HTTP Working Group
Internet-Draft
Obsoletes: 7230,7231,7232,7233,7235 (if approved)
Intended status: Standards Track
Expires: December 2, 2018

R. Fielding, Ed.
Adobe
M. Nottingham, Ed.
Fastly
J. Reschke, Ed.
greenbytes
May 31, 2018

HTTP Semantics
draft-ietf-httpbis-semantics-01

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines the semantics of HTTP: its architecture, terminology, the "http" and "https" Uniform Resource Identifier (URI) schemes, core request methods, request header fields, response status codes, response header fields, and content negotiation.

This document obsoletes RFC 7231, RFC 7232, RFC 7233, RFC 7235, and portions of RFC 7230.

Editorial Note

This note is to be removed before publishing as an RFC.

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/>.

Working Group information can be found at <https://httpwg.org/>; source code and issues list for this draft can be found at <https://github.com/httpwg/http-core>.

The changes in this draft are summarized in Appendix G.2.

Status of This Memo

This Internet-Draft will expire on December 2, 2018.

Copyright Notice

Table of Contents

1. Introduction

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

o "HTTP Semantics" (this document)

o "HTTP Caching" [Caching]

o "HTTP/1.1 Messaging" [Messaging]

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.

Each 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.

HTTP provides a uniform interface for interacting with a resource (Section 2.5), regardless of its type, nature, or implementation, via the manipulation and transfer of representations (Section 6).

This document defines semantics that are common to all versions of HTTP. HTTP semantics include the intentions defined by each request method (Section 7), extensions to those semantics that might be described in request header fields (Section 8), the meaning of status codes to indicate a machine-readable response (Section 9), and the meaning of other control data and resource metadata that might be given in response header fields (Section 10).

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" (Section 6.4).

This document defines HTTP/1.1 range requests, partial responses, and the multipart/byteranges media type.

This document obsoletes the portions of RFC 7230 that are independent of the HTTP/1.1 messaging syntax and connection management, with the changes being summarized in Appendix B. The other parts of RFC 7230 are obsoleted by "HTTP/1.1 Messaging" [Messaging]. This document also obsoletes RFC 7231 (see Appendix C), RFC 7232 (see Appendix D), RFC 7233 (see Appendix E), and RFC 7235 (see Appendix F).

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Conformance criteria and considerations regarding error handling are defined in Section 3.

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 11, that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix A shows the collected grammar with all list operators expanded to standard ABNF notation.

As a convention, ABNF rule names prefixed with "obs-" denote "obsolete" grammar rules that appear for historical reasons.

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), HTAB (horizontal tab), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character). [[CREF1: Range also uses CHAR, which is probably a bug.]]

The rules below are defined in [Messaging]:

```
  obs-fold         = <obs-fold, see [Messaging], Section 5.2>
  protocol-name    = <protocol-name, see [Messaging], Section 9.7>
  protocol-version = <protocol-version, see [Messaging], Section 9.7>
  request-target   = <request-target, see [Messaging], Section 3.2>
```

This specification uses the terms "character", "character encoding scheme", "charset", and "protocol element" as they are defined in [RFC6365].

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 (Section 2 of [Messaging]) across a reliable transport- or session-layer "connection" (Section 9 of [Messaging]). 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 [RFC3986] to indicate the target resource (Section 5.1) and relationships between resources.

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).

```
         request   >
    UA ======================================= O
                                <   response
```

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 (Section 3 of [Messaging]), followed by header fields containing request modifiers, client information, and representation metadata (Section 5 of [Messaging]), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 6 of [Messaging]).

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 (Section 4 of [Messaging]), possibly followed by header fields containing server information, resource metadata, and representation

metadata (Section 5 of [Messaging]), an empty line to indicate the end of the header section, and finally a message body containing the payload body (if any, Section 6 of [Messaging]).

A connection might be used for multiple request/response exchanges, as defined in Section 9.3 of [Messaging].

The following example illustrates a typical message exchange for a GET request (Section 7.3.1) 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. 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.

```
         >             >             >             >
    UA =========== A =========== B =========== C =========== O
               <             <             <             <
```

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, as described in Section 5.6.2.

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, [RFC5246]) 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.

For example, an "interception proxy" [RFC3040] (also commonly known as a "transparent proxy" [RFC1919] or "captive portal") differs from an HTTP proxy because it is not selected by the client. Instead, an interception proxy filters or redirects outgoing TCP port 80 packets (and occasionally other common port traffic). Interception proxies are commonly found on public network access points, as a means of enforcing account subscription prior to allowing use of non-local Internet services, and within corporate firewalls to enforce network usage policies.

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. Some non-standard HTTP extensions (e.g., [RFC4559]) have been known to violate this requirement, resulting in security and interoperability problems.

2.3. 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.

```
            >             >
       UA =========== A =========== B - - - - - - C - - - - - - O
                  <             <
```

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 [Caching].

There is a wide variety of architectures and configurations of caches deployed across the World Wide Web and inside large organizations. These include national hierarchies of proxy caches to save transoceanic bandwidth, collaborative systems that broadcast or multicast cache entries, archives of pre-fetched cache entries for use in off-line or high-latency environments, and so on.

2.4. Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) [RFC3986] are used throughout HTTP as the means for identifying resources (Section 2.5). URI references are used to target requests, indicate redirects, and define relationships.

The definitions of "URI-reference", "absolute-URI", "relative-part", "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>
  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 (Section 5.3).

2.5. 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), as described in Section 2.4.

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 (Section 7) and a few request-modifying header fields (Section 8). If there is a conflict between the method semantics and any semantic implied by the URI itself, as described in Section 7.2.1, the method semantics take precedence.

IANA maintains the registry of URI Schemes [BCP115] at <https://www.iana.org/assignments/uri-schemes/>. [[CREF2: Although requests might target any URI scheme, the following schemes are inherent to HTTP servers:]]

URI Scheme	Description	Reference
http	Hypertext Transfer Protocol	Section 2.5.1
https	Hypertext Transfer Protocol Secure	Section 2.5.2

2.5.1. 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 ([RFC0793]) 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 ([RFC3986], Section 3.2.2). 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, as defined in Section 3.5 of [RFC3986].

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 12.1 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 (Section 2 of [Messaging]) containing the URI's identifying data to the server. If the server responds to that request with a non-interim HTTP

response message, as described in Section 9, 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 ([RFC3986], Section 3.2.1) 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.

2.5.2. 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 ([RFC5246]).

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 [RFC6265], can allow information set by one service to impact communication with other services within a matching group of host domains.

The process for authoritative access to an "https" identified resource is defined in [RFC2818].

2.5.3. 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. Conformance

3.1. 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.

3.2. Role-based Requirements

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).

3.3. Parsing Elements

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.

HTTP does not have specific length limitations for many of its protocol elements because the lengths that might be appropriate will vary widely, depending on the deployment context and purpose of the implementation. Hence, interoperability between senders and recipients depends on shared expectations regarding what is a reasonable length for each protocol element. Furthermore, what is commonly understood to be a reasonable length for some protocol elements has changed over the course of the past two decades of HTTP use and is expected to continue changing in the future.

At a minimum, a recipient MUST be able to parse and process protocol element lengths that are at least as long as the values that it generates for those same protocol elements in other messages. For example, an origin server that publishes very long URI references to its own resources needs to be able to parse and process those same references when received as a request target.

3.4. Error Handling

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.

3.5. Protocol Versioning

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 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. [[CREF3: For example, the version "HTTP/1.1" is defined by the combined specifications of this document, "HTTP Caching" [Caching], and "HTTP/1.1 Messaging" [Messaging].]]

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).

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.

HTTP's major version number is incremented when an incompatible message syntax is introduced. The minor number is 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.

[[CREF4: When a major version of HTTP does not define any minor versions, the minor version "0" is implied and ought to be used when referring to that protocol within a protocol element that requires sending a minor version.]]

4. Message Abstraction

[[CREF5: Each major version of HTTP defines its own syntax for the inclusion of information in messages. Nevertheless, a common abstraction is that a message includes some form of envelope/framing, a potential set of named data fields, and a potential body. This section defines the abstraction for message fields as field-name and field-value pairs.]]

4.1. Field Names

Header fields are key:value pairs that can be used to communicate data about the message, its payload, the target resource, or the connection (i.e., control data).

The requirements for header field names are defined in [BCP90].

The field-name token labels the corresponding field-value as having the semantics defined by that header field. For example, the Date header field is defined in Section 10.1.1.2 as containing the origination timestamp for the message in which it appears.

```
  field-name     = token
```

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 are defined for all versions of HTTP. 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. Header field extensibility is discussed in Section 4.1.2.

The following field names are defined by this document:

Header Field Name	Protocol	Status	Reference
Accept	http	standard	Section 8.4.2
Accept-Charset	http	standard	Section 8.4.3
Accept-Encoding	http	standard	Section 8.4.4
Accept-Language	http	standard	Section 8.4.5
Accept-Ranges	http	standard	Section 10.4.1
Allow	http	standard	Section 10.4.2
Authorization	http	standard	Section 8.5.3
Content-Encoding	http	standard	Section 6.2.2
Content-Language	http	standard	Section 6.2.3
Content-Length	http	standard	Section 6.2.4
Content-Location	http	standard	Section 6.2.5
Content-Range	http	standard	Section 6.3.3
Content-Type	http	standard	Section 6.2.1
Date	http	standard	Section 10.1.1.2
ETag	http	standard	Section 10.2.3
Expect	http	standard	Section 8.1.1
From	http	standard	Section 8.6.1
Host	http	standard	Section 5.4
If-Match	http	standard	Section 8.2.3
If-Modified-Since	http	standard	Section 8.2.5
If-None-Match	http	standard	Section 8.2.4
If-Range	http	standard	Section 8.2.7
If-Unmodified-Since	http	standard	Section 8.2.6
Last-Modified	http	standard	Section 10.2.2
Location	http	standard	Section 10.1.2
Max-Forwards	http	standard	Section 8.1.2
Proxy-Authenticate	http	standard	Section 10.3.2
Proxy-Authorization	http	standard	Section 8.5.4
Range	http	standard	Section 8.3
Referer	http	standard	Section 8.6.2
Retry-After	http	standard	Section 10.1.3
Server	http	standard	Section 10.4.3
Trailer	http	standard	Section 4.4
User-Agent	http	standard	Section 8.6.3
Vary	http	standard	Section 10.1.4
Via	http	standard	Section 5.6.1
WWW-Authenticate	http	standard	Section 10.3.1

4.1.1. Field Name Registry

HTTP header fields are registered within the "Message Headers" registry located at <https://www.iana.org/assignments/message-headers>, as defined by [BCP90], with the protocol "http".

4.1.2. Field Extensibility

Header fields are fully extensible: there is no limit on the introduction of new field names, each presumably defining new semantics, nor on the number of header fields used in a given message. Existing fields are defined in each part of this specification and in many other specifications outside this document set.

New header fields can be defined such that, when they are understood by a recipient, they might override or enhance the interpretation of previously defined header fields, define preconditions on request evaluation, or refine the meaning of responses.

A proxy MUST forward unrecognized header fields unless the field-name is listed in the Connection header field (Section 9.1 of [Messaging]) or the proxy is specifically configured to block, or otherwise transform, such fields. Other recipients SHOULD ignore unrecognized header fields. These requirements allow HTTP's functionality to be enhanced without requiring prior update of deployed intermediaries.

All defined header fields ought to be registered with IANA in the "Message Headers" registry.

4.1.3. Considerations for New Fields

Authors of specifications defining new fields are advised to keep the name as short as practical and not to prefix the name with "X-" unless the header field will never be used on the Internet. (The "X-" prefix idiom has been extensively misused in practice; it was intended to only be used as a mechanism for avoiding name collisions inside proprietary software or intranet processing, since the prefix would ensure that private names never collide with a newly registered Internet name; see [BCP178] for further information).

Authors of specifications defining new header fields are advised to consider documenting:

- Whether the field is a single value or whether it can be a list (delimited by commas; see Section 5 of [Messaging]).

If it does not use the list syntax, document how to treat messages where the field occurs multiple times (a sensible default would be to ignore the field, but this might not always be the right choice).

Note that intermediaries and software libraries might combine multiple header field instances into a single one, despite the field's definition not allowing the list syntax. A robust format enables recipients to discover these situations (good example: "Content-Type", as the comma can only appear inside quoted strings; bad example: "Location", as a comma can occur inside a URI).

- Under what conditions the header field can be used; e.g., only in responses or requests, in all messages, only on responses to a particular request method, etc.

- Whether the field should be stored by origin servers that understand it upon a PUT request.

- Whether the field semantics are further refined by the context, such as by existing request methods or status codes.

- Whether it is appropriate to list the field-name in the Connection header field (i.e., if the header field is to be hop-by-hop; see Section 9.1 of [Messaging]).

- Under what conditions intermediaries are allowed to insert, delete, or modify the field's value.

- Whether it is appropriate to list the field-name in a Vary response header field (e.g., when the request header field is used by an origin server's content selection algorithm; see Section 10.1.4).

- Whether the header field is useful or allowable in trailers (see Section 7.1 of [Messaging]).

- Whether the header field ought to be preserved across redirects.

- Whether it introduces any additional security considerations, such as disclosure of privacy-related data.

4.2. Field Values

This specification does not use ABNF rules to define each "Field-Name: Field Value" pair, as was done in earlier editions. Instead, this specification uses ABNF rules that are named according to each registered field name, wherein the rule defines the valid grammar for that field's corresponding field values (i.e., after the field-value has been extracted by a generic field parser).

```
  field-value    = *( field-content / obs-fold )
  field-content  = field-vchar [ 1*( SP / HTAB ) field-vchar ]
  field-vchar    = VCHAR / obs-text
```

Historically, HTTP header field values could be extended over multiple lines by preceding each extra line with at least one space or horizontal tab (obs-fold). [[CREF6: This document assumes that any such obs-fold has been replaced with one or more SP octets prior to interpreting the field value, as described in Section 5.2 of [Messaging].]]

Historically, HTTP has allowed field content with text in the ISO-8859-1 charset [ISO-8859-1], supporting other charsets only through use of [RFC2047] encoding. In practice, most HTTP header field values use only a subset of the US-ASCII charset [USASCII]. Newly defined header fields SHOULD limit their field values to US-ASCII octets. A recipient SHOULD treat other octets in field content (obs-text) as opaque data.

4.2.1. Field Order

The order in which header fields with differing field names are received is not significant. However, it is good practice to send header fields that contain control data first, such as Host on requests and Date on responses, so that implementations can decide when not to handle a message as early as possible. A server MUST NOT apply a request to the target resource until the entire request header section is received, since later header fields might include conditionals, authentication credentials, or deliberately misleading duplicate header fields that would impact request processing.

A sender MUST NOT generate multiple header fields with the same field name in a message unless either the entire field value for that header field is defined as a comma-separated list [i.e., #(values)] or the header field is a well-known exception (as noted below).

A recipient MAY combine multiple header fields with the same field name into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field value to

the combined field value in order, separated by a comma. The order in which header fields with the same field name are received is therefore significant to the interpretation of the combined field value; a proxy MUST NOT change the order of these field values when forwarding a message.

> Note: In practice, the "Set-Cookie" header field ([RFC6265]) often appears multiple times in a response message and does not use the list syntax, violating the above requirements on multiple header fields with the same name. Since it cannot be combined into a single field-value, recipients ought to handle "Set-Cookie" as a special case while processing header fields. (See Appendix A.2.3 of [Kri2001] for details.)

4.2.2. Field Limits

HTTP does not place a predefined limit on the length of each header field or on the length of the header section as a whole, as described in Section 3. Various ad hoc limitations on individual header field length are found in practice, often depending on the specific field semantics.

A server that receives a request header field, or set of fields, larger than it wishes to process MUST respond with an appropriate 4xx (Client Error) status code. Ignoring such header fields would increase the server's vulnerability to request smuggling attacks (Section 11.2 of [Messaging]).

A client MAY discard or truncate received header fields that are larger than the client wishes to process if the field semantics are such that the dropped value(s) can be safely ignored without changing the message framing or response semantics.

4.2.3. Field Value Components

Most HTTP header field values are defined using common syntax components (token, quoted-string, and comment) separated by whitespace or specific delimiting characters. Delimiters are chosen from the set of US-ASCII visual characters not allowed in a token (DQUOTE and "(),/:;<=>?@[\]{}").

```
  token          = 1*tchar

  tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
                 / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
                 / DIGIT / ALPHA
                 ; any VCHAR, except delimiters
```

A string of text is parsed as a single value if it is quoted using double-quote marks.

```
  quoted-string  = DQUOTE *( qdtext / quoted-pair ) DQUOTE
  qdtext         = HTAB / SP /%x21 / %x23-5B / %x5D-7E / obs-text
  obs-text       = %x80-FF
```

Comments can be included in some HTTP header fields by surrounding the comment text with parentheses. Comments are only allowed in fields containing "comment" as part of their field value definition.

```
  comment        = "(" *( ctext / quoted-pair / comment ) ")"
  ctext          = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-text
```

The backslash octet ("\") can be used as a single-octet quoting mechanism within quoted-string and comment constructs. Recipients that process the value of a quoted-string MUST handle a quoted-pair as if it were replaced by the octet following the backslash.

```
  quoted-pair    = "\" ( HTAB / SP / VCHAR / obs-text )
```

A sender SHOULD NOT generate a quoted-pair in a quoted-string except where necessary to quote DQUOTE and backslash octets occurring within that string. A sender SHOULD NOT generate a quoted-pair in a comment except where necessary to quote parentheses ["(" and ")"] and backslash octets occurring within that comment.

4.2.4. Designing New Field Values

New header field values typically have their syntax defined using ABNF ([RFC5234]), using the extension defined in Section 11 as necessary, and are usually constrained to the range of US-ASCII characters. Header fields needing a greater range of characters can use an encoding such as the one defined in [RFC5987].

Leading and trailing whitespace in raw field values is removed upon field parsing (Section 5.1 of [Messaging]). Field definitions where leading or trailing whitespace in values is significant will have to use a container syntax such as quoted-string (Section 4.2.3).

Because commas (",") are used as a generic delimiter between field-values, they need to be treated with care if they are allowed in the field-value. Typically, components that might contain a comma are protected with double-quotes using the quoted-string ABNF production.

For example, a textual date and a URI (either of which might contain a comma) could be safely carried in field-values like these:

```
  Example-URI-Field: "http://example.com/a.html,foo",
                     "http://without-a-comma.example.com/"
  Example-Date-Field: "Sat, 04 May 1996", "Wed, 14 Sep 2005"
```

Note that double-quote delimiters almost always are used with the quoted-string production; using a different syntax inside double-quotes will likely cause unnecessary confusion.

Many header fields use a format including (case-insensitively) named parameters (for instance, Content-Type, defined in Section 6.2.1). Allowing both unquoted (token) and quoted (quoted-string) syntax for the parameter value enables recipients to use existing parser components. When allowing both forms, the meaning of a parameter value ought to be independent of the syntax used for it (for an example, see the notes on parameter handling for media types in Section 6.1.1).

4.3. Whitespace

This specification uses three rules to denote the use of linear whitespace: OWS (optional whitespace), RWS (required whitespace), and BWS ("bad" whitespace).

The OWS rule is used where zero or more linear whitespace octets might appear. For protocol elements where optional whitespace is preferred to improve readability, a sender SHOULD generate the optional whitespace as a single SP; otherwise, a sender SHOULD NOT generate optional whitespace except as needed to white out invalid or unwanted protocol elements during in-place message filtering.

The RWS rule is used when at least one linear whitespace octet is required to separate field tokens. A sender SHOULD generate RWS as a single SP.

The BWS rule is used where the grammar allows optional whitespace only for historical reasons. A sender MUST NOT generate BWS in messages. A recipient MUST parse for such bad whitespace and remove it before interpreting the protocol element.

```
  OWS            = *( SP / HTAB )
                 ; optional whitespace
  RWS            = 1*( SP / HTAB )
                 ; required whitespace
  BWS            = OWS
                 ; "bad" whitespace
```

4.4. Trailer

[[CREF7: The "Trailer" header field in a message indicates fields that the sender anticipates sending after the message header block (i.e., during or after the payload is sent). This is typically used to supply metadata that might be dynamically generated while the data is sent, such as a message integrity check, digital signature, or post-processing status.]]

```
  Trailer = 1#field-name
```

[[CREF8: How, where, and when trailer fields might be sent depends on both the protocol in use (HTTP version and/or transfer coding) and the semantics of each named header field. Many header fields cannot be processed outside the header section because their evaluation is necessary for message routing, authentication, or configuration prior to receiving the representation data.]]

5. Message Routing

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

HTTP is used in a wide variety of applications, ranging from general-purpose computers to home appliances. In some cases, communication options are hard-coded in a client's configuration. However, most HTTP clients rely on the same resource identification mechanism and configuration techniques as general-purpose Web browsers.

HTTP communication is initiated by a user agent for some purpose. The purpose is a combination of request semantics and a target resource upon which to apply those semantics. A URI reference (Section 2.4) is typically used as an identifier for the "target resource", which a user agent would resolve to its absolute form in order to obtain the "target URI". The target URI excludes the reference's fragment component, if any, since fragment identifiers are reserved for client-side processing ([RFC3986], Section 3.5).

5.2. Routing Inbound

Once the target URI is determined, a client needs to decide whether a network request is necessary to accomplish the desired semantics and, if so, where that request is to be directed.

If the client has a cache [Caching] and the request can be satisfied by it, then the request is usually directed there first.

If the request is not satisfied by a cache, then a typical client will check its configuration to determine whether a proxy is to be used to satisfy the request. Proxy configuration is implementation-dependent, but is often based on URI prefix matching, selective authority matching, or both, and the proxy itself is usually identified by an "http" or "https" URI. If a proxy is applicable, the client connects inbound by establishing (or reusing) a connection to that proxy.

If no proxy is applicable, a typical client will invoke a handler routine, usually specific to the target URI's scheme, to connect directly to an authority for the target resource. How that is accomplished is dependent on the target URI scheme and defined by its associated specification, similar to how this specification defines origin server access for resolution of the "http" (Section 2.5.1) and "https" (Section 2.5.2) schemes.

HTTP requirements regarding connection management are defined in Section 9 of [Messaging].

5.3. Effective Request URI

Once an inbound connection is obtained, the client sends an HTTP request message (Section 2 of [Messaging]).

[[CREF9: Depending on the nature of the request, the client's target URI might be split into components and transmitted (or implied) within various parts of a request message. These parts are recombined by each recipient, in accordance with their local configuration and incoming connection context, to form an "effective request URI" for identifying the intended target resource with respect to that server. Section 3.3 of [Messaging] defines how a server determines the effective request URI for an HTTP/1.1 request.]]

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

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 12 for security considerations regarding message routing.

5.4. Host

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 2.4
```

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 2.5.1). 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.

5.5. Associating a Response to a Request

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 9.2) 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.

5.6. Message Forwarding

As described in Section 2.2, 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 9.1 of [Messaging], 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.

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.

5.6.1. Via

The "Via" header field indicates the presence of intermediate protocols and recipients between the user agent and the server (on requests) or between the origin server and the client (on responses), similar to the "Received" header field in email (Section 3.6.7 of [RFC5322]). Via can be used for tracking message forwards, avoiding request loops, and identifying the protocol capabilities of senders along the request/response chain.

```
  Via = 1#( received-protocol RWS received-by [ RWS comment ] )

  received-protocol = [ protocol-name "/" ] protocol-version
                      ; see [Messaging], Section 9.7
  received-by       = ( uri-host [ ":" port ] ) / pseudonym
  pseudonym         = token
```

Multiple Via field values represent each proxy or gateway that has forwarded the message. Each intermediary appends its own information about how the message was received, such that the end result is ordered according to the sequence of forwarding recipients.

A proxy MUST send an appropriate Via header field, as described below, in each message that it forwards. An HTTP-to-HTTP gateway MUST send an appropriate Via header field in each inbound request message and MAY send a Via header field in forwarded response messages.

For each intermediary, the received-protocol indicates the protocol and protocol version used by the upstream sender of the message. Hence, the Via field value records the advertised protocol capabilities of the request/response chain such that they remain visible to downstream recipients; this can be useful for determining what backwards-incompatible features might be safe to use in response, or within a later request, as described in Section 3.5. For brevity, the protocol-name is omitted when the received protocol is HTTP.

The received-by portion of the field value is normally the host and optional port number of a recipient server or client that subsequently forwarded the message. However, if the real host is considered to be sensitive information, a sender MAY replace it with

a pseudonym. If a port is not provided, a recipient MAY interpret that as meaning it was received on the default TCP port, if any, for the received-protocol.

A sender MAY generate comments in the Via header field to identify the software of each recipient, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional, and a recipient MAY remove them prior to forwarding the message.

For example, a request message could be sent from an HTTP/1.0 user agent to an internal proxy code-named "fred", which uses HTTP/1.1 to forward the request to a public proxy at p.example.net, which completes the request by forwarding it to the origin server at www.example.com. The request received by www.example.com would then have the following Via header field:

```
  Via: 1.0 fred, 1.1 p.example.net
```

An intermediary used as a portal through a network firewall SHOULD NOT forward the names and ports of hosts within the firewall region unless it is explicitly enabled to do so. If not enabled, such an intermediary SHOULD replace each received-by host of any host behind the firewall by an appropriate pseudonym for that host.

An intermediary MAY combine an ordered subsequence of Via header field entries into a single such entry if the entries have identical received-protocol values. For example,

```
  Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy
```

could be collapsed to

```
  Via: 1.0 ricky, 1.1 mertz, 1.0 lucy
```

A sender SHOULD NOT combine multiple entries unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. A sender MUST NOT combine entries that have different received-protocol values.

5.6.2. Transformations

Some intermediaries include features for transforming messages and their payloads. A proxy might, for example, convert between image formats in order to save cache space or to reduce the amount of traffic on a slow link. However, operational problems might occur when these transformations are applied to payloads intended for critical applications, such as medical imaging or scientific data

analysis, particularly when integrity checks or digital signatures are used to ensure that the payload received is identical to the original.

An HTTP-to-HTTP proxy is called a "transforming proxy" if it is designed or configured to modify messages in a semantically meaningful way (i.e., modifications, beyond those required by normal HTTP processing, that change the message in a way that would be significant to the original sender or potentially significant to downstream recipients). For example, a transforming proxy might be acting as a shared annotation server (modifying responses to include references to a local annotation database), a malware filter, a format transcoder, or a privacy filter. Such transformations are presumed to be desired by whichever client (or client organization) selected the proxy.

If a proxy receives a request-target with a host name that is not a fully qualified domain name, it MAY add its own domain to the host name it received when forwarding the request. A proxy MUST NOT change the host name if the request-target contains a fully qualified domain name.

A proxy MUST NOT modify the "absolute-path" and "query" parts of the received request-target when forwarding it to the next inbound server, except as noted above to replace an empty path with "/" or "*".

A proxy MAY modify the message body through application or removal of a transfer coding (Section 7 of [Messaging]).

A proxy MUST NOT transform the payload (Section 6.3) of a message that contains a no-transform cache-control directive (Section 5.2 of [Caching]).

A proxy MAY transform the payload of a message that does not contain a no-transform cache-control directive. A proxy that transforms a payload MUST add a Warning header field with the warn-code of 214 ("Transformation Applied") if one is not already in the message (see Section 5.5 of [Caching]). A proxy that transforms the payload of a 200 (OK) response can further inform downstream recipients that a transformation has been applied by changing the response status code to 203 (Non-Authoritative Information) (Section 9.3.4).

A proxy SHOULD NOT modify header fields that provide information about the endpoints of the communication chain, the resource state, or the selected representation (other than the payload) unless the field's definition specifically allows such modification or the modification is deemed necessary for privacy or security.

6. 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 Section 8.2 and constructing the payload for 200 (OK) and 304 (Not Modified) responses to GET (Section 7.3.1).

6.1. 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 ) )
```

6.1.1. Media Type

HTTP uses media types [RFC2046] in the Content-Type (Section 6.2.1) and Accept (Section 8.4.2) 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"
```

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.

6.1.1.1. Charset

HTTP uses charset names to indicate or negotiate the character encoding scheme of a textual representation [RFC6365]. A charset is identified by a case-insensitive token.

```
  charset = token
```

Charset names ought to be registered in the IANA "Character Sets" registry (<https://www.iana.org/assignments/character-sets>) according to the procedures defined in [RFC2978].

6.1.1.2. Canonicalization and Text Defaults

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]. However, the performance characteristics of email deployments (i.e., store and forward messages to peers) are significantly different from those common to HTTP and the Web (server-based information services). Furthermore, MIME's constraints for the sake of compatibility with older mail transfer protocols do not apply to HTTP (see Appendix B of [Messaging]).

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.

6.1.1.3. 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 Section 9.3.7.

6.1.2. 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
```

Content-coding values are used in the Accept-Encoding (Section 8.4.4) and Content-Encoding (Section 6.2.2) header fields.

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

Name	Description	Reference
compress	UNIX "compress" data format [Welch]	Section 6.1.2.1
deflate	"deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950])	Section 6.1.2.2
gzip	GZIP file format [RFC1952]	Section 6.1.2.3
identity	Reserved (synonym for "no encoding" in Accept-Encoding)	Section 8.4.4
x-compress	Deprecated (alias for compress)	Section 6.1.2.1
x-gzip	Deprecated (alias for gzip)	Section 6.1.2.3

6.1.2.1. Compress Coding

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".

6.1.2.2. Deflate Coding

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.

6.1.2.3. Gzip Coding

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".

6.1.2.4. Content Coding Extensibility

Additional content codings, outside the scope of this specification, have been specified for use in HTTP. All such content codings ought to be registered within the "HTTP Content Coding Registry".

6.1.2.4.1. Content Coding Registry

The "HTTP Content Coding Registry", maintained by IANA at <https://www.iana.org/assignments/http-parameters/>, registers content-coding names.

Content coding registrations MUST include the following fields:

- Name
- Description
- Pointer to specification text

Names of content codings MUST NOT overlap with names of transfer codings (Section 7 of [Messaging]), unless the encoding transformation is identical (as is the case for the compression codings defined in Section 6.1.2).

Values to be added to this namespace require IETF Review (see Section 4.1 of [RFC5226]) and MUST conform to the purpose of content coding defined in Section 6.1.2.

6.1.3. Language Tags

A language tag, as defined in [RFC5646], 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 defined in Section 8.4.5, 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.

6.1.4. Range Units

A representation can be partitioned into subranges according to various structural units, depending on the structure inherent in the representation's media type. This "range unit" is used in the Accept-Ranges (Section 10.4.1) response header field to advertise support for range requests, the Range (Section 8.3) request header field to delineate the parts of a representation that are requested, and the Content-Range (Section 6.3.3) payload header field to describe which part of a representation is being transferred.

```
  range-unit       = bytes-unit / other-range-unit
```

The following range unit names are defined by this document:

Range Unit Name	Description	Reference
bytes	a range of octets	Section 6.1.4.1
none	reserved as keyword, indicating no ranges are supported	Section 10.4.1

6.1.4.1. Byte Ranges

Since representation data is transferred in payloads as a sequence of octets, a byte range is a meaningful substructure for any representation transferable over HTTP (Section 6). The "bytes" range

unit is defined for expressing subranges of the data's octet sequence.

```
  bytes-unit       = "bytes"
```

A byte-range request can specify a single range of bytes or a set of ranges within a single representation.

```
  byte-ranges-specifier = bytes-unit "=" byte-range-set
  byte-range-set  = 1#( byte-range-spec / suffix-byte-range-spec )
  byte-range-spec = first-byte-pos "-" [ last-byte-pos ]
  first-byte-pos  = 1*DIGIT
  last-byte-pos   = 1*DIGIT
```

The first-byte-pos value in a byte-range-spec gives the byte-offset of the first byte in a range. The last-byte-pos value gives the byte-offset of the last byte in the range; that is, the byte positions specified are inclusive. Byte offsets start at zero.

Examples of byte-ranges-specifier values:

- The first 500 bytes (byte offsets 0-499, inclusive):

```
     bytes=0-499
```

- The second 500 bytes (byte offsets 500-999, inclusive):

```
     bytes=500-999
```

A byte-range-spec is invalid if the last-byte-pos value is present and less than the first-byte-pos.

A client can limit the number of bytes requested without knowing the size of the selected representation. If the last-byte-pos value is absent, or if the value is greater than or equal to the current length of the representation data, the byte range is interpreted as the remainder of the representation (i.e., the server replaces the value of last-byte-pos with a value that is one less than the current length of the selected representation).

A client can request the last N bytes of the selected representation using a suffix-byte-range-spec.

```
  suffix-byte-range-spec = "-" suffix-length
  suffix-length = 1*DIGIT
```

If the selected representation is shorter than the specified suffix-length, the entire representation is used.

Additional examples, assuming a representation of length 10000:

- The final 500 bytes (byte offsets 9500-9999, inclusive):

```
bytes=-500
```

  Or:

```
bytes=9500-
```

- The first and last bytes only (bytes 0 and 9999):

```
bytes=0-0,-1
```

- Other valid (but not canonical) specifications of the second 500 bytes (byte offsets 500-999, inclusive):

```
bytes=500-600,601-999
bytes=500-700,601-999
```

If a valid byte-range-set includes at least one byte-range-spec with a first-byte-pos that is less than the current length of the representation, or at least one suffix-byte-range-spec with a non-zero suffix-length, then the byte-range-set is satisfiable. Otherwise, the byte-range-set is unsatisfiable.

In the byte-range syntax, first-byte-pos, last-byte-pos, and suffix-length are expressed as decimal number of octets. Since there is no predefined limit to the length of a payload, recipients MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows.

6.1.4.2. Other Range Units

Range units are intended to be extensible. New range units ought to be registered with IANA, as defined in Section 6.1.4.3.

```
other-range-unit = token
```

6.1.4.3. Range Unit Registry

The "HTTP Range Unit Registry" defines the namespace for the range unit names and refers to their corresponding specifications. It is maintained at <https://www.iana.org/assignments/http-parameters>.

Registration of an HTTP Range Unit MUST include the following fields:

- Name

- Description

- Pointer to specification text

Values to be added to this namespace require IETF Review (see [RFC5226], Section 4.1).

6.2. 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:

Header Field Name	Defined in...
Content-Type	Section 6.2.1
Content-Encoding	Section 6.2.2
Content-Language	Section 6.2.3
Content-Length	Section 6.2.4
Content-Location	Section 6.2.5

6.2.1. Content-Type

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 6.1.1. 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.

6.2.2. Content-Encoding

The "Content-Encoding" header field indicates what content codings have been applied to the representation, beyond those inherent in the media type, and thus what decoding mechanisms have to be applied in order to obtain data in the media type referenced by the Content-Type header field. Content-Encoding is primarily used to allow a representation's data to be compressed without losing the identity of its underlying media type.

```
  Content-Encoding = 1#content-coding
```

An example of its use is

```
  Content-Encoding: gzip
```

If one or more encodings have been applied to a representation, the sender that applied the encodings MUST generate a Content-Encoding header field that lists the content codings in the order in which they were applied. Additional information about the encoding parameters can be provided by other header fields not defined by this specification.

Unlike Transfer-Encoding (Section 6.1 of [Messaging]), the codings listed in Content-Encoding are a characteristic of the representation; the representation is defined in terms of the coded form, and all other metadata about the representation is about the coded form unless otherwise noted in the metadata definition. Typically, the representation is only decoded just prior to rendering or analogous usage.

If the media type includes an inherent encoding, such as a data format that is always compressed, then that encoding would not be

restated in Content-Encoding even if it happens to be the same algorithm as one of the content codings. Such a content coding would only be listed if, for some bizarre reason, it is applied a second time to form the representation. Likewise, an origin server might choose to publish the same data as multiple representations that differ only in whether the coding is defined as part of Content-Type or Content-Encoding, since some user agents will behave differently in their handling of each response (e.g., open a "Save as ..." dialog instead of automatic decompression and rendering of content).

An origin server MAY respond with a status code of 415 (Unsupported Media Type) if a representation in the request message has a content coding that is not acceptable.

6.2.3. Content-Language

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
```

Language tags are defined in Section 6.1.3. 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.

6.2.4. Content-Length

[[CREF10: The "Content-Length" header field indicates the number of data octets (body length) for the representation. In some cases, Content-Length is used to define or estimate message framing.]]

```
  Content-Length = 1*DIGIT
```

An example is

```
  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 (indicating an empty payload body). 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 (Section 7.3.2); 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.

A server MAY send a Content-Length header field in a 304 (Not Modified) response to a conditional GET request (Section 9.4.5); 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 (Section 7.3.6).

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 (Section 12.5).

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.

6.2.5. 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 (Section 5.3). It is representation metadata. It has the same syntax and semantics as the header field of the same name defined for MIME body parts in Section 4 of [RFC2557]. However, its appearance in an HTTP message has some special implications for HTTP recipients.

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 (Section 7.3.1) or HEAD (Section 7.3.2) 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 (Section 7.3.4) or POST (Section 7.3.3), 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.

- 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.

- 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.

- 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.

6.3. Payload

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).

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.

Header Field Name	Defined in...
Content-Range	Section 6.3.3
Trailer	Section 4.4
Transfer-Encoding	Section 6.1 of [Messaging]

6.3.1. Purpose

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 (Section 7.3.4) represents the desired state of the target resource if the request is successfully applied, whereas a representation in the payload of a POST request (Section 7.3.3) 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 (Section 7.3.1) represents the current state of the target resource, as observed at the time of the message origination date (Section 10.1.1.2), 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.

6.3.2. Identification

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:

- 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.

- 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 (Section 5.3).

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.

6.3.3. Content-Range

The "Content-Range" header field is sent in a single part 206 (Partial Content) response to indicate the partial range of the selected representation enclosed as the message payload, sent in each part of a multipart 206 response to indicate the range enclosed within each body part, and sent in 416 (Range Not Satisfiable) responses to provide information about the selected representation.

```
  Content-Range       = byte-content-range
                      / other-content-range

  byte-content-range  = bytes-unit SP
                        ( byte-range-resp / unsatisfied-range )

  byte-range-resp     = byte-range "/" ( complete-length / "*" )
  byte-range          = first-byte-pos "-" last-byte-pos
  unsatisfied-range   = "*/" complete-length

  complete-length     = 1*DIGIT

  other-content-range = other-range-unit SP other-range-resp
  other-range-resp    = *CHAR
```

If a 206 (Partial Content) response contains a Content-Range header field with a range unit (Section 6.1.4) that the recipient does not understand, the recipient MUST NOT attempt to recombine it with a stored representation. A proxy that receives such a message SHOULD forward it downstream.

For byte ranges, a sender SHOULD indicate the complete length of the representation from which the range has been extracted, unless the complete length is unknown or difficult to determine. An asterisk character ("*") in place of the complete-length indicates that the representation length was unknown when the header field was generated.

The following example illustrates when the complete length of the selected representation is known by the sender to be 1234 bytes:

```
  Content-Range: bytes 42-1233/1234
```

and this second example illustrates when the complete length is unknown:

```
Content-Range: bytes 42-1233/*
```

A Content-Range field value is invalid if it contains a byte-range-resp that has a last-byte-pos value less than its first-byte-pos value, or a complete-length value less than or equal to its last-byte-pos value. The recipient of an invalid Content-Range MUST NOT attempt to recombine the received content with a stored representation.

A server generating a 416 (Range Not Satisfiable) response to a byte-range request SHOULD send a Content-Range header field with an unsatisfied-range value, as in the following example:

```
Content-Range: bytes */1234
```

The complete-length in a 416 response indicates the current length of the selected representation.

The Content-Range header field has no meaning for status codes that do not explicitly describe its semantic. For this specification, only the 206 (Partial Content) and 416 (Range Not Satisfiable) status codes describe a meaning for Content-Range.

The following are examples of Content-Range values in which the selected representation contains a total of 1234 bytes:

- The first 500 bytes:

  ```
  Content-Range: bytes 0-499/1234
  ```

- The second 500 bytes:

  ```
  Content-Range: bytes 500-999/1234
  ```

- All except for the first 500 bytes:

  ```
  Content-Range: bytes 500-1233/1234
  ```

- The last 500 bytes:

  ```
  Content-Range: bytes 734-1233/1234
  ```

6.3.4. Media Type multipart/byteranges

When a 206 (Partial Content) response message includes the content of multiple ranges, they are transmitted as body parts in a multipart message body ([RFC2046], Section 5.1) with the media type of "multipart/byteranges".

The multipart/byteranges media type includes one or more body parts, each with its own Content-Type and Content-Range fields. The required boundary parameter specifies the boundary string used to separate each body part.

Implementation Notes:

1. Additional CRLFs might precede the first boundary string in the body.

2. Although [RFC2046] permits the boundary string to be quoted, some existing implementations handle a quoted boundary string incorrectly.

3. A number of clients and servers were coded to an early draft of the byteranges specification that used a media type of multipart/x-byteranges, which is almost (but not quite) compatible with this type.

Despite the name, the "multipart/byteranges" media type is not limited to byte ranges. The following example uses an "exampleunit" range unit:

```
HTTP/1.1 206 Partial Content
Date: Tue, 14 Nov 1995 06:25:24 GMT
Last-Modified: Tue, 14 July 04:58:08 GMT
Content-Length: 2331785
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES

--THIS_STRING_SEPARATES
Content-Type: video/example
Content-Range: exampleunit 1.2-4.3/25

...the first range...
--THIS_STRING_SEPARATES
Content-Type: video/example
Content-Range: exampleunit 11.2-14.3/25

...the second range
--THIS_STRING_SEPARATES--
```

The following information serves as the registration form for the multipart/byteranges media type.

Type name: multipart

Subtype name: byteranges

Required parameters: boundary

Optional parameters: N/A

Encoding considerations: only "7bit", "8bit", or "binary" are permitted

Security considerations: see Section 12

Interoperability considerations: N/A

Published specification: This specification (see Section 6.3.4).

Applications that use this media type: HTTP components supporting multiple ranges in a single request.

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: See Aut hors' Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors' Addresses section.

Change controller: IESG

6.4. 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.

6.4.1. 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 8.4 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:

- 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?);

- 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;

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

- 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 (Section 10.1.4) is often sent in a response subject to proactive negotiation to indicate what parts of the request information were used in the selection algorithm.

6.4.2. 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.

7. Request Methods

7.1. 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 8) if those additional semantics do not conflict with the method. For example, a client can send conditional request header fields (Section 8.2) to make the requested action conditional on the current state of the target resource.

```
  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.	7.3.1
HEAD	Same as GET, but only transfer the status line and header section.	7.3.2
POST	Perform resource-specific processing on the request payload.	7.3.3
PUT	Replace all current representations of the target resource with the request payload.	7.3.4
DELETE	Remove all current representations of the target resource.	7.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	7.3.6
OPTIONS	Describe the communication options for the target resource.	7.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	7.3.8

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

The set of methods allowed by a target resource can be listed in an Allow header field (Section 10.4.2). 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.

7.2. Common Method Properties

Method	Safe	Idempotent	Reference
CONNECT	no	no	Section 7.3.6
DELETE	no	yes	Section 7.3.5
GET	yes	yes	Section 7.3.1
HEAD	yes	yes	Section 7.3.2
OPTIONS	yes	yes	Section 7.3.7
POST	no	no	Section 7.3.3
PUT	no	yes	Section 7.3.4
TRACE	yes	yes	Section 7.3.8

7.2.1. 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.

7.2.2. 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.

7.2.3. 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 [Caching]. 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.

7.3. Method Definitions

7.3.1. 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 (see Section 12.3 for related security considerations). 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 (Section 8.3).

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 [Caching]).

7.3.2. 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 6.3) 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 [Caching]). A HEAD response might also have an effect on previously cached responses to GET; see Section 4.3.5 of [Caching].

7.3.3. 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):

- Providing a block of data, such as the fields entered into an HTML form, to a data-handling process;
- Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles;
- Creating a new resource that has yet to be identified by the origin server; and
- 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 10.1.2) 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 [Caching]). 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 6.2.5).

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.

7.3.4. 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:

a. reconfigure the target resource to reflect the new media type;

b. transform the PUT representation to a format consistent with that of the resource before saving it as the new resource state; or,

c. 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 10.2), 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 8.2).

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 6.3.3), 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 [Caching]).

7.3.5. 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 [Caching]).

7.3.6. 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 3.2 of [Messaging]); 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.

7.3.7. 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 (Section 3.2 of [Messaging]) 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 8.1.2). 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.

7.3.8. 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 10.1 of [Messaging]). 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 8.1.2).

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 Section 8.5 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 5.6.1) 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.

7.4. Method Extensibility

Additional methods, outside the scope of this specification, have been specified for use in HTTP. All such methods ought to be registered within the "Hypertext Transfer Protocol (HTTP) Method Registry".

7.4.1. Method Registry

The "Hypertext Transfer Protocol (HTTP) Method Registry", maintained by IANA at <https://www.iana.org/assignments/http-methods>, registers method names.

HTTP method registrations MUST include the following fields:

- Method Name (see Section 7)

- Safe ("yes" or "no", see Section 7.2.1)

- Idempotent ("yes" or "no", see Section 7.2.2)

- Pointer to specification text

Values to be added to this namespace require IETF Review (see [RFC5226], Section 4.1).

7.4.2. Considerations for New Methods

Standardized methods are generic; that is, they are potentially applicable to any resource, not just one particular media type, kind of resource, or application. As such, it is preferred that new methods be registered in a document that isn't specific to a single application or data format, since orthogonal technologies deserve orthogonal specification.

Since message parsing (Section 6 of [Messaging]) needs to be independent of method semantics (aside from responses to HEAD), definitions of new methods cannot change the parsing algorithm or prohibit the presence of a message body on either the request or the response message. Definitions of new methods can specify that only a zero-length message body is allowed by requiring a Content-Length header field with a value of "0".

A new method definition needs to indicate whether it is safe (Section 7.2.1), idempotent (Section 7.2.2), cacheable (Section 7.2.3), what semantics are to be associated with the payload

body if any is present in the request and what refinements the method makes to header field or status code semantics. If the new method is cacheable, its definition ought to describe how, and under what conditions, a cache can store a response and use it to satisfy a subsequent request. The new method ought to describe whether it can be made conditional (Section 8.2) and, if so, how a server responds when the condition is false. Likewise, if the new method might have some use for partial response semantics (Section 8.3), it ought to document this, too.

> Note: Avoid defining a method name that starts with "M-", since that prefix might be misinterpreted as having the semantics assigned to it by [RFC2774].

8. 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.

8.1. Controls

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

Header Field Name	Defined in...
Cache-Control	Section 5.2 of [Caching]
Expect	Section 8.1.1
Host	Section 5.4
Max-Forwards	Section 8.1.2
Pragma	Section 5.4 of [Caching]
TE	Section 7.4 of [Messaging]

8.1.1. Expect

The "Expect" header field in a request indicates a certain set of behaviors (expectations) that need to be supported by the server in order to properly handle this request. The only such expectation defined by this specification is 100-continue.

```
  Expect  = "100-continue"
```

The Expect field-value is case-insensitive.

A server that receives an Expect field-value other than 100-continue MAY respond with a 417 (Expectation Failed) status code to indicate that the unexpected expectation cannot be met.

A 100-continue expectation informs recipients that the client is about to send a (presumably large) message body in this request and wishes to receive a 100 (Continue) interim response if the request-line and header fields are not sufficient to cause an immediate success, redirect, or error response. This allows the client to wait for an indication that it is worthwhile to send the message body before actually doing so, which can improve efficiency when the message body is huge or when the client anticipates that an error is likely (e.g., when sending a state-changing method, for the first time, without previously verified authentication credentials).

For example, a request that begins with

```
PUT /somewhere/fun HTTP/1.1
Host: origin.example.com
Content-Type: video/h264
Content-Length: 1234567890987
Expect: 100-continue
```

allows the origin server to immediately respond with an error message, such as 401 (Unauthorized) or 405 (Method Not Allowed), before the client starts filling the pipes with an unnecessary data transfer.

Requirements for clients:

- A client MUST NOT generate a 100-continue expectation in a request that does not include a message body.

- A client that will wait for a 100 (Continue) response before sending the request message body MUST send an Expect header field containing a 100-continue expectation.

- A client that sends a 100-continue expectation is not required to wait for any specific length of time; such a client MAY proceed to send the message body even if it has not yet received a response. Furthermore, since 100 (Continue) responses cannot be sent through an HTTP/1.0 intermediary, such a client SHOULD NOT wait for an indefinite period before sending the message body.

- A client that receives a 417 (Expectation Failed) status code in response to a request containing a 100-continue expectation SHOULD repeat that request without a 100-continue expectation, since the 417 response merely indicates that the response chain does not support expectations (e.g., it passes through an HTTP/1.0 server).

Requirements for servers:

- A server that receives a 100-continue expectation in an HTTP/1.0 request MUST ignore that expectation.

- A server MAY omit sending a 100 (Continue) response if it has already received some or all of the message body for the corresponding request, or if the framing indicates that there is no message body.

- A server that sends a 100 (Continue) response MUST ultimately send a final status code, once the message body is received and processed, unless the connection is closed prematurely.

- A server that responds with a final status code before reading the entire message body SHOULD indicate in that response whether it intends to close the connection or continue reading and discarding the request message (see Section 9.6 of [Messaging]).

An origin server MUST, upon receiving an HTTP/1.1 (or later) request-line and a complete header section that contains a 100-continue expectation and indicates a request message body will follow, either send an immediate response with a final status code, if that status can be determined by examining just the request-line and header fields, or send an immediate 100 (Continue) response to encourage the client to send the request's message body. The origin server MUST NOT wait for the message body before sending the 100 (Continue) response.

A proxy MUST, upon receiving an HTTP/1.1 (or later) request-line and a complete header section that contains a 100-continue expectation and indicates a request message body will follow, either send an immediate response with a final status code, if that status can be determined by examining just the request-line and header fields, or begin forwarding the request toward the origin server by sending a corresponding request-line and header section to the next inbound server. If the proxy believes (from configuration or past interaction) that the next inbound server only supports HTTP/1.0, the proxy MAY generate an immediate 100 (Continue) response to encourage the client to begin sending the message body.

Note: The Expect header field was added after the original publication of HTTP/1.1 [RFC2068] as both the means to request an interim 100 (Continue) response and the general mechanism for indicating must-understand extensions. However, the extension mechanism has not been used by clients and the must-understand requirements have not been implemented by many servers, rendering the extension mechanism useless. This specification has removed the extension mechanism in order to simplify the definition and processing of 100-continue.

8.1.2. Max-Forwards

The "Max-Forwards" header field provides a mechanism with the TRACE (Section 7.3.8) and OPTIONS (Section 7.3.7) request methods to limit the number of times that the request is forwarded by proxies. This can be useful when the client is attempting to trace a request that appears to be failing or looping mid-chain.

```
  Max-Forwards = 1*DIGIT
```

The Max-Forwards value is a decimal integer indicating the remaining number of times this request message can be forwarded.

Each intermediary that receives a TRACE or OPTIONS request containing a Max-Forwards header field MUST check and update its value prior to forwarding the request. If the received value is zero (0), the intermediary MUST NOT forward the request; instead, the intermediary MUST respond as the final recipient. If the received Max-Forwards value is greater than zero, the intermediary MUST generate an updated Max-Forwards field in the forwarded message with a field-value that is the lesser of a) the received value decremented by one (1) or b) the recipient's maximum supported value for Max-Forwards.

A recipient MAY ignore a Max-Forwards header field received with any other request methods.

8.2. Preconditions

A conditional request is an HTTP request with one or more request header fields that indicate a precondition to be tested before applying the request method to the target resource. Section 8.2.1 defines when preconditions are applied. Section 8.2.2 defines the order of evaluation when more than one precondition is present.

Conditional GET requests are the most efficient mechanism for HTTP cache updates [Caching]. 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 6) 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 following request header fields allow a client to place a precondition on the state of the target resource, so that the action corresponding to the method semantics will not be applied if the precondition evaluates to false. Each precondition defined by this specification consists of a comparison between a set of validators obtained from prior representations of the target resource to the current state of validators for the selected representation (Section 10.2). Hence, these preconditions evaluate whether the state of the target resource has changed since a given state known by the client. The effect of such an evaluation depends on the method semantics and choice of conditional, as defined in Section 8.2.1.

Header Field Name	Defined in...
If-Match	Section 8.2.3
If-None-Match	Section 8.2.4
If-Modified-Since	Section 8.2.5
If-Unmodified-Since	Section 8.2.6
If-Range	Section 8.2.7

8.2.1. Evaluation

Except when excluded below, a recipient cache or origin server MUST evaluate received request preconditions after it has successfully performed its normal request checks and just before it would perform the action associated with the request method. A server MUST ignore all received preconditions if its response to the same request without those conditions would have been a status code other than a 2xx (Successful) or 412 (Precondition Failed). In other words, redirects and failures take precedence over the evaluation of preconditions in conditional requests.

A server that is not the origin server for the target resource and cannot act as a cache for requests on the target resource MUST NOT evaluate the conditional request header fields defined by this specification, and it MUST forward them if the request is forwarded, since the generating client intends that they be evaluated by a server that can provide a current representation. Likewise, a server MUST ignore the conditional request header fields defined by this specification when received with a request method that does not involve the selection or modification of a selected representation, such as CONNECT, OPTIONS, or TRACE.

Conditional request header fields that are defined by extensions to HTTP might place conditions on all recipients, on the state of the target resource in general, or on a group of resources. For instance, the "If" header field in WebDAV can make a request conditional on various aspects of multiple resources, such as locks, if the recipient understands and implements that field ([RFC4918], Section 10.4).

Although conditional request header fields are defined as being usable with the HEAD method (to keep HEAD's semantics consistent with those of GET), there is no point in sending a conditional HEAD because a successful response is around the same size as a 304 (Not Modified) response and more useful than a 412 (Precondition Failed) response.

8.2.2. Precedence

When more than one conditional request header field is present in a request, the order in which the fields are evaluated becomes important. In practice, the fields defined in this document are consistently implemented in a single, logical order, since "lost update" preconditions have more strict requirements than cache validation, a validated cache is more efficient than a partial response, and entity tags are presumed to be more accurate than date validators.

A recipient cache or origin server MUST evaluate the request preconditions defined by this specification in the following order:

1. When recipient is the origin server and If-Match is present, evaluate the If-Match precondition:

   * if true, continue to step 3

   * if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 8.2.3)

2. When recipient is the origin server, If-Match is not present, and If-Unmodified-Since is present, evaluate the If-Unmodified-Since precondition:

   * if true, continue to step 3

   * if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 8.2.6)

3. When If-None-Match is present, evaluate the If-None-Match precondition:

   * if true, continue to step 5

   * if false for GET/HEAD, respond 304 (Not Modified)

   * if false for other methods, respond 412 (Precondition Failed)

4. When the method is GET or HEAD, If-None-Match is not present, and If-Modified-Since is present, evaluate the If-Modified-Since precondition:

   * if true, continue to step 5

   * if false, respond 304 (Not Modified)

5. When the method is GET and both Range and If-Range are present, evaluate the If-Range precondition:

   * if the validator matches and the Range specification is applicable to the selected representation, respond 206 (Partial Content)

6. Otherwise,

   * all conditions are met, so perform the requested action and respond according to its success or failure.

Any extension to HTTP/1.1 that defines additional conditional request header fields ought to define its own expectations regarding the order for evaluating such fields in relation to those defined in this document and other conditionals that might be found in practice.

8.2.3. If-Match

The "If-Match" header field makes the request method conditional on the recipient origin server either having at least one current representation of the target resource, when the field-value is "*", or having a current representation of the target resource that has an entity-tag matching a member of the list of entity-tags provided in the field-value.

An origin server MUST use the strong comparison function when comparing entity-tags for If-Match (Section 10.2.3.2), since the client intends this precondition to prevent the method from being applied if there have been any changes to the representation data.

```
  If-Match = "*" / 1#entity-tag
```

Examples:

```
  If-Match: "xyzzy"
  If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
  If-Match: *
```

If-Match is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on the same resource (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Match header field MUST evaluate the condition prior to performing the method (Section 8.2.1). If the field-value is "*", the condition is false if the origin server does not have a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if none of the listed tags match the entity-tag of the selected representation.

An origin server MUST NOT perform the requested method if a received If-Match condition evaluates to false; instead, the origin server MUST respond with either a) the 412 (Precondition Failed) status code or b) one of the 2xx (Successful) status codes if the origin server has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of it, perhaps because the prior response was lost or a compatible change was made by some other user agent). In the latter case, the origin server MUST NOT send a validator header field in the response unless it can

verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Match header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

8.2.4. If-None-Match

The "If-None-Match" header field makes the request method conditional on a recipient cache or origin server either not having any current representation of the target resource, when the field-value is "*", or having a selected representation with an entity-tag that does not match any of those listed in the field-value.

A recipient MUST use the weak comparison function when comparing entity-tags for If-None-Match (Section 10.2.3.2), since weak entity-tags can be used for cache validation even if there have been changes to the representation data.

```
  If-None-Match = "*" / 1#entity-tag
```

Examples:

```
  If-None-Match: "xyzzy"
  If-None-Match: W/"xyzzy"
  If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
  If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"
  If-None-Match: *
```

If-None-Match is primarily used in conditional GET requests to enable efficient updates of cached information with a minimum amount of transaction overhead. When a client desires to update one or more stored responses that have entity-tags, the client SHOULD generate an If-None-Match header field containing a list of those entity-tags when making a GET request; this allows recipient servers to send a 304 (Not Modified) response to indicate when one of those stored responses matches the selected representation.

If-None-Match can also be used with a value of "*" to prevent an unsafe request method (e.g., PUT) from inadvertently modifying an existing representation of the target resource when the client believes that the resource does not have a current representation (Section 7.2.1). This is a variation on the "lost update" problem that might arise if more than one client attempts to create an initial representation for the target resource.

An origin server that receives an If-None-Match header field MUST evaluate the condition prior to performing the method

(Section 8.2.1). If the field-value is "*", the condition is false if the origin server has a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if one of the listed tags match the entity-tag of the selected representation.

An origin server MUST NOT perform the requested method if the condition evaluates to false; instead, the origin server MUST respond with either a) the 304 (Not Modified) status code if the request method is GET or HEAD or b) the 412 (Precondition Failed) status code for all other request methods.

Requirements on cache handling of a received If-None-Match header field are defined in Section 4.3.2 of [Caching].

8.2.5. If-Modified-Since

The "If-Modified-Since" header field makes a GET or HEAD request method conditional on the selected representation's modification date being more recent than the date provided in the field-value. Transfer of the selected representation's data is avoided if that data has not changed.

```
  If-Modified-Since = HTTP-date
```

An example of the field is:

```
  If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

A recipient MUST ignore If-Modified-Since if the request contains an If-None-Match header field; the condition in If-None-Match is considered to be a more accurate replacement for the condition in If-Modified-Since, and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-None-Match.

A recipient MUST ignore the If-Modified-Since header field if the received field-value is not a valid HTTP-date, or if the request method is neither GET nor HEAD.

A recipient MUST interpret an If-Modified-Since field-value's timestamp in terms of the origin server's clock.

If-Modified-Since is typically used for two distinct purposes: 1) to allow efficient updates of a cached representation that does not have an entity-tag and 2) to limit the scope of a web traversal to resources that have recently changed.

When used for cache updates, a cache will typically use the value of the cached message's Last-Modified field to generate the field value of If-Modified-Since. This behavior is most interoperable for cases where clocks are poorly synchronized or when the server has chosen to only honor exact timestamp matches (due to a problem with Last-Modified dates that appear to go "back in time" when the origin server's clock is corrected or a representation is restored from an archived backup). However, caches occasionally generate the field value based on other data, such as the Date header field of the cached message or the local clock time that the message was received, particularly when the cached message does not contain a Last-Modified field.

When used for limiting the scope of retrieval to a recent time window, a user agent will generate an If-Modified-Since field value based on either its own local clock or a Date header field received from the server in a prior response. Origin servers that choose an exact timestamp match based on the selected representation's Last-Modified field will not be able to help the user agent limit its data transfers to only those changed during the specified window.

An origin server that receives an If-Modified-Since header field SHOULD evaluate the condition prior to performing the method (Section 8.2.1). The origin server SHOULD NOT perform the requested method if the selected representation's last modification date is earlier than or equal to the date provided in the field-value; instead, the origin server SHOULD generate a 304 (Not Modified) response, including only those metadata that are useful for identifying or updating a previously cached response.

Requirements on cache handling of a received If-Modified-Since header field are defined in Section 4.3.2 of [Caching].

8.2.6. If-Unmodified-Since

The "If-Unmodified-Since" header field makes the request method conditional on the selected representation's last modification date being earlier than or equal to the date provided in the field-value. This field accomplishes the same purpose as If-Match for cases where the user agent does not have an entity-tag for the representation.

```
  If-Unmodified-Since = HTTP-date
```

An example of the field is:

```
  If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

A recipient MUST ignore If-Unmodified-Since if the request contains an If-Match header field; the condition in If-Match is considered to be a more accurate replacement for the condition in If-Unmodified-Since, and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-Match.

A recipient MUST ignore the If-Unmodified-Since header field if the received field-value is not a valid HTTP-date.

A recipient MUST interpret an If-Unmodified-Since field-value's timestamp in terms of the origin server's clock.

If-Unmodified-Since is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on a resource that does not supply entity-tags with its representations (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Unmodified-Since header field MUST evaluate the condition prior to performing the method (Section 8.2.1). The origin server MUST NOT perform the requested method if the selected representation's last modification date is more recent than the date provided in the field-value; instead the origin server MUST respond with either a) the 412 (Precondition Failed) status code or b) one of the 2xx (Successful) status codes if the origin server has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of that because the prior response message was lost or a compatible change was made by some other user agent). In the latter case, the origin server MUST NOT send a validator header field in the response unless it can verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Unmodified-Since header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

8.2.7. If-Range

The "If-Range" header field provides a special conditional request mechanism that is similar to the If-Match and If-Unmodified-Since header fields but that instructs the recipient to ignore the Range header field if the validator doesn't match, resulting in transfer of the new selected representation instead of a 412 (Precondition Failed) response.

If a client has a partial copy of a representation and wishes to have an up-to-date copy of the entire representation, it could use the Range header field with a conditional GET (using either or both of If-Unmodified-Since and If-Match.) However, if the precondition fails because the representation has been modified, the client would then have to make a second request to obtain the entire current representation.

The "If-Range" header field allows a client to "short-circuit" the second request. Informally, its meaning is as follows: if the representation is unchanged, send me the part(s) that I am requesting in Range; otherwise, send me the entire representation.

```
  If-Range = entity-tag / HTTP-date
```

A client MUST NOT generate an If-Range header field in a request that does not contain a Range header field. A server MUST ignore an If-Range header field received in a request that does not contain a Range header field. An origin server MUST ignore an If-Range header field received in a request for a target resource that does not support Range requests.

A client MUST NOT generate an If-Range header field containing an entity-tag that is marked as weak. A client MUST NOT generate an If-Range header field containing an HTTP-date unless the client has no entity-tag for the corresponding representation and the date is a strong validator in the sense defined by Section 10.2.2.2.

A server that evaluates an If-Range precondition MUST use the strong comparison function when comparing entity-tags (Section 10.2.3.2) and MUST evaluate the condition as false if an HTTP-date validator is provided that is not a strong validator in the sense defined by Section 10.2.2.2. A valid entity-tag can be distinguished from a valid HTTP-date by examining the first two characters for a DQUOTE.

If the validator given in the If-Range header field matches the current validator for the selected representation of the target resource, then the server SHOULD process the Range header field as requested. If the validator does not match, the server MUST ignore the Range header field. Note that this comparison by exact match, including when the validator is an HTTP-date, differs from the "earlier than or equal to" comparison used when evaluating an If-Unmodified-Since conditional.

8.3. Range

The "Range" header field on a GET request modifies the method semantics to request transfer of only one or more subranges of the selected representation data, rather than the entire selected representation data.

```
  Range = byte-ranges-specifier / other-ranges-specifier
  other-ranges-specifier = other-range-unit "=" other-range-set
  other-range-set = 1*VCHAR
```

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.

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.

A server MAY ignore the Range header field. However, origin servers and intermediate caches ought to support byte ranges when possible, since Range supports efficient recovery from partially failed transfers and partial retrieval of large representations. A server MUST ignore a Range header field received with a request method other than GET.

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

An origin server MUST ignore a Range header field that contains a range unit it does not understand. A proxy MAY discard a Range header field that contains a range unit it does not understand.

A server that supports range requests MAY ignore or reject a Range header field that consists of more than two overlapping ranges, or a set of many small ranges that are not listed in ascending order, since both are indications of either a broken client or a deliberate denial-of-service attack (Section 12.13). A client SHOULD NOT

request multiple ranges that are inherently less efficient to process and transfer than a single range that encompasses the same data.

A client that is requesting multiple ranges SHOULD list those ranges in ascending order (the order in which they would typically be received in a complete representation) unless there is a specific need to request a later part earlier. For example, a user agent processing a large representation with an internal catalog of parts might need to request later parts first, particularly if the representation consists of pages stored in reverse order and the user agent wishes to transfer one page at a time.

The Range header field is evaluated after evaluating the precondition header fields defined in Section 8.2, and only if the result in absence of the Range header field would be a 200 (OK) response. In other words, Range is ignored when a conditional GET would result in a 304 (Not Modified) response.

The If-Range header field (Section 8.2.7) can be used as a precondition to applying the Range header field.

If all of the preconditions are true, the server supports the Range header field for the target resource, and the specified range(s) are valid and satisfiable (as defined in Section 6.1.4.1), the server SHOULD send a 206 (Partial Content) response with a payload containing one or more partial representations that correspond to the satisfiable ranges requested.

If all of the preconditions are true, the server supports the Range header field for the target resource, and the specified range(s) are invalid or unsatisfiable, the server SHOULD send a 416 (Range Not Satisfiable) response.

8.4. 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 6.4.1. 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 8.4.2
Accept-Charset	Section 8.4.3
Accept-Encoding	Section 8.4.4
Accept-Language	Section 8.4.5

8.4.1. 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 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.

8.4.2. Accept

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 8.4.1), 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.

8.4.3. Accept-Charset

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 6.1.1.1. 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 8.4.1. 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 12.11).

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.

8.4.4. Accept-Encoding

The "Accept-Encoding" header field can be used by user agents to indicate what response content-codings (Section 6.1.2) are acceptable in the response. An "identity" token is used as a synonym for "no encoding" in order to communicate when no encoding is preferred.

```
  Accept-Encoding  = #( codings [ weight ] )
  codings          = content-coding / "identity" / "*"
```

Each codings value MAY be given an associated quality value representing the preference for that encoding, as defined in Section 8.4.1. The asterisk "*" symbol in an Accept-Encoding field matches any available content-coding not explicitly listed in the header field.

For example,

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

A request without an Accept-Encoding header field implies that the user agent has no preferences regarding content-codings. Although this allows the server to use any content-coding in a response, it does not imply that the user agent will be able to correctly process all encodings.

A server tests whether a content-coding for a given representation is acceptable using these rules:

1. If no Accept-Encoding field is in the request, any content-coding is considered acceptable by the user agent.

2. If the representation has no content-coding, then it is acceptable by default unless specifically excluded by the Accept-Encoding field stating either "identity;q=0" or "*;q=0" without a more specific entry for "identity".

3. If the representation's content-coding is one of the content-codings listed in the Accept-Encoding field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in Section 8.4.1, a qvalue of 0 means "not acceptable".)

4. If multiple content-codings are acceptable, then the acceptable content-coding with the highest non-zero qvalue is preferred.

An Accept-Encoding header field with a combined field-value that is empty implies that the user agent does not want any content-coding in response. If an Accept-Encoding header field is present in a request and none of the available representations for the response have a content-coding that is listed as acceptable, the origin server SHOULD send a response without any content-coding.

> Note: Most HTTP/1.0 applications do not recognize or obey qvalues associated with content-codings. This means that qvalues might not work and are not permitted with x-gzip or x-compress.

8.4.5. 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 6.1.3.

```
  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 8.4.1. 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 12.11).

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.

8.5. Authentication Credentials

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.

Two header fields are used for carrying authentication credentials. 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 8.5.3
Proxy-Authorization	Section 8.5.4

8.5.1. 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 12.14.1.

```
  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 9.5.4).

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.

8.5.2. 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 5.3) 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.

8.5.3. Authorization

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 [Caching] for details of and requirements pertaining to handling of the Authorization field by HTTP caches.

8.5.4. Proxy-Authorization

The "Proxy-Authorization" header field allows the client to identify itself (or its user) to a proxy that requires authentication. Its value consists of credentials containing the authentication information of the client for the proxy and/or realm of the resource being requested.

```
  Proxy-Authorization = credentials
```

Unlike Authorization, the Proxy-Authorization header field applies only to the next inbound proxy that demanded authentication using the Proxy-Authenticate field. When multiple proxies are used in a chain, the Proxy-Authorization header field is consumed by the first inbound proxy that was expecting to receive credentials. A proxy MAY relay

the credentials from the client request to the next proxy if that is the mechanism by which the proxies cooperatively authenticate a given request.

8.5.5. Authentication Scheme Extensibility

Aside from the general framework, this document does not specify any authentication schemes. New and existing authentication schemes are specified independently and ought to be registered within the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry". For example, the "basic" and "digest" authentication schemes are defined by RFC 7617 and RFC 7616, respectively.

8.5.5.1. Authentication Scheme Registry

The "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" defines the namespace for the authentication schemes in challenges and credentials. It is maintained at <https://www.iana.org/assignments/http-authschemes>.

Registrations MUST include the following fields:

- Authentication Scheme Name
- Pointer to specification text
- Notes (optional)

Values to be added to this namespace require IETF Review (see [RFC5226], Section 4.1).

8.5.5.2. Considerations for New Authentication Schemes

There are certain aspects of the HTTP Authentication framework that put constraints on how new authentication schemes can work:

- HTTP authentication is presumed to be stateless: all of the information necessary to authenticate a request MUST be provided in the request, rather than be dependent on the server remembering prior requests. Authentication based on, or bound to, the underlying connection is outside the scope of this specification and inherently flawed unless steps are taken to ensure that the connection cannot be used by any party other than the authenticated user (see Section 2.2).

- The authentication parameter "realm" is reserved for defining protection spaces as described in Section 8.5.2. New schemes MUST NOT use it in a way incompatible with that definition.

- The "token68" notation was introduced for compatibility with existing authentication schemes and can only be used once per challenge or credential. Thus, new schemes ought to use the auth-param syntax instead, because otherwise future extensions will be impossible.

- The parsing of challenges and credentials is defined by this specification and cannot be modified by new authentication schemes. When the auth-param syntax is used, all parameters ought to support both token and quoted-string syntax, and syntactical constraints ought to be defined on the field value after parsing (i.e., quoted-string processing). This is necessary so that recipients can use a generic parser that applies to all authentication schemes.

  Note: The fact that the value syntax for the "realm" parameter is restricted to quoted-string was a bad design choice not to be repeated for new parameters.

- Definitions of new schemes ought to define the treatment of unknown extension parameters. In general, a "must-ignore" rule is preferable to a "must-understand" rule, because otherwise it will be hard to introduce new parameters in the presence of legacy recipients. Furthermore, it's good to describe the policy for defining new parameters (such as "update the specification" or "use this registry").

- Authentication schemes need to document whether they are usable in origin-server authentication (i.e., using WWW-Authenticate), and/or proxy authentication (i.e., using Proxy-Authenticate).

- The credentials carried in an Authorization header field are specific to the user agent and, therefore, have the same effect on HTTP caches as the "private" Cache-Control response directive

(Section 5.2.2.6 of [Caching]), within the scope of the request in which they appear.

Therefore, new authentication schemes that choose not to carry credentials in the Authorization header field (e.g., using a newly defined header field) will need to explicitly disallow caching, by mandating the use of either Cache-Control request directives (e.g., "no-store", Section 5.2.1.5 of [Caching]) or response directives (e.g., "private").

8.6. 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 8.6.1
Referer	Section 8.6.2
User-Agent	Section 8.6.3

8.6.1. From

The "From" header field contains an Internet email address for a human user who controls the requesting user agent. The address ought to be machine-usable, as defined by "mailbox" in Section 3.4 of [RFC5322]:

```
  From    = mailbox

  mailbox = <mailbox, see [RFC5322], Section 3.4>
```

An example is:

```
  From: webmaster@example.org
```

The From header field is rarely sent by non-robotic user agents. A user agent SHOULD NOT send a From header field without explicit configuration by the user, since that might conflict with the user's privacy interests or their site's security policy.

A robotic user agent SHOULD send a valid From header field so that the person responsible for running the robot can be contacted if

problems occur on servers, such as if the robot is sending excessive, unwanted, or invalid requests.

A server SHOULD NOT use the From header field for access control or authentication, since most recipients will assume that the field value is public information.

8.6.2. Referer

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 12.8 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.

8.6.3. User-Agent

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 5 of [Messaging]), 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.

9. 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

9.1. Overview of Status Codes

The status codes listed below are defined in this specification. 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 [Caching]; all other status codes are not cacheable by default.

Value	Description	Reference
100	Continue	Section 9.2.1
101	Switching Protocols	Section 9.2.2
200	OK	Section 9.3.1
201	Created	Section 9.3.2
202	Accepted	Section 9.3.3
203	Non-Authoritative Information	Section 9.3.4
204	No Content	Section 9.3.5
205	Reset Content	Section 9.3.6
206	Partial Content	Section 9.3.7
300	Multiple Choices	Section 9.4.1
301	Moved Permanently	Section 9.4.2
302	Found	Section 9.4.3
303	See Other	Section 9.4.4
304	Not Modified	Section 9.4.5
305	Use Proxy	Section 9.4.6
306	(Unused)	Section 9.4.7
307	Temporary Redirect	Section 9.4.8
400	Bad Request	Section 9.5.1
401	Unauthorized	Section 9.5.2
402	Payment Required	Section 9.5.3
403	Forbidden	Section 9.5.4
404	Not Found	Section 9.5.5
405	Method Not Allowed	Section 9.5.6
406	Not Acceptable	Section 9.5.7
407	Proxy Authentication Required	Section 9.5.8
408	Request Timeout	Section 9.5.9
409	Conflict	Section 9.5.10
410	Gone	Section 9.5.11
411	Length Required	Section 9.5.12
412	Precondition Failed	Section 9.5.13
413	Payload Too Large	Section 9.5.14
414	URI Too Long	Section 9.5.15
415	Unsupported Media Type	Section 9.5.16
416	Range Not Satisfiable	Section 9.5.17
417	Expectation Failed	Section 9.5.18
426	Upgrade Required	Section 9.5.19
500	Internal Server Error	Section 9.6.1
501	Not Implemented	Section 9.6.2
502	Bad Gateway	Section 9.6.3
503	Service Unavailable	Section 9.6.4
504	Gateway Timeout	Section 9.6.5
505	HTTP Version Not Supported	Section 9.6.6

Note that this list is not exhaustive -- it does not include extension status codes defined in other specifications (Section 9.7).

9.2. 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).

9.2.1. 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 8.1.1. 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.

9.2.2. 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 9.7 of [Messaging]), 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.

9.3. Successful 2xx

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

9.3.1. 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 [Caching]).

9.3.2. 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 10.2 for a discussion of the meaning and purpose of validator header fields, such as ETag and Last-Modified, in a 201 response.

9.3.3. 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.

9.3.4. 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 5.6.2). 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 [Caching]), 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 [Caching]).

9.3.5. 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 [Caching]).

9.3.6. 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.

9.3.7. 206 Partial Content

The 206 (Partial Content) status code indicates that the server is successfully fulfilling a range request for the target resource by transferring one or more parts of the selected representation that correspond to the satisfiable ranges found in the request's Range header field (Section 8.3).

When a 206 response is generated, the server MUST generate the following header fields, in addition to those required in the subsections below, if the field would have been sent in a 200 (OK) response to the same request: Date, Cache-Control, ETag, Expires, Content-Location, and Vary.

If a 206 is generated in response to a request with an If-Range header field, the sender SHOULD NOT generate other representation header fields beyond those required, because the client is understood to already have a prior response containing those header fields. Otherwise, the sender MUST generate all of the representation header fields that would have been sent in a 200 (OK) response to the same request.

A 206 response is cacheable by default; i.e., unless otherwise indicated by explicit cache controls (see Section 4.2.2 of [Caching]).

9.3.7.1. Single Part

If a single part is being transferred, the server generating the 206 response MUST generate a Content-Range header field, describing what range of the selected representation is enclosed, and a payload consisting of the range. For example:

```
     HTTP/1.1 206 Partial Content
     Date: Wed, 15 Nov 1995 06:25:24 GMT
     Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
     Content-Range: bytes 21010-47021/47022
     Content-Length: 26012
     Content-Type: image/gif

     ... 26012 bytes of partial image data ...
```

9.3.7.2. Multiple Parts

If multiple parts are being transferred, the server generating the 206 response MUST generate a "multipart/byteranges" payload, as defined in Section 6.3.4, and a Content-Type header field containing the multipart/byteranges media type and its required boundary parameter. To avoid confusion with single-part responses, a server MUST NOT generate a Content-Range header field in the HTTP header section of a multiple part response (this field will be sent in each part instead).

Within the header area of each body part in the multipart payload, the server MUST generate a Content-Range header field corresponding to the range being enclosed in that body part. If the selected representation would have had a Content-Type header field in a 200 (OK) response, the server SHOULD generate that same Content-Type field in the header area of each body part. For example:

```
     HTTP/1.1 206 Partial Content
     Date: Wed, 15 Nov 1995 06:25:24 GMT
     Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
     Content-Length: 1741
     Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES

     --THIS_STRING_SEPARATES
     Content-Type: application/pdf
     Content-Range: bytes 500-999/8000

     ...the first range...
     --THIS_STRING_SEPARATES
     Content-Type: application/pdf
     Content-Range: bytes 7000-7999/8000

     ...the second range
     --THIS_STRING_SEPARATES--
```

When multiple ranges are requested, a server MAY coalesce any of the ranges that overlap, or that are separated by a gap that is smaller than the overhead of sending multiple parts, regardless of the order

in which the corresponding byte-range-spec appeared in the received Range header field. Since the typical overhead between parts of a multipart/byteranges payload is around 80 bytes, depending on the selected representation's media type and the chosen boundary parameter length, it can be less efficient to transfer many small disjoint parts than it is to transfer the entire selected representation.

A server MUST NOT generate a multipart response to a request for a single range, since a client that does not request multiple parts might not support multipart responses. However, a server MAY generate a multipart/byteranges payload with only a single body part if multiple ranges were requested and only one range was found to be satisfiable or only one range remained after coalescing. A client that cannot process a multipart/byteranges response MUST NOT generate a request that asks for multiple ranges.

When a multipart response payload is generated, the server SHOULD send the parts in the same order that the corresponding byte-range-spec appeared in the received Range header field, excluding those ranges that were deemed unsatisfiable or that were coalesced into other ranges. A client that receives a multipart response MUST inspect the Content-Range header field present in each body part in order to determine which range is contained in that body part; a client cannot rely on receiving the same ranges that it requested, nor the same order that it requested.

9.3.7.3. **Combining Parts**

A response might transfer only a subrange of a representation if the connection closed prematurely or if the request used one or more Range specifications. After several such transfers, a client might have received several ranges of the same representation. These ranges can only be safely combined if they all have in common the same strong validator (Section 10.2.1).

A client that has received multiple partial responses to GET requests on a target resource MAY combine those responses into a larger continuous range if they share the same strong validator.

If the most recent response is an incomplete 200 (OK) response, then the header fields of that response are used for any combined response and replace those of the matching stored responses.

If the most recent response is a 206 (Partial Content) response and at least one of the matching stored responses is a 200 (OK), then the combined response header fields consist of the most recent 200 response's header fields. If all of the matching stored responses

are 206 responses, then the stored response with the most recent header fields is used as the source of header fields for the combined response, except that the client MUST use other header fields provided in the new response, aside from Content-Range, to replace all instances of the corresponding header fields in the stored response.

The combined response message body consists of the union of partial content ranges in the new response and each of the selected responses. If the union consists of the entire range of the representation, then the client MUST process the combined response as if it were a complete 200 (OK) response, including a Content-Length header field that reflects the complete length. Otherwise, the client MUST process the set of continuous ranges as one of the following: an incomplete 200 (OK) response if the combined response is a prefix of the representation, a single 206 (Partial Content) response containing a multipart/byteranges body, or multiple 206 (Partial Content) responses, each with one continuous range that is indicated by a Content-Range header field.

9.4. 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 10.1.2) 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 7.2.1, 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.

9.4.1. 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 6.4).

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 [Caching]).

> 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.

9.4.2. 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 [Caching]).

9.4.3. 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.

9.4.4. 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.

9.4.5. 304 Not Modified

The 304 (Not Modified) status code indicates that a conditional GET or HEAD request has been received and would have resulted in a 200 (OK) response if it were not for the fact that the condition evaluated to false. In other words, there is no need for the server to transfer a representation of the target resource because the request indicates that the client, which made the request conditional, already has a valid representation; the server is therefore redirecting the client to make use of that stored representation as if it were the payload of a 200 (OK) response.

The server generating a 304 response MUST generate any of the following header fields that would have been sent in a 200 (OK) response to the same request: Cache-Control, Content-Location, Date, ETag, Expires, and Vary.

Since the goal of a 304 response is to minimize information transfer when the recipient already has one or more cached representations, a sender SHOULD NOT generate representation metadata other than the above listed fields unless said metadata exists for the purpose of guiding cache updates (e.g., Last-Modified might be useful if the response does not have an ETag field).

Requirements on a cache that receives a 304 response are defined in Section 4.3.4 of [Caching]. If the conditional request originated with an outbound client, such as a user agent with its own cache sending a conditional GET to a shared proxy, then the proxy SHOULD forward the 304 response to that client.

A 304 response cannot contain a message-body; it is always terminated by the first empty line after the header fields.

9.4.6. 305 Use Proxy

The 305 (Use Proxy) status code was defined in a previous version of this specification and is now deprecated (Appendix B of [RFC7231]).

9.4.7. 306 (Unused)

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

9.4.8. 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).

9.5. 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.

9.5.1. 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).

9.5.2. 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 10.3.1) 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 8.5.3). 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.

9.5.3. 402 Payment Required

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

9.5.4. 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).

9.5.5. 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 [Caching]).

9.5.6. 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 [Caching]).

9.5.7. 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 8.4), 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 9.4.1.

9.5.8. 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 10.3.2) 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 8.5.4).

9.5.9. 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 9.1 of [Messaging]) 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.

9.5.10. 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.

9.5.11. 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 [Caching]).

9.5.12. 411 Length Required

The 411 (Length Required) status code indicates that the server refuses to accept the request without a defined Content-Length (Section 6.2.4). 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.

9.5.13. 412 Precondition Failed

The 412 (Precondition Failed) status code indicates that one or more conditions given in the request header fields evaluated to false when tested on the server. This response code allows the client to place preconditions on the current resource state (its current representations and metadata) and, thus, prevent the request method from being applied if the target resource is in an unexpected state.

9.5.14. 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.

9.5.15. 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 3.2 of [Messaging]) 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 [Caching]).

9.5.16. 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.

9.5.17. 416 Range Not Satisfiable

The 416 (Range Not Satisfiable) status code indicates that none of the ranges in the request's Range header field (Section 8.3) overlap the current extent of the selected resource or that the set of ranges requested has been rejected due to invalid ranges or an excessive request of small or overlapping ranges.

For byte ranges, failing to overlap the current extent means that the first-byte-pos of all of the byte-range-spec values were greater than the current length of the selected representation. When this status code is generated in response to a byte-range request, the sender SHOULD generate a Content-Range header field specifying the current length of the selected representation (Section 6.3.3).

For example:

```
HTTP/1.1 416 Range Not Satisfiable
Date: Fri, 20 Jan 2012 15:41:54 GMT
Content-Range: bytes */47022
```

> Note: Because servers are free to ignore Range, many implementations will simply respond with the entire selected representation in a 200 (OK) response. That is partly because most clients are prepared to receive a 200 (OK) to complete the task (albeit less efficiently) and partly because clients might not stop making an invalid partial request until they have received a complete representation. Thus, clients cannot depend on receiving a 416 (Range Not Satisfiable) response even when it is most appropriate.

9.5.18. 417 Expectation Failed

The 417 (Expectation Failed) status code indicates that the expectation given in the request's Expect header field (Section 8.1.1) could not be met by at least one of the inbound servers.

9.5.19. 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 9.7 of [Messaging]).

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.
```

9.6. 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.

9.6.1. 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.

9.6.2. 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 [Caching]).

9.6.3. 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.

9.6.4. 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 10.1.3) 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.

9.6.5. 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.

9.6.6. 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 3.5, 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.

9.7. Status Code Extensibility

Additional status codes, outside the scope of this specification, have been specified for use in HTTP. All such status codes ought to be registered within the "Hypertext Transfer Protocol (HTTP) Status Code Registry".

9.7.1. Status Code Registry

The "Hypertext Transfer Protocol (HTTP) Status Code Registry", maintained by IANA at <https://www.iana.org/assignments/http-status-codes>, registers status-code numbers.

A registration MUST include the following fields:

o Status Code (3 digits)

- Short Description

- Pointer to specification text

Values to be added to the HTTP status code namespace require IETF Review (see [RFC5226], Section 4.1).

9.7.2. Considerations for New Status Codes

When it is necessary to express semantics for a response that are not defined by current status codes, a new status code can be registered. Status codes are generic; they are potentially applicable to any resource, not just one particular media type, kind of resource, or application of HTTP. As such, it is preferred that new status codes be registered in a document that isn't specific to a single application.

New status codes are required to fall under one of the categories defined in Section 9. To allow existing parsers to process the response message, new status codes cannot disallow a payload, although they can mandate a zero-length payload body.

Proposals for new status codes that are not yet widely deployed ought to avoid allocating a specific number for the code until there is clear consensus that it will be registered; instead, early drafts can use a notation such as "4NN", or "3N0" .. "3N9", to indicate the class of the proposed status code(s) without consuming a number prematurely.

The definition of a new status code ought to explain the request conditions that would cause a response containing that status code (e.g., combinations of request header fields and/or method(s)) along with any dependencies on response header fields (e.g., what fields are required, what fields can modify the semantics, and what header field semantics are further refined when used with the new status code).

The definition of a new status code ought to specify whether or not it is cacheable. Note that all status codes can be cached if the response they occur in has explicit freshness information; however, status codes that are defined as being cacheable are allowed to be cached without explicit freshness information. Likewise, the definition of a status code can place constraints upon cache behavior. See [Caching] for more information.

Finally, the definition of a new status code ought to indicate whether the payload has any implied association with an identified resource (Section 6.3.2).

10. 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.

10.1. 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 [Caching]
Cache-Control	Section 5.2 of [Caching]
Expires	Section 5.3 of [Caching]
Date	Section 10.1.1.2
Location	Section 10.1.2
Retry-After	Section 10.1.3
Vary	Section 10.1.4
Warning	Section 5.5 of [Caching]

10.1.1. Origination Date

10.1.1.1. 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.

10.1.1.2. Date

The "Date" header field represents the date and time at which the message was originated, having the same semantics as the Origination Date Field (orig-date) defined in Section 3.6.1 of [RFC5322]. The field value is an HTTP-date, as defined in Section 10.1.1.1.

```
     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.

10.1.2. Location

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 6.2.5) 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.

10.1.3. Retry-After

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.

10.1.4. Vary

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).

```
  Vary = "*" / 1#field-name
```

A Vary field value of "*" signals that anything about the request might play a role in selecting the response representation, possibly including elements outside the message syntax (e.g., the client's network address). A recipient will not be able to determine whether this response is appropriate for a later request without forwarding the request to the origin server. A proxy MUST NOT generate a Vary field with a "*" value.

A Vary field value consisting of a comma-separated list of names indicates that the named request header fields, known as the selecting header fields, might have a role in selecting the representation. The potential selecting header fields are not limited to those defined by this specification.

For example, a response that contains

```
  Vary: accept-encoding, accept-language
```

indicates that the origin server might have used the request's Accept-Encoding and Accept-Language fields (or lack thereof) as determining factors while choosing the content for this response.

An origin server might send Vary with a list of fields for two purposes:

1. To inform cache recipients that they MUST NOT use this response to satisfy a later request unless the later request has the same values for the listed fields as the original request (Section 4.1 of [Caching]). In other words, Vary expands the cache key required to match a new request to the stored cache entry.

2. To inform user agent recipients that this response is subject to content negotiation (Section 8.4) and that a different representation might be sent in a subsequent request if additional parameters are provided in the listed header fields (proactive negotiation).

An origin server SHOULD send a Vary header field when its algorithm for selecting a representation varies based on aspects of the request message other than the method and request target, unless the variance cannot be crossed or the origin server has been deliberately configured to prevent cache transparency. For example, there is no need to send the Authorization field name in Vary because reuse across users is constrained by the field definition (Section 8.5.3). Likewise, an origin server might use Cache-Control directives (Section 5.2 of [Caching]) to supplant Vary if it considers the variance less significant than the performance cost of Vary's impact on caching.

10.2. Validators

Validator header fields convey metadata about the selected representation (Section 6). 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 Section 8.2.

Header Field Name	Defined in...
ETag	Section 10.2.3
Last-Modified	Section 10.2.2

This specification defines two forms of metadata that are commonly used to observe resource state and test for preconditions: modification dates (Section 10.2.2) and opaque entity tags (Section 10.2.3). 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.

10.2.1. 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.

10.2.2. Last-Modified

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
```

10.2.2.1. 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 ([Caching]) 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.

10.2.2.2. 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

- 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.

10.2.3. ETag

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. An entity-tag is an opaque validator for differentiating between multiple representations of the same resource, regardless of whether those multiple representations are due to resource state changes over time, content negotiation resulting in multiple representations being valid at the same time, or both. An entity-tag consists of an opaque quoted string, possibly prefixed by a weakness indicator.

```
  ETag       = entity-tag

  entity-tag = [ weak ] opaque-tag
  weak       = %x57.2F ; "W/", case-sensitive
  opaque-tag = DQUOTE *etagc DQUOTE
  etagc      = %x21 / %x23-7E / obs-text
             ; VCHAR except double quotes, plus obs-text
```

> Note: Previously, opaque-tag was defined to be a quoted-string ([RFC2616], Section 3.11); thus, some recipients might perform backslash unescaping. Servers therefore ought to avoid backslash characters in entity tags.

An entity-tag can be more reliable for validation than a modification date in situations where it is inconvenient to store modification dates, where the one-second resolution of HTTP date values is not sufficient, or where modification dates are not consistently maintained.

Examples:

```
ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""
```

An entity-tag can be either a weak or strong validator, with strong being the default. If an origin server provides an entity-tag for a representation and the generation of that entity-tag does not satisfy all of the characteristics of a strong validator (Section 10.2.1), then the origin server MUST mark the entity-tag as weak by prefixing its opaque value with "W/" (case-sensitive).

10.2.3.1. Generation

The principle behind entity-tags is that only the service author knows the implementation of a resource well enough to select the most accurate and efficient validation mechanism for that resource, and that any such mechanism can be mapped to a simple sequence of octets for easy comparison. Since the value is opaque, there is no need for the client to be aware of how each entity-tag is constructed.

For example, a resource that has implementation-specific versioning applied to all changes might use an internal revision number, perhaps combined with a variance identifier for content negotiation, to accurately differentiate between representations. Other implementations might use a collision-resistant hash of representation content, a combination of various file attributes, or a modification timestamp that has sub-second resolution.

An origin server SHOULD send an ETag for any selected representation for which detection of changes can be reasonably and consistently determined, since the entity-tag's use in conditional requests and evaluating cache freshness ([Caching]) can result in a substantial reduction of HTTP network traffic and can be a significant factor in improving service scalability and reliability.

10.2.3.2. Comparison

There are two entity-tag comparison functions, depending on whether or not the comparison context allows the use of weak validators:

- Strong comparison: two entity-tags are equivalent if both are not weak and their opaque-tags match character-by-character.

- Weak comparison: two entity-tags are equivalent if their opaque-tags match character-by-character, regardless of either or both being tagged as "weak".

The example below shows the results for a set of entity-tag pairs and both the weak and strong comparison function results:

ETag 1	ETag 2	Strong Comparison	Weak Comparison
W/"1"	W/"1"	no match	match
W/"1"	W/"2"	no match	no match
W/"1"	"1"	no match	match
"1"	"1"	match	match

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

Consider a resource that is subject to content negotiation (Section 6.4), and where the representations sent in response to a GET request vary based on the Accept-Encoding request header field (Section 8.4.4):

>> Request:

```
GET /index HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip
```

In this case, the response might or might not use the gzip content coding. If it does not, the response might look like:

>> Response:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
ETag: "123-a"
Content-Length: 70
Vary: Accept-Encoding
Content-Type: text/plain

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

An alternative representation that does use gzip content coding would be:

>> Response:

```
  HTTP/1.1 200 OK
  Date: Fri, 26 Mar 2010 00:05:00 GMT
  ETag: "123-b"
  Content-Length: 43
  Vary: Accept-Encoding
  Content-Type: text/plain
  Content-Encoding: gzip

  ...binary data...
```

Note: Content codings are a property of the representation data, so a strong entity-tag for a content-encoded representation has to be distinct from the entity tag of an unencoded representation to prevent potential conflicts during cache updates and range requests. In contrast, transfer codings (Section 7 of [Messaging]) apply only during message transfer and do not result in distinct entity-tags.

10.2.4. When to Use Entity-Tags and Last-Modified Dates

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

- SHOULD send an entity-tag validator unless it is not feasible to generate one.

- MAY send a weak entity-tag instead of a strong entity-tag, if performance considerations support the use of weak entity-tags, or if it is unfeasible to send a strong entity-tag.

- 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 both a strong entity-tag and a Last-Modified value in successful responses to a retrieval request.

A client:

- MUST send that entity-tag in any cache validation request (using If-Match or If-None-Match) if an entity-tag has been provided by the origin server.

- 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.

- 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.

- 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.

10.3. 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 10.3.1
Proxy-Authenticate	Section 10.3.2

10.3.1. WWW-Authenticate

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.

10.3.2. Proxy-Authenticate

The "Proxy-Authenticate" header field consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the proxy for this effective request URI (Section 5.3). A proxy MUST send at least one Proxy-Authenticate header field in each 407 (Proxy Authentication Required) response that it generates.

```
  Proxy-Authenticate = 1#challenge
```

Unlike WWW-Authenticate, the Proxy-Authenticate header field applies only to the next outbound client on the response chain. This is because only the client that chose a given proxy is likely to have the credentials necessary for authentication. However, when multiple proxies are used within the same administrative domain, such as office and regional caching proxies within a large corporate network, it is common for credentials to be generated by the user agent and passed through the hierarchy until consumed. Hence, in such a configuration, it will appear as if Proxy-Authenticate is being forwarded because each proxy will send the same challenge set.

Note that the parsing considerations for WWW-Authenticate apply to this header field as well; see Section 10.3.1 for details.

10.4. 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 10.4.1
Allow	Section 10.4.2
Server	Section 10.4.3

10.4.1. Accept-Ranges

The "Accept-Ranges" header field allows a server to indicate that it supports range requests for the target resource.

```
  Accept-Ranges     = acceptable-ranges
  acceptable-ranges = 1#range-unit / "none"
```

An origin server that supports byte-range requests for a given target resource MAY send

```
  Accept-Ranges: bytes
```

to indicate what range units are supported. A client MAY generate range requests without having received this header field for the resource involved. Range units are defined in Section 6.1.4.

A server that does not support any kind of range request for the target resource MAY send

```
  Accept-Ranges: none
```

to advise the client not to attempt a range request.

10.4.2. Allow

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.

10.4.3. Server

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 5 of [Messaging]), 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 8.6.3.

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.

11. ABNF List Extension: #rule

A #rule extension to the ABNF rules of [RFC5234] is used to improve readability in the definitions of some header field values.

A construct "#" is defined, similar to "*", for defining comma-delimited lists of elements. The full form is "<n>#<m>element" indicating at least <n> and at most <m> elements, each separated by a single comma (",") and optional whitespace (OWS).

In any production that uses the list construct, a sender MUST NOT generate empty list elements. In other words, a sender MUST generate lists that satisfy the following syntax:

```
1#element => element *( OWS "," OWS element )
```

and:

```
#element => [ 1#element ]
```

and for n >= 1 and m > 1:

```
<n>#<m>element => element <n-1>*<m-1>( OWS "," OWS element )
```

For compatibility with legacy list rules, a recipient MUST parse and ignore a reasonable number of empty list elements: enough to handle common mistakes by senders that merge values, but not so much that they could be used as a denial-of-service mechanism. In other words, a recipient MUST accept lists that satisfy the following syntax:

```
#element => [ ( "," / element ) *( OWS "," [ OWS element ] ) ]

1#element => *( "," OWS ) element *( OWS "," [ OWS element ] )
```

Empty elements do not contribute to the count of elements present. For example, given these ABNF productions:

```
example-list      = 1#example-list-elmt
example-list-elmt = token ; see Section 4.2.3
```

Then the following are valid values for example-list (not including the double quotes, which are present for delimitation only):

```
"foo,bar"
"foo ,bar,"
"foo , ,bar,charlie   "
```

In contrast, the following values would be invalid, since at least one non-empty element is required by the example-list production:

```
""
","
",   ,"
```

Appendix A shows the collected ABNF for recipients after the list constructs have been expanded.

12. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns relevant to HTTP semantics and its use for transferring information over the Internet. Considerations related to message syntax, parsing, and routing are discussed in Section 11 of [Messaging].

The list of considerations below is not exhaustive. Most security concerns related to HTTP semantics are about securing server-side applications (code behind the HTTP interface), securing user agent processing of payloads received via HTTP, or secure use of the Internet in general, rather than security of the protocol. Various organizations maintain topical information and links to current research on Web application security (e.g., [OWASP]).

12.1. 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 2.5.1). 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 2.5.1) 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 2.5.2) 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]).

12.2. 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 7 of [Caching].

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.

12.3. Attacks Based on File and Path Names

Origin servers frequently make use of their local file system to manage the mapping from effective request URI to resource representations. Most file systems are not designed to protect against malicious file or path names. Therefore, an origin server needs to avoid accessing names that have a special significance to the system when mapping the request target to files, folders, or directories.

For example, UNIX, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the

current one, and they use specially named paths or file names to send data to system devices. Similar naming conventions might exist within other types of storage systems. Likewise, local storage systems have an annoying tendency to prefer user-friendliness over security when handling invalid or unexpected characters, recomposition of decomposed characters, and case-normalization of case-insensitive names.

Attacks based on such special names tend to focus on either denial-of-service (e.g., telling the server to read from a COM port) or disclosure of configuration and source files that are not meant to be served.

12.4. Attacks Based on Command, Code, or Query Injection

Origin servers often use parameters within the URI as a means of identifying system services, selecting database entries, or choosing a data source. However, data received in a request cannot be trusted. An attacker could construct any of the request data elements (method, request-target, header fields, or body) to contain data that might be misinterpreted as a command, code, or query when passed through a command invocation, language interpreter, or database interface.

For example, SQL injection is a common attack wherein additional query language is inserted within some part of the request-target or header fields (e.g., Host, Referer, etc.). If the received data is used directly within a SELECT statement, the query language might be interpreted as a database command instead of a simple string value. This type of implementation vulnerability is extremely common, in spite of being easy to prevent.

In general, resource implementations ought to avoid use of request data in contexts that are processed or interpreted as instructions. Parameters ought to be compared to fixed strings and acted upon as a result of that comparison, rather than passed through an interface that is not prepared for untrusted data. Received data that isn't based on fixed parameters ought to be carefully filtered or encoded to avoid being misinterpreted.

Similar considerations apply to request data when it is stored and later processed, such as within log files, monitoring tools, or when included within a data format that allows embedded scripts.

12.5. 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 (Section 3.3).

To promote interoperability, specific recommendations are made for minimum size limits on request-line (Section 3 of [Messaging]) and header fields (Section 5 of [Messaging]). 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 9.5.15) or a request payload that is too large (Section 9.5.14). 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.

12.6. Disclosure of Personal Information

Clients are often privy to large amounts of personal information, including both information provided by the user to interact with resources (e.g., the user's name, location, mail address, passwords, encryption keys, etc.) and information about the user's browsing activity over time (e.g., history, bookmarks, etc.). Implementations need to prevent unintentional disclosure of personal information.

12.7. 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.

12.8. Disclosure of Sensitive Information in URIs

URIs are intended to be shared, not secured, even when they identify secure resources. URIs are often shown on displays, added to templates when a page is printed, and stored in a variety of unprotected bookmark lists. It is therefore unwise to include information within a URI that is sensitive, personally identifiable, or a risk to disclose.

Authors of services ought to avoid GET-based forms for the submission of sensitive data because that data will be placed in the request-target. Many existing servers, proxies, and user agents log or display the request-target in places where it might be visible to third parties. Such services ought to use POST-based form submission instead.

Since the Referer header field tells a target site about the context that resulted in a request, it has the potential to reveal information about the user's immediate browsing history and any personal information that might be found in the referring resource's URI. Limitations on the Referer header field are described in Section 8.6.2 to address some of its security considerations.

12.9. Disclosure of Fragment after Redirects

Although fragment identifiers used within URI references are not sent in requests, implementers ought to be aware that they will be visible to the user agent and any extensions or scripts running as a result of the response. In particular, when a redirect occurs and the original request's fragment identifier is inherited by the new reference in Location (Section 10.1.2), this might have the effect of disclosing one site's fragment to another site. If the first site uses personal information in fragments, it ought to ensure that redirects to other sites include a (possibly empty) fragment component in order to block that inheritance.

12.10. Disclosure of Product Information

The User-Agent (Section 8.6.3), Via (Section 5.6.1), and Server (Section 10.4.3) header fields often reveal information about the respective sender's software systems. In theory, this can make it easier for an attacker to exploit known security holes; in practice, attackers tend to try all potential holes regardless of the apparent software versions being used.

Proxies that serve as a portal through a network firewall ought to take special precautions regarding the transfer of header information that might identify hosts behind the firewall. The Via header field allows intermediaries to replace sensitive machine names with pseudonyms.

12.11. Browser Fingerprinting

Browser fingerprinting is a set of techniques for identifying a specific user agent over time through its unique set of characteristics. These characteristics might include information related to its TCP behavior, feature capabilities, and scripting environment, though of particular interest here is the set of unique characteristics that might be communicated via HTTP. Fingerprinting is considered a privacy concern because it enables tracking of a user agent's behavior over time without the corresponding controls that the user might have over other forms of data collection (e.g., cookies). Many general-purpose user agents (i.e., Web browsers) have taken steps to reduce their fingerprints.

There are a number of request header fields that might reveal information to servers that is sufficiently unique to enable fingerprinting. The From header field is the most obvious, though it is expected that From will only be sent when self-identification is desired by the user. Likewise, Cookie header fields are deliberately designed to enable re-identification, so fingerprinting concerns only apply to situations where cookies are disabled or restricted by the user agent's configuration.

The User-Agent header field might contain enough information to uniquely identify a specific device, usually when combined with other characteristics, particularly if the user agent sends excessive details about the user's system or extensions. However, the source of unique information that is least expected by users is proactive negotiation (Section 8.4), including the Accept, Accept-Charset, Accept-Encoding, and Accept-Language header fields.

In addition to the fingerprinting concern, detailed use of the Accept-Language header field can reveal information the user might

consider to be of a private nature. For example, understanding a given language set might be strongly correlated to membership in a particular ethnic group. An approach that limits such loss of privacy would be for a user agent to omit the sending of Accept-Language except for sites that have been whitelisted, perhaps via interaction after detecting a Vary header field that indicates language negotiation might be useful.

In environments where proxies are used to enhance privacy, user agents ought to be conservative in sending proactive negotiation header fields. General-purpose user agents that provide a high degree of header field configurability ought to inform users about the loss of privacy that might result if too much detail is provided. As an extreme privacy measure, proxies could filter the proactive negotiation header fields in relayed requests.

12.12. Validator Retention

The validators defined by this specification are not intended to ensure the validity of a representation, guard against malicious changes, or detect man-in-the-middle attacks. At best, they enable more efficient cache updates and optimistic concurrent writes when all participants are behaving nicely. At worst, the conditions will fail and the client will receive a response that is no more harmful than an HTTP exchange without conditional requests.

An entity-tag can be abused in ways that create privacy risks. For example, a site might deliberately construct a semantically invalid entity-tag that is unique to the user or user agent, send it in a cacheable response with a long freshness time, and then read that entity-tag in later conditional requests as a means of re-identifying that user or user agent. Such an identifying tag would become a persistent identifier for as long as the user agent retained the original cache entry. User agents that cache representations ought to ensure that the cache is cleared or replaced whenever the user performs privacy-maintaining actions, such as clearing stored cookies or changing to a private browsing mode.

12.13. Denial-of-Service Attacks Using Range

Unconstrained multiple range requests are susceptible to denial-of-service attacks because the effort required to request many overlapping ranges of the same data is tiny compared to the time, memory, and bandwidth consumed by attempting to serve the requested data in many parts. Servers ought to ignore, coalesce, or reject egregious range requests, such as requests for more than two overlapping ranges or for many small ranges in a single set, particularly when the ranges are requested out of order for no

apparent reason. Multipart range requests are not designed to support random access.

12.14. Authentication Considerations

Everything about the topic of HTTP authentication is a security consideration, so the list of considerations below is not exhaustive. Furthermore, it is limited to security considerations regarding the authentication framework, in general, rather than discussing all of the potential considerations for specific authentication schemes (which ought to be documented in the specifications that define those schemes). Various organizations maintain topical information and links to current research on Web application security (e.g., [OWASP]), including common pitfalls for implementing and using the authentication schemes found in practice.

12.14.1. Confidentiality of Credentials

The HTTP authentication framework does not define a single mechanism for maintaining the confidentiality of credentials; instead, each authentication scheme defines how the credentials are encoded prior to transmission. While this provides flexibility for the development of future authentication schemes, it is inadequate for the protection of existing schemes that provide no confidentiality on their own, or that do not sufficiently protect against replay attacks. Furthermore, if the server expects credentials that are specific to each individual user, the exchange of those credentials will have the effect of identifying that user even if the content within credentials remains confidential.

HTTP depends on the security properties of the underlying transport- or session-level connection to provide confidential transmission of header fields. In other words, if a server limits access to authenticated users using this framework, the server needs to ensure that the connection is properly secured in accordance with the nature of the authentication scheme used. For example, services that depend on individual user authentication often require a connection to be secured with TLS ("Transport Layer Security", [RFC5246]) prior to exchanging any credentials.

12.14.2. Credentials and Idle Clients

Existing HTTP clients and user agents typically retain authentication information indefinitely. HTTP does not provide a mechanism for the origin server to direct clients to discard these cached credentials, since the protocol has no awareness of how credentials are obtained or managed by the user agent. The mechanisms for expiring or

revoking credentials can be specified as part of an authentication scheme definition.

Circumstances under which credential caching can interfere with the application's security model include but are not limited to:

- Clients that have been idle for an extended period, following which the server might wish to cause the client to re-prompt the user for credentials.

- Applications that include a session termination indication (such as a "logout" or "commit" button on a page) after which the server side of the application "knows" that there is no further reason for the client to retain the credentials.

User agents that cache credentials are encouraged to provide a readily accessible mechanism for discarding cached credentials under user control.

12.14.3. Protection Spaces

Authentication schemes that solely rely on the "realm" mechanism for establishing a protection space will expose credentials to all resources on an origin server. Clients that have successfully made authenticated requests with a resource can use the same authentication credentials for other resources on the same origin server. This makes it possible for a different resource to harvest authentication credentials for other resources.

This is of particular concern when an origin server hosts resources for multiple parties under the same canonical root URI (Section 8.5.2). Possible mitigation strategies include restricting direct access to authentication credentials (i.e., not making the content of the Authorization request header field available), and separating protection spaces by using a different host name (or port number) for each party.

13. IANA Considerations

This section is to be removed before publishing as an RFC.

The change controller for the following registrations is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

13.1. URI Scheme Registration

Please update the registry of URI Schemes [BCP115] at <https://www.iana.org/assignments/uri-schemes/> with the permanent schemes listed in the first table of Section 2.5.

13.2. Method Registration

Please update the "Hypertext Transfer Protocol (HTTP) Method Registry" at <https://www.iana.org/assignments/http-methods> with the registration procedure of Section 7.4.1 and the method names summarized in the table of Section 7.2.

13.3. Status Code Registration

Please update the "Hypertext Transfer Protocol (HTTP) Status Code Registry" at <https://www.iana.org/assignments/http-status-codes> with the registration procedure of Section 9.7.1 and the status code values summarized in the table of Section 9.1.

13.4. Header Field Registration

Please update the "Message Headers" registry of "Permanent Message Header Field Names" at <https://www.iana.org/assignments/message-headers> with the header field names listed in the table of Section 4.1.

13.5. Authentication Scheme Registration

Please update the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" at <https://www.iana.org/assignments/http-authschemes> with the registration procedure of Section 8.5.5.1. No authentication schemes are defined in this document.

13.6. Content Coding Registration

Please update the "HTTP Content Coding Registry" at <https://www.iana.org/assignments/http-parameters/> with the registration procedure of Section 6.1.2.4.1 and the content coding names summarized in the table of Section 6.1.2.

13.7. Range Unit Registration

Please update the "HTTP Range Unit Registry" at <https://www.iana.org/assignments/http-parameters/> with the registration procedure of Section 6.1.4.3 and the range unit names summarized in the table of Section 6.1.4.

13.8. Media Type Registration

Please update the "Media Types" registry at <https://www.iana.org/assignments/media-types> with the registration information in Section 6.3.4 for the media type "multipart/byteranges".

14. References

14.1. Normative References

[Caching] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", draft-ietf-httpbis-cache-01 (work in progress), May 2018.

[Messaging] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1 Messaging", draft-ietf-httpbis-messaging-01 (work in progress), May 2018.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1950] Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <https://www.rfc-editor.org/info/rfc1950>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/info/rfc1951>.

[RFC1952] Deutsch, P., Gailly, J-L., Adler, M., Deutsch, L., and G. Randers-Pehrson, "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc1952>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4647] Phillips, A., Ed. and M. Davis, Ed., "Matching of Language Tags", BCP 47, RFC 4647, DOI 10.17487/RFC4647, September 2006, <https://www.rfc-editor.org/info/rfc4647>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, DOI 10.17487/RFC6365, September 2011, <https://www.rfc-editor.org/info/rfc6365>.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[Welch] Welch, T., "A Technique for High-Performance Data Compression", IEEE Computer 17(6), June 1984.

14.2. Informative References

[BCP115] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 115, RFC 4395, February 2006, <https://www.rfc-editor.org/info/bcp115>.

[BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013, <https://www.rfc-editor.org/info/bcp13>.

[BCP178] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, June 2012, <https://www.rfc-editor.org/info/bcp178>.

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004, <https://www.rfc-editor.org/info/bcp90>.

[Georgiev]
Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software", In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12), pp. 38-49, October 2012, <http://doi.acm.org/10.1145/2382196.2382204>.

[ISO-8859-1]
International Organization for Standardization, "Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1", ISO/IEC 8859-1:1998, 1998.

[Kri2001] Kristol, D., "HTTP Cookies: Standards, Privacy, and Politics", ACM Transactions on Internet Technology 1(2), November 2001, <http://arxiv.org/abs/cs.SE/0105018>.

[OWASP] van der Stock, A., Ed., "A Guide to Building Secure Web Applications and Web Services", The Open Web Application Security Project (OWASP) 2.0.1, July 2005, <https://www.owasp.org/>.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Doctoral Dissertation, University of California, Irvine, September 2000, <https://roy.gbiv.com/pubs/dissertation/top.htm>.

[RFC1919] Chatel, M., "Classical versus Transparent IP Proxies", RFC 1919, DOI 10.17487/RFC1919, March 1996, <https://www.rfc-editor.org/info/rfc1919>.

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, DOI 10.17487/RFC1945, May 1996, <https://www.rfc-editor.org/info/rfc1945>.

[RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, DOI 10.17487/RFC2047, November 1996, <https://www.rfc-editor.org/info/rfc2047>.

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, DOI 10.17487/RFC2068, January 1997, <https://www.rfc-editor.org/info/rfc2068>.

[RFC2145] Mogul, J., Fielding, R., Gettys, J., and H. Nielsen, "Use and Interpretation of HTTP Version Numbers", RFC 2145, DOI 10.17487/RFC2145, May 1997, <https://www.rfc-editor.org/info/rfc2145>.

[RFC2295] Holtman, K. and A. Mutz, "Transparent Content Negotiation in HTTP", RFC 2295, DOI 10.17487/RFC2295, March 1998, <https://www.rfc-editor.org/info/rfc2295>.

[RFC2388] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 2388, DOI 10.17487/RFC2388, August 1998, <https://www.rfc-editor.org/info/rfc2388>.

[RFC2557] Palme, F., Hopmann, A., Shelness, N., and E. Stefferud, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, DOI 10.17487/RFC2557, March 1999, <https://www.rfc-editor.org/info/rfc2557>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc2616>.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <https://www.rfc-editor.org/info/rfc2617>.

[RFC2774] Frystyk, H., Leach, P., and S. Lawrence, "An HTTP Extension Framework", RFC 2774, DOI 10.17487/RFC2774, February 2000, <https://www.rfc-editor.org/info/rfc2774>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC2978] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, DOI 10.17487/RFC2978, October 2000, <https://www.rfc-editor.org/info/rfc2978>.

[RFC3040] Cooper, I., Melve, I., and G. Tomlinson, "Internet Web Replication and Caching Taxonomy", RFC 3040, DOI 10.17487/RFC3040, January 2001, <https://www.rfc-editor.org/info/rfc3040>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC4559] Jaganathan, K., Zhu, L., and J. Brezak, "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, DOI 10.17487/RFC4559, June 2006, <https://www.rfc-editor.org/info/rfc4559>.

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <https://www.rfc-editor.org/info/rfc4918>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <https://www.rfc-editor.org/info/rfc5226>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5322] Resnick, P., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/info/rfc5789>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC5987] Reschke, J., "Character Set and Language Encoding for Hypertext Transfer Protocol (HTTP) Header Field Parameters", RFC 5987, DOI 10.17487/RFC5987, August 2010, <https://www.rfc-editor.org/info/rfc5987>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://www.rfc-editor.org/info/rfc5988>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <https://www.rfc-editor.org/info/rfc6585>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <https://www.rfc-editor.org/info/rfc7232>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <https://www.rfc-editor.org/info/rfc7233>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

[RFC7238] Reschke, J., "The Hypertext Transfer Protocol Status Code 308 (Permanent Redirect)", RFC 7238, DOI 10.17487/RFC7238, June 2014, <https://www.rfc-editor.org/info/rfc7238>.

[RFC7616] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015, <https://www.rfc-editor.org/info/rfc7616>.

[RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <https://www.rfc-editor.org/info/rfc7617>.

Appendix A. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 11.

```
Accept = [ ( "," / ( media-range [ accept-params ] ) ) *( OWS "," [
 OWS ( media-range [ accept-params ] ) ] ) ]
Accept-Charset = *( "," OWS ) ( ( charset / "*" ) [ weight ] ) *( OWS
 "," [ OWS ( ( charset / "*" ) [ weight ] ) ] )
Accept-Encoding = [ ( "," / ( codings [ weight ] ) ) *( OWS "," [ OWS
 ( codings [ weight ] ) ] ) ]
Accept-Language = *( "," OWS ) ( language-range [ weight ] ) *( OWS
 "," [ OWS ( language-range [ weight ] ) ] )
Accept-Ranges = acceptable-ranges
Allow = [ ( "," / method ) *( OWS "," [ OWS method ] ) ]
Authorization = credentials

BWS = OWS

Content-Encoding = *( "," OWS ) content-coding *( OWS "," [ OWS
 content-coding ] )
Content-Language = *( "," OWS ) language-tag *( OWS "," [ OWS
 language-tag ] )
Content-Length = 1*DIGIT
Content-Location = absolute-URI / partial-URI
Content-Range = byte-content-range / other-content-range
Content-Type = media-type

Date = HTTP-date

ETag = entity-tag
Expect = "100-continue"

From = mailbox

GMT = %x47.4D.54 ; GMT

HTTP-date = IMF-fixdate / obs-date
Host = uri-host [ ":" port ]

IMF-fixdate = day-name "," SP date1 SP time-of-day SP GMT
If-Match = "*" / ( *( "," OWS ) entity-tag *( OWS "," [ OWS
 entity-tag ] ) )
If-Modified-Since = HTTP-date
If-None-Match = "*" / ( *( "," OWS ) entity-tag *( OWS "," [ OWS
 entity-tag ] ) )
If-Range = entity-tag / HTTP-date
If-Unmodified-Since = HTTP-date
```

```
   Last-Modified = HTTP-date
   Location = URI-reference

   Max-Forwards = 1*DIGIT

   OWS = *( SP / HTAB )

   Proxy-Authenticate = *( "," OWS ) challenge *( OWS "," [ OWS
    challenge ] )
   Proxy-Authorization = credentials

   RWS = 1*( SP / HTAB )
   Range = byte-ranges-specifier / other-ranges-specifier
   Referer = absolute-URI / partial-URI
   Retry-After = HTTP-date / delay-seconds

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

   Trailer = *( "," OWS ) field-name *( OWS "," [ OWS field-name ] )

   URI-reference = <URI-reference, see [RFC3986], Section 4.1>
   User-Agent = product *( RWS ( product / comment ) )

   Vary = "*" / ( *( "," OWS ) field-name *( OWS "," [ OWS field-name ]
    ) )
   Via = *( "," OWS ) ( received-protocol RWS received-by [ RWS comment
    ] ) *( OWS "," [ OWS ( received-protocol RWS received-by [ RWS
    comment ] ) ] )

   WWW-Authenticate = *( "," OWS ) challenge *( OWS "," [ OWS challenge
    ] )

   absolute-URI = <absolute-URI, see [RFC3986], Section 4.3>
   absolute-path = 1*( "/" segment )
   accept-ext = OWS ";" OWS token [ "=" ( token / quoted-string ) ]
   accept-params = weight *accept-ext
   acceptable-ranges = ( *( "," OWS ) range-unit *( OWS "," [ OWS
    range-unit ] ) ) / "none"
   asctime-date = day-name SP date3 SP time-of-day SP year
   auth-param = token BWS "=" BWS ( token / quoted-string )
   auth-scheme = token
   authority = <authority, see [RFC3986], Section 3.2>

   byte-content-range = bytes-unit SP ( byte-range-resp /
    unsatisfied-range )
   byte-range = first-byte-pos "-" last-byte-pos
   byte-range-resp = byte-range "/" ( complete-length / "*" )
   byte-range-set = *( "," OWS ) ( byte-range-spec /
```

```
    suffix-byte-range-spec ) *( OWS "," [ OWS ( byte-range-spec /
    suffix-byte-range-spec ) ] )
   byte-range-spec = first-byte-pos "-" [ last-byte-pos ]
   byte-ranges-specifier = bytes-unit "=" byte-range-set
   bytes-unit = "bytes"

   challenge = auth-scheme [ 1*SP ( token68 / [ ( "," / auth-param ) *(
    OWS "," [ OWS auth-param ] ) ] ) ]
   charset = token
   codings = content-coding / "identity" / "*"
   comment = "(" *( ctext / quoted-pair / comment ) ")"
   complete-length = 1*DIGIT
   content-coding = token
   credentials = auth-scheme [ 1*SP ( token68 / [ ( "," / auth-param )
    *( OWS "," [ OWS auth-param ] ) ] ) ]
   ctext = HTAB / SP / %x21-27 ; '!'-'''
    / %x2A-5B ; '*'-'['
    / %x5D-7E ; ']'-'~'
    / obs-text

   date1 = day SP month SP year
   date2 = day "-" month "-" 2DIGIT
   date3 = month SP ( 2DIGIT / ( SP DIGIT ) )
   day = 2DIGIT
   day-name = %x4D.6F.6E ; Mon
    / %x54.75.65 ; Tue
    / %x57.65.64 ; Wed
    / %x54.68.75 ; Thu
    / %x46.72.69 ; Fri
    / %x53.61.74 ; Sat
    / %x53.75.6E ; Sun
   day-name-l = %x4D.6F.6E.64.61.79 ; Monday
    / %x54.75.65.73.64.61.79 ; Tuesday
    / %x57.65.64.6E.65.73.64.61.79 ; Wednesday
    / %x54.68.75.72.73.64.61.79 ; Thursday
    / %x46.72.69.64.61.79 ; Friday
    / %x53.61.74.75.72.64.61.79 ; Saturday
    / %x53.75.6E.64.61.79 ; Sunday
   delay-seconds = 1*DIGIT

   entity-tag = [ weak ] opaque-tag
   etagc = "!" / %x23-7E ; '#'-'~'
    / obs-text

   field-content = field-vchar [ 1*( SP / HTAB ) field-vchar ]
   field-name = token
   field-value = *( field-content / obs-fold )
   field-vchar = VCHAR / obs-text
```

```
   first-byte-pos = 1*DIGIT
   fragment = <fragment, see [RFC3986], Section 3.5>

   hour = 2DIGIT
   http-URI = "http://" authority path-abempty [ "?" query ] [ "#"
    fragment ]
   https-URI = "https://" authority path-abempty [ "?" query ] [ "#"
    fragment ]

   language-range = <language-range, see [RFC4647], Section 2.1>
   language-tag = <Language-Tag, see [RFC5646], Section 2.1>
   last-byte-pos = 1*DIGIT

   mailbox = <mailbox, see [RFC5322], Section 3.4>
   media-range = ( "*/*" / ( type "/*" ) / ( type "/" subtype ) ) *( OWS
    ";" OWS parameter )
   media-type = type "/" subtype *( OWS ";" OWS parameter )
   method = token
   minute = 2DIGIT
   month = %x4A.61.6E ; Jan
    / %x46.65.62 ; Feb
    / %x4D.61.72 ; Mar
    / %x41.70.72 ; Apr
    / %x4D.61.79 ; May
    / %x4A.75.6E ; Jun
    / %x4A.75.6C ; Jul
    / %x41.75.67 ; Aug
    / %x53.65.70 ; Sep
    / %x4F.63.74 ; Oct
    / %x4E.6F.76 ; Nov
    / %x44.65.63 ; Dec

   obs-date = rfc850-date / asctime-date
   obs-fold = <obs-fold, see [Messaging], Section 5.2>
   obs-text = %x80-FF
   opaque-tag = DQUOTE *etagc DQUOTE
   other-content-range = other-range-unit SP other-range-resp
   other-range-resp = *CHAR
   other-range-set = 1*VCHAR
   other-range-unit = token
   other-ranges-specifier = other-range-unit "=" other-range-set

   parameter = token "=" ( token / quoted-string )
   partial-URI = relative-part [ "?" query ]
   path-abempty = <path-abempty, see [RFC3986], Section 3.3>
   port = <port, see [RFC3986], Section 3.2.3>
   product = token [ "/" product-version ]
   product-version = token
```

```
   protocol-name = <protocol-name, see [Messaging], Section 9.7>
   protocol-version = <protocol-version, see [Messaging], Section 9.7>
   pseudonym = token

   qdtext = HTAB / SP / "!" / %x23-5B ; '#'-'['
    / %x5D-7E ; ']'-'~'
    / obs-text
   query = <query, see [RFC3986], Section 3.4>
   quoted-pair = "\" ( HTAB / SP / VCHAR / obs-text )
   quoted-string = DQUOTE *( qdtext / quoted-pair ) DQUOTE
   qvalue = ( "0" [ "." *3DIGIT ] ) / ( "1" [ "." *3"0" ] )

   range-unit = bytes-unit / other-range-unit
   received-by = ( uri-host [ ":" port ] ) / pseudonym
   received-protocol = [ protocol-name "/" ] protocol-version
   relative-part = <relative-part, see [RFC3986], Section 4.2>
   request-target = <request-target, see [Messaging], Section 3.2>
   rfc850-date = day-name-l "," SP date2 SP time-of-day SP GMT

   second = 2DIGIT
   segment = <segment, see [RFC3986], Section 3.3>
   subtype = token
   suffix-byte-range-spec = "-" suffix-length
   suffix-length = 1*DIGIT

   tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "." /
    "^" / "_" / "`" / "|" / "~" / DIGIT / ALPHA
   time-of-day = hour ":" minute ":" second
   token = 1*tchar
   token68 = 1*( ALPHA / DIGIT / "-" / "." / "_" / "~" / "+" / "/" )
    *"="
   type = token

   unsatisfied-range = "*/" complete-length
   uri-host = <host, see [RFC3986], Section 3.2.2>

   weak = %x57.2F ; W/
   weight = OWS ";" OWS "q=" qvalue

   year = 4DIGIT
```

Appendix B. Changes from RFC 7230

Most of the sections introducing HTTP's design goals, history, architecture, conformance criteria, protocol versioning, URIs, message routing, and header field values have been moved here (without substantive change).

Appendix C. Changes from RFC 7231

None yet.

Appendix D. Changes from RFC 7232

None yet.

Appendix E. Changes from RFC 7233

None yet.

Appendix F. Changes from RFC 7235

None yet.

Appendix G. Change Log

This section is to be removed before publishing as an RFC.

G.1. Between RFC723x and draft 00

The changes were purely editorial:

- Change boilerplate and abstract to indicate the "draft" status, and update references to ancestor specifications.
- Remove version "1.1" from document title, indicating that this specification applies to all HTTP versions.
- Adjust historical notes.
- Update links to sibling specifications.
- Replace sections listing changes from RFC 2616 by new empty sections referring to RFC 723x.
- Remove acknowledgements specific to RFC 723x.
- Move "Acknowledgements" to the very end and make them unnumbered.

G.2. Since draft-ietf-httpbis-semantics-00

The changes in this draft are editorial, with respect to HTTP as a whole, to merge core HTTP semantics into this document:

- Merged introduction, architecture, conformance, and ABNF extensions from RFC 7230 (Messaging).

- Rearranged architecture to extract conformance, http(s) schemes, and protocol versioning into a separate major section.

- Moved discussion of MIME differences to [Messaging] since that is primarily concerned with transforming 1.1 messages.

- Merged entire content of RFC 7232 (Conditional Requests).

- Merged entire content of RFC 7233 (Range Requests).

- Merged entire content of RFC 7235 (Auth Framework).

- Moved all extensibility tips, registration procedures, and registry tables from the IANA considerations to normative sections, reducing the IANA considerations to just instructions that will be removed prior to publication as an RFC.

Index

Acknowledgments

This edition of the HTTP specification builds on the many contributions that went into RFC 1945, RFC 2068, RFC 2145, and RFC 2616, including substantial contributions made by the previous authors, editors, and Working Group Chairs: Tim Berners-Lee, Ari Luotonen, Roy T. Fielding, Henrik Frystyk Nielsen, Jim Gettys, Jeffrey C. Mogul, Larry Masinter, Paul J. Leach, and Yves Lafon.

See Section 10 of [RFC7230] for further acknowledgements from prior revisions.

In addition, this document has reincorporated the HTTP Authentication Framework, previously defined in RFC 7235 and RFC 2617. We thank John Franks, Phillip M. Hallam-Baker, Jeffery L. Hostetler, Scott D. Lawrence, Paul J. Leach, Ari Luotonen, and Lawrence C. Stewart for their work on that specification. See Section 6 of [RFC2617] for further acknowledgements.

[[newacks: New acks to be added here.]]

Authors' Addresses

Roy T. Fielding (editor)
Adobe
345 Park Ave
San Jose, CA 95110
USA

EMail: fielding@gbiv.com
URI: https://roy.gbiv.com/

Mark Nottingham (editor)
Fastly

EMail: mnot@mnot.net
URI: https://www.mnot.net/

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: https://greenbytes.de/tech/webdav/