the current instance in Coordinated Universal Time. An origin server MAY send a Date header field if the response is in the 1xx (Informational) or 5xx (Server Error) class of status codes. An origin server MUST send a Date header field in all other cases.

A recipient with a clock that receives a response message without a Date header field MUST record the time it was received and append a corresponding Date header field to the message's header section if it is cached or forwarded downstream.

A user agent MAY send a Date header field in a request, though generally will not do so unless it is believed to convey useful information to the server. For example, custom applications of HTTP might convey a Date if the server is expected to adjust its interpretation of the user's request based on differences between the user agent and server clocks.

👉 Location (7.1.2.)

The "Location" header field is used in some responses to refer to a specific resource in relation to the response. The type of relationship is defined by the combination of request method and status code semantics.

```
Location = URI-reference
```

The field value consists of a single URI-reference. When it has the form of a relative reference ([RFC3986], Section 4.2), the final value is computed by resolving it against the effective request URI ([RFC3986], Section 5).

For 201 (Created) responses, the Location value refers to the primary resource created by the request. For 3xx (Redirection) responses, the Location value refers to the preferred target resource for automatically redirecting the request.

If the Location value provided in a 3xx (Redirection) response does not have a fragment component, a user agent MUST process the redirection as if the value inherits the fragment component of the URI reference used to generate the request target (i.e., the redirection inherits the original reference's fragment, if any).

For example, a GET request generated for the URI reference "http://www.example.org/~tim" might result in a 303 (See Other) response containing the header field:

```
Location: /People.html#tim
```

which suggests that the user agent redirect to "http://www.example.org/People.html#tim"

Likewise, a GET request generated for the URI reference "http://www.example.org/index.html#larry" might result in a 301 (Moved Permanently) response containing the header field:

```
Location: http://www.example.net/index.html
```

which suggests that the user agent redirect to "http://www.example.net/index.html#larry", preserving the original fragment identifier.

There are circumstances in which a fragment identifier in a Location value would not be appropriate. For example, the Location header field in a 201 (Created) response is supposed to provide a URI that is specific to the created resource.

Note: Some recipients attempt to recover from Location fields that are not valid URI references. This specification does not mandate or define such processing, but does allow it for the sake of robustness.

Note: The Content-Location header field (Section 3.1.4.2) differs from Location in that the Content-Location refers to the most specific resource corresponding to the enclosed representation. It is therefore possible for a response to contain both the Location and Content-Location header fields.

👉 Retry-After (7.1.3.)

Servers send the "Retry-After" header field to indicate how long the user agent ought to wait before making a follow-up request. When sent with a 503 (Service Unavailable) response, Retry-After indicates how long the service is expected to be unavailable to the client. When sent with any 3xx (Redirection) response, Retry-After indicates the minimum time that the user agent is asked to wait before issuing the redirected request.

The value of this field can be either an HTTP-date or a number of seconds to delay after the response is received.

```
Retry-After = HTTP-date / delay-seconds
```

A delay-seconds value is a non-negative decimal integer, representing time in seconds.

```
delay-seconds = 1*DIGIT
```

Two examples of its use are

```
Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
```

```
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

7.1.4. Vary // ❌ Header field not mandatory

The "Vary" header field in a response describes what parts of a request message, aside from the method, Host header field, and request target, might influence the origin server's process for selecting and representing this response. The value consists of either a single asterisk ("*") or a list of header field names (case-insensitive).

7.2. Validator Header Fields

Validator header fields convey metadata about the selected representation (Section 3). In responses to safe requests, validator fields describe the selected representation chosen by the origin server while handling the response. Note that, depending on the status code semantics, the selected representation for a given response is not necessarily the same as the representation enclosed as response payload.

In a successful response to a state-changing request, validator fields describe the new representation that has replaced the prior selected representation as a result of processing the request.

For example, an ETag header field in a 201 (Created) response communicates the entity-tag of the newly created resource's representation, so that it can be used in later conditional requests to prevent the "lost update" problem [RFC7232].

Header Field Name	Defined in...
ETag	Section 2.3 of [RFC7232]
Last-Modified	Section 2.2 of [RFC7232]

7.3. Authentication Challenges

Authentication challenges indicate what mechanisms are available for the client to provide authentication credentials in future requests.

Header Field Name	Defined in...
WWW-Authenticate	Section 4.1 of [RFC7235]
Proxy-Authenticate	Section 4.3 of [RFC7235]

7.4. Response Context

The remaining response header fields provide more information about

the target resource for potential use in later requests.

Header Field Name	Defined in...
Accept-Ranges	Section 2.3 of [RFC7233]
Allow	Section 7.4.1
Server	Section 7.4.2

👉 Allow (7.4.1.)

The "Allow" header field lists the set of methods advertised as supported by the target resource. The purpose of this field is strictly to inform the recipient of valid request methods associated with the resource.

```
Allow = #method
```

Example of use:

```
Allow: GET, HEAD, PUT
```

The actual set of allowed methods is defined by the origin server at the time of each request. An origin server MUST generate an Allow field in a 405 (Method Not Allowed) response and MAY do so in any other response. An empty Allow field value indicates that the resource allows no methods, which might occur in a 405 response if the resource has been temporarily disabled by configuration.

A proxy MUST NOT modify the Allow header field -- it does not need to understand all of the indicated methods in order to handle them according to the generic message handling rules.

👉 **Server (7.4.2.)**

The "Server" header field contains information about the software used by the origin server to handle the request, which is often used by clients to help identify the scope of reported interoperability problems, to work around or tailor requests to avoid particular server limitations, and for analytics regarding server or operating system use. An origin server MAY generate a Server field in its responses.

```
Server = product *( RWS ( product / comment ) )
```

The Server field-value consists of one or more product identifiers, each followed by zero or more comments (Section 3.2 of [RFC7230]), which together identify the origin server software and its significant subproducts. By convention, the product identifiers are listed in decreasing order of their significance for identifying the origin server software. Each product identifier consists of a name and optional version, as defined in Section 5.5.3.

Example:

```
Server: CERN/3.0 libwww/2.17
```

An origin server SHOULD NOT generate a Server field containing needlessly fine-grained detail and SHOULD limit the addition of subproducts by third parties. Overly long and detailed Server field values increase response latency and potentially reveal internal implementation details that might make it (slightly) easier for attackers to find and exploit known security holes.

8. IANA Considerations

Method	Safe	Idempotent	Reference
CONNECT	no	no	Section 4.3.6
DELETE	no	yes	Section 4.3.5
GET	yes	yes	Section 4.3.1
HEAD	yes	yes	Section 4.3.2
OPTIONS	yes	yes	Section 4.3.7
POST	no	no	Section 4.3.3
PUT	no	yes	Section 4.3.4
TRACE	yes	yes	Section 4.3.8

==========

RFC 7232

👉 **HEADER**

==========

1. Introduction

Conditional requests are HTTP requests [RFC7231] that include one or more header fields indicating a precondition to be tested before applying the method semantics to the target resource. This document defines the HTTP/1.1 conditional request mechanisms in terms of the architecture, syntax notation, and conformance criteria defined in [RFC7230].

Conditional GET requests are the most efficient mechanism for HTTP cache updates [RFC7234]. Conditionals can also be applied to state-changing methods, such as PUT and DELETE, to prevent the "lost update" problem: one client accidentally overwriting the work of another client that has been acting in parallel.

Conditional request preconditions are based on the state of the target resource as a whole (its current value set) or the state as observed in a previously obtained representation (one value in that set). A resource might have multiple current representations, each with its own observable state. The conditional request mechanisms assume that the mapping of requests to a "selected representation" (Section 3 of [RFC7231]) will be consistent over time if the server intends to take advantage of conditionals. Regardless, if the mapping is inconsistent and the server is unable to select the appropriate representation, then no harm will result when the precondition evaluates to false.

The conditional request preconditions defined by this specification (Section 3) are evaluated when applicable to the recipient (Section 5) according to their order of precedence (Section 6).

2. Validators

This specification defines two forms of metadata that are commonly used to observe resource state and test for preconditions: modification dates (Section 2.2) and opaque entity tags (Section 2.3). Additional metadata that reflects resource state has been defined by various extensions of HTTP, such as Web Distributed Authoring and Versioning (WebDAV, [RFC4918]), that are beyond the scope of this specification. A resource metadata value is referred to as a "validator" when it is used within a precondition.

2.1. Weak versus Strong

Validators come in two flavors: strong or weak. Weak validators are easy to generate but are far less useful for comparisons. Strong validators are ideal for comparisons but can be very difficult (and occasionally impossible) to generate efficiently. Rather than impose that all forms of resource adhere to the same strength of validator, HTTP exposes the type of validator in use and imposes restrictions on when weak validators can be used as preconditions.

A "strong validator" is representation metadata that changes value whenever a change occurs to the representation data that would be observable in the payload body of a 200 (OK) response to GET.

A strong validator might change for reasons other than a change to the representation data, such as when a semantically significant part of the representation metadata is changed (e.g., Content-Type), but it is in the best interests of the origin server to only change the value when it is necessary to invalidate the stored responses held by remote caches and authoring tools.

Cache entries might persist for arbitrarily long periods, regardless of expiration times. Thus, a cache might attempt to validate an entry using a validator that it obtained in the distant past. A strong validator is unique across all versions of all representations associated with a particular resource over time. However, there is no implication of uniqueness across representations of different resources (i.e., the same strong validator might be in use for representations of multiple resources at the same time and does not imply that those representations are equivalent).

There are a variety of strong validators used in practice. The best are based on strict revision control, wherein each change to a representation always results in a unique node name and revision identifier being assigned before the representation is made accessible to GET. A collision-resistant hash function applied to the representation data is also sufficient if the data is available prior to the response header fields being sent and

the digest does not need to be recalculated every time a validation request is received. However, if a resource has distinct representations that differ only in their metadata, such as might occur with content negotiation over media types that happen to share the same data format, then the origin server needs to incorporate additional information in the validator to distinguish those representations.

In contrast, a "weak validator" is representation metadata that might not change for every change to the representation data. This weakness might be due to limitations in how the value is calculated, such as clock resolution, an inability to ensure uniqueness for all possible representations of the resource, or a desire of the resource owner to group representations by some self-determined set of equivalency rather than unique sequences of data. An origin server SHOULD change a weak entity-tag whenever it considers prior representations to be unacceptable as a substitute for the current representation. In other words, a weak entity-tag ought to change whenever the origin server wants caches to invalidate old responses.

For example, the representation of a weather report that changes in content every second, based on dynamic measurements, might be grouped into sets of equivalent representations (from the origin server's perspective) with the same weak validator in order to allow cached representations to be valid for a reasonable period of time (perhaps adjusted dynamically based on server load or weather quality). Likewise, a representation's modification time, if defined with only one-second resolution, might be a weak validator if it is possible for the representation to be modified twice during a single second and retrieved between those modifications.

Likewise, a validator is weak if it is shared by two or more representations of a given resource at the same time, unless those representations have identical representation data. For example, if the origin server sends the same validator for a representation with a gzip content coding applied as it does for a representation with no content coding, then that validator is weak. However, two simultaneous representations might share the same strong validator if they differ only in the representation metadata, such as when two different media types are available for the same representation data.

Strong validators are usable for all conditional requests, including cache validation, partial content ranges, and "lost update" avoidance. Weak validators are only usable when the client does not require exact equality with previously obtained representation data, such as when validating a cache entry or limiting a web traversal to recent changes.

👉 Last-Modified (2.2.)

The "Last-Modified" header field in a response provides a timestamp indicating the date and time at which the origin server believes the selected representation was last modified, as determined at the conclusion of handling the request.

```
Last-Modified = HTTP-date
```

An example of its use is

```
Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

2.2.1. Generation

An origin server SHOULD send Last-Modified for any selected representation for which a last modification date can be reasonably and consistently determined, since its use in conditional requests and evaluating cache freshness ([RFC7234]) results in a substantial reduction of HTTP traffic on the Internet and can be a significant factor in improving service scalability and reliability.

A representation is typically the sum of many parts behind the resource interface. The last-modified time would usually be the most recent time that any of those parts were changed. How that value is determined for any given resource is an implementation detail beyond the scope of this specification. What matters to HTTP is how recipients of the Last-Modified header field can use its value to make conditional requests and test the validity of locally cached responses.

An origin server SHOULD obtain the Last-Modified value of the representation as close as possible to the time that it generates the Date field value for its response. This allows

a recipient to make an accurate assessment of the representation's modification time, especially if the representation changes near the time that the response is generated.

An origin server with a clock MUST NOT send a Last-Modified date that is later than the server's time of message origination (Date). If the last modification time is derived from implementation-specific

metadata that evaluates to some time in the future, according to the origin server's clock, then the origin server MUST replace that value with the message origination date. This prevents a future modification date from having an adverse impact on cache validation.

An origin server without a clock MUST NOT assign Last-Modified values to a response unless these values were associated with the resource by some other system or user with a reliable clock.

2.2.2. Comparison

A Last-Modified time, when used as a validator in a request, is implicitly weak unless it is possible to deduce that it is strong, using the following rules:

o The validator is being compared by an origin server to the actual current validator for the representation and,

o That origin server reliably knows that the associated representation did not change twice during the second covered by the presented validator.

or

o The validator is about to be used by a client in an If-Modified-Since, If-Unmodified-Since, or If-Range header field, because the client has a cache entry for the associated representation, and

o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and

o The presented Last-Modified time is at least 60 seconds before the Date value.

or

o The validator is being compared by an intermediate cache to the validator stored in its cache entry for the representation, and

o That cache entry includes a Date value, which gives the time when the origin server sent the original response, and

o The presented Last-Modified time is at least 60 seconds before the Date value.

This method relies on the fact that if two different responses were sent by the origin server during the same second, but both had the same Last-Modified time, then at least one of those responses would have a Date value equal to its Last-Modified time. The arbitrary 60-second limit guards against the possibility that the Date and Last-Modified values are generated from different clocks or at somewhat different times during the preparation of the response. An implementation MAY use a value larger than 60 seconds, if it is believed that 60 seconds is too short.

2.3. ETag //❌ Header Field not mandatory

The "ETag" header field in a response provides the current entity-tag for the selected representation, as determined at the conclusion of handling the request.

2.3.1. Generation

2.3.2. Comparison

2.3.3. Example: Entity-Tags Varying on Content-Negotiated Resources

2.4. When to Use *(Entity-Tags and)* Last-Modified Dates

In 200 (OK) responses to GET or HEAD, an origin server:

o SHOULD send a Last-Modified value if it is feasible to send one.

In other words, the preferred behavior for an origin server is to send a Last-Modified value in successful responses to a retrieval request.

A client:

o SHOULD send the Last-Modified value in non-subrange cache validation requests (using If-Modified-Since) if only a Last-Modified value has been provided by the origin server.

o MAY send the Last-Modified value in subrange cache validation requests (using If-Unmodified-Since) if only a Last-Modified value has been provided by an HTTP/1.0 origin server. The user agent SHOULD provide a way to disable this, in case of difficulty.

o SHOULD send both validators in cache validation requests if both an entity-tag and a Last-Modified value have been provided by the origin server. This allows both HTTP/1.0 and HTTP/1.1 caches to respond appropriately.

3. Precondition Header Fields //❌ Header Field not mandatory

This section defines the syntax and semantics of HTTP/1.1 header fields for applying preconditions on requests. Section 5 defines when the preconditions are applied. Section 6 defines the order of evaluation when more than one precondition is present.

3.1. If-Match

3.2. If-None-Match

3.3. If-Modified-Since

3.4. If-Unmodified-Since

3.5. If-Range

… I think the rest of RFC 7232 is not useful for us ?!

==========

RFC 7233

👉 **HEADER**

==========

1. Introduction

Hypertext Transfer Protocol (HTTP) clients often encounter interrupted data transfers as a result of canceled requests or dropped connections. When a client has stored a partial representation, it is desirable to request the remainder of that representation in a subsequent request rather than transfer the entire representation. Likewise, devices with limited local storage might benefit from being able to request only a subset of a larger representation, such as a single page of a very large document, or the dimensions of an embedded image.

This document defines HTTP/1.1 range requests, partial responses, and the multipart/byteranges media type. Range requests are an OPTIONAL feature of HTTP, designed so that recipients not implementing this feature (or not supporting it for the target resource) can respond as if it is a normal GET request without impacting interoperability. Partial responses are indicated by a distinct status code to not be mistaken for full responses by caches that might not implement the feature.

Although the range request mechanism is designed to allow for extensible range types, this specification only defines requests for byte ranges.

⇒ I think we don't need this RFC ...

==========

RFC 7234

👉 **HEADER**

==========

1. Introduction

HTTP is typically used for distributed information systems, where performance can be improved by the use of response caches. This document defines aspects of HTTP/1.1 related to caching and reusing response messages.

An HTTP cache is a local store of response messages and the subsystem that controls storage, retrieval, and deletion of messages in it. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests.

Any client or server MAY employ a cache, though a cache cannot be used by a server that is acting as a tunnel.

A shared cache is a cache that stores responses to be reused by more than one user; shared caches are usually (but not always) deployed as a part of an intermediary. A private cache, in contrast, is dedicated to a single user; often, they are deployed as a component of a user agent.

The goal of caching in HTTP/1.1 is to significantly improve performance by reusing a prior response message to satisfy a current request. A stored response is considered "fresh", as defined in Section 4.2, if the response can be reused without "validation" (checking with the origin server to see if the cached response remains valid for this request). A fresh response can therefore reduce both latency and network overhead each time it is reused.

When a cached response is not fresh, it might still be reusable if it can be freshened by validation (Section 4.3) or if the origin is unavailable (Section 4.2.4).

⇒ I think we don't need this RFC ...

==========

RFC 7235

👉 **HEADER**

==========

1. Introduction

HTTP provides a general framework for access control and authentication, via an extensible set of challenge-response authentication schemes, which can be used by a server to challenge a client request and by a client to provide authentication information. This document defines HTTP/1.1 authentication in terms of the architecture defined in "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing" [RFC7230], including the general framework previously described in "HTTP Authentication: Basic and Digest Access Authentication" [RFC2617] and the related fields and status codes previously defined in "Hypertext Transfer Protocol -- HTTP/1.1" [RFC2616].

The IANA Authentication Scheme Registry (Section 5.1) lists registered authentication schemes and their corresponding specifications, including the "basic" and "digest" authentication schemes previously defined by RFC 2617.

2. Access Authentication Framework

2.1. Challenge and Response

HTTP provides a simple challenge-response authentication framework that can be used by a server to challenge a client request and by a client to provide authentication information. It uses a case- insensitive token as a means to identify the authentication scheme, followed by additional information necessary for achieving authentication via that scheme. The latter can be either a comma- separated list of parameters or a single sequence of characters capable of holding base64-encoded information.

Authentication parameters are name=value pairs, where the name token is matched case-insensitively, and each parameter name MUST only occur once per challenge.

```
auth-scheme = token

auth-param = token BWS "=" BWS ( token / quoted-string )

token68 = 1*( ALPHA / DIGIT / "-" / "." / "_" / "~" / "+" / "/" ) * "="
```

The token68 syntax allows the 66 unreserved URI characters ([RFC3986]), plus a few others, so that it can hold a base64, base64url (URL and filename safe alphabet), base32, or base16 (hex) encoding, with or without padding, but excluding whitespace ([RFC4648]).

A 401 (Unauthorized) response message is used by an origin server to challenge the authorization of a user agent, including a WWW-Authenticate header field containing at least one challenge applicable to the requested resource.

A 407 (Proxy Authentication Required) response message is used by a proxy to challenge the authorization of a client, including a Proxy-Authenticate header field containing at least one challenge applicable to the proxy for the requested resource.

```
challenge = auth-scheme [ 1*SP ( token68 / #auth-param ) ]
```

Note: Many clients fail to parse a challenge that contains an unknown scheme. A workaround for this problem is to list well- supported schemes (such as "basic") first.

A user agent that wishes to authenticate itself with an origin server -- usually, but not necessarily, after receiving a 401 (Unauthorized) -- can do so by including an Authorization header field with the request.

A client that wishes to authenticate itself with a proxy -- usually, but not necessarily, after receiving a 407 (Proxy Authentication Required) -- can do so by including a Proxy-Authorization header field with the request.

Both the Authorization field value and the Proxy-Authorization field value contain the client's credentials for the realm of the resource being requested, based upon a challenge received in a response (possibly at some point in the past). When creating their values, the user agent ought to do so by selecting the challenge with what it considers to be the most secure auth-scheme that it understands, obtaining credentials from the user as appropriate. Transmission of credentials within header field values implies significant security considerations regarding the confidentiality of the underlying connection, as described in Section 6.1.

```
credentials = auth-scheme [ 1*SP ( token68 / #auth-param ) ]
```

Upon receipt of a request for a protected resource that omits credentials, contains invalid credentials (e.g., a bad password) or partial credentials (e.g., when the authentication scheme requires more than one round trip), an origin server SHOULD send a 401 (Unauthorized) response that contains a WWW-Authenticate header field with at least one (possibly new) challenge applicable to the requested resource.

Likewise, upon receipt of a request that omits proxy credentials or contains invalid or partial proxy credentials, a proxy that requires authentication SHOULD generate a 407 (Proxy Authentication Required) response that contains a Proxy-Authenticate header field with at least one (possibly new) challenge applicable to the proxy.

A server that receives valid credentials that are not adequate to gain access ought to respond with the 403 (Forbidden) status code (Section 6.5.3 of [RFC7231]).

HTTP does not restrict applications to this simple challenge-response framework for access authentication. Additional mechanisms can be used, such as authentication at the transport level or via message encapsulation, and with additional header fields specifying authentication information. However, such additional mechanisms are not defined by this specification.

2.2. Protection Space (Realm)

The "realm" authentication parameter is reserved for use by authentication schemes that wish to indicate a scope of protection.

A protection space is defined by the canonical root URI (the scheme and authority components of the effective request URI; see Section 5.5 of [RFC7230]) of the server being accessed, in combination with the realm value if present. These realms allow the protected resources on a server to be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database. The realm value is a string, generally assigned by the origin server, that can have additional semantics specific to the authentication scheme. Note that a response can have multiple challenges with the same auth-scheme but with different realms.

The protection space determines the domain over which credentials can be automatically applied. If a prior request has been authorized, the user agent MAY reuse the same credentials for all other requests within that protection space for a period of time determined by the authentication scheme, parameters, and/or user preferences (such as a configurable inactivity timeout). Unless specifically allowed by the authentication scheme, a single protection space cannot extend outside the scope of its server.

For historical reasons, a sender MUST only generate the quoted-string syntax. Recipients might have to support both token and quoted-string syntax for maximum interoperability with existing clients that have been accepting both notations for a long time.

3. Status Code Definitions

3.1. 401 Unauthorized

The 401 (Unauthorized) status code indicates that the request has not been applied because it lacks valid authentication credentials for the target resource. The server generating a 401 response MUST send a WWW-Authenticate header field (Section 4.1) containing at least one challenge applicable to the target resource.

If the request included authentication credentials, then the 401 response indicates that authorization has been refused for those credentials. The user agent MAY repeat the request with a new or replaced Authorization header field (Section 4.2). If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user agent SHOULD present the enclosed representation to the user, since it usually contains relevant diagnostic information.

3.2. 407 Proxy Authentication Required

The 407 (Proxy Authentication Required) status code is similar to 401 (Unauthorized), but it indicates that the client needs to authenticate itself in order to use a proxy. The proxy MUST send a Proxy-Authenticate header field (Section 4.3) containing a challenge applicable to that proxy for the target resource. The client MAY repeat the request with a new or replaced Proxy-Authorization header field (Section 4.4).

4. Header Field Definitions

This section defines the syntax and semantics of header fields related to the HTTP authentication framework.

👉 WWW-Authenticate (4.1.)

The "WWW-Authenticate" header field indicates the authentication scheme(s) and parameters applicable to the target resource.

```
WWW-Authenticate = 1#challenge
```

A server generating a 401 (Unauthorized) response MUST send a WWW-Authenticate header field containing at least one challenge. A server MAY generate a WWW-Authenticate header field in other response messages to indicate that supplying credentials (or different credentials) might affect the response.

A proxy forwarding a response MUST NOT modify any WWW-Authenticate fields in that response.

User agents are advised to take special care in parsing the field value, as it might contain more than one challenge, and each challenge can contain a comma-separated list of authentication parameters. Furthermore, the header field itself can occur multiple times.

For instance:

```
WWW-Authenticate: Newauth realm="apps", type=1,

                  title="Login to \"apps\"", Basic realm="simple"
```

This header field contains two challenges; one for the "Newauth" scheme with a realm value of "apps", and two additional parameters "type" and "title", and another one for the "Basic" scheme with a realm value of "simple".

Note: The challenge grammar production uses the list syntax as well. Therefore, a sequence of comma, whitespace, and comma can be considered either as applying to the preceding challenge, or to be an empty entry in the list of challenges. In practice, this ambiguity does not affect the semantics of the header field value and thus is harmless.

👉 Authorization (4.2.)

The "Authorization" header field allows a user agent to authenticate itself with an origin server -- usually, but not necessarily, after receiving a 401 (Unauthorized) response. Its value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.

```
    Authorization = credentials
```

If a request is authenticated and a realm specified, the same credentials are presumed to be valid for all other requests within this realm (assuming that the authentication scheme itself does not require otherwise, such as credentials that vary according to a challenge value or using synchronized clocks).

A proxy forwarding a request MUST NOT modify any Authorization fields in that request. See Section 3.2 of [RFC7234] for details of and requirements pertaining to handling of the Authorization field by HTTP caches.

4.3. Proxy-Authenticate //❌ Header not mandatory

4.4. Proxy-Authorization //❌ Header not mandatory