Internet Engineering Task Force (IETF)
Request for Comments: 9110
Obsoletes: 2818, 7230, 7231, 7232, 7233, 7235, 7538, 7615, 7694
STD: 97
Updates: 3864
Category: Standards Track
ISSN: 2070-1721

R. Fielding, Editor
Adobe
M. Nottingham, Editor
Fastly
J. Reschke, Editor
greenbytes

June 2022

HTTP Semantics

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document describes the overall architecture of HTTP, establishes common terminology, and defines aspects of the protocol that are shared by all versions. In this definition are core protocol elements, extensibility mechanisms, and the "http" and "https" Uniform Resource Identifier (URI) schemes.

This document updates RFC 3864 and obsoletes RFCs 2818, 7231, 7232, 7233, 7235, 7538, 7615, 7694, and portions of 7230.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841[1].

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9110[2].

Copyright Notice

[1] https://www.rfc-editor.org/rfc/rfc7841.html#section-2
[2] https://www.rfc-editor.org/info/rfc9110
[3] https://trustee.ietf.org/license-info

Table of Contents

1 Introduction....11
1.1 Purpose....11
1.2 History and Evolution....11
1.3 Core Semantics....11
1.4 Specifications Obsoleted by This Document....12

2 Conformance....13
2.1 Syntax Notation....13
2.2 Requirements Notation....13
2.3 Length Requirements....14
2.4 Error Handling....14
2.5 Protocol Version....14

3 Terminology and Core Concepts....16
3.1 Resources....16
3.2 Representations....16
3.3 Connections, Clients, and Servers....16
3.4 Messages....17
3.5 User Agents....17
3.6 Origin Server....17
3.7 Intermediaries....18
3.8 Caches....19
3.9 Example Message Exchange....19

4 Identifiers in HTTP....21
4.1 URI References....21
4.2 HTTP-Related URI Schemes....21
4.2.1 http URI Scheme....21
4.2.2 https URI Scheme....22
4.2.3 http(s) Normalization and Comparison....22
4.2.4 Deprecation of userinfo in http(s) URIs....23
4.2.5 http(s) References with Fragment Identifiers....23
4.3 Authoritative Access....23
4.3.1 URI Origin....24
4.3.2 http Origins....24
4.3.3 https Origins....24
4.3.4 https Certificate Verification....25
4.3.5 IP-ID Reference Identity....26

5 Fields....27
5.1 Field Names....27
5.2 Field Lines and Combined Field Value....27

5.3 Field Order....27
5.4 Field Limits....28
5.5 Field Values....28
5.6 Common Rules for Defining Field Values....29
5.6.1 Lists (#rule ABNF Extension)....29
5.6.1.1 Sender Requirements....29
5.6.1.2 Recipient Requirements....30
5.6.2 Tokens....30
5.6.3 Whitespace....30
5.6.4 Quoted Strings....31
5.6.5 Comments....31
5.6.6 Parameters....31
5.6.7 Date/Time Formats....32

6 Message Abstraction....35

6.1 Framing and Completeness....35
6.2 Control Data....35
6.3 Header Fields....36
6.4 Content....36
6.4.1 Content Semantics....37
6.4.2 Identifying Content....37
6.5 Trailer Fields....38
6.5.1 Limitations on Use of Trailers....38
6.5.2 Processing Trailer Fields....38
6.6 Message Metadata....39
6.6.1 Date....39
6.6.2 Trailer....39

7 Routing HTTP Messages....41

7.1 Determining the Target Resource....41
7.2 Host and :authority....41
7.3 Routing Inbound Requests....42
7.3.1 To a Cache....42
7.3.2 To a Proxy....42
7.3.3 To the Origin....42
7.4 Rejecting Misdirected Requests....42
7.5 Response Correlation....43
7.6 Message Forwarding....43
7.6.1 Connection....43
7.6.2 Max-Forwards....44
7.6.3 Via....44
7.7 Message Transformations....45
7.8 Upgrade....46

8 Representation Data and Metadata....48

8.1 Representation Data....48

8.2 Representation Metadata ... 48
8.3 Content-Type ... 48
8.3.1 Media Type ... 48
8.3.2 Charset ... 49
8.3.3 Multipart Types ... 49
8.4 Content-Encoding ... 49
8.4.1 Content Codings ... 50
8.4.1.1 Compress Coding ... 50
8.4.1.2 Deflate Coding ... 50
8.4.1.3 Gzip Coding ... 51
8.5 Content-Language ... 51
8.5.1 Language Tags ... 51
8.6 Content-Length ... 52
8.7 Content-Location ... 53
8.8 Validator Fields ... 54
8.8.1 Weak versus Strong ... 54
8.8.2 Last-Modified ... 55
8.8.2.1 Generation ... 55
8.8.2.2 Comparison ... 56
8.8.3 ETag ... 56
8.8.3.1 Generation ... 57
8.8.3.2 Comparison ... 57
8.8.3.3 Example: Entity Tags Varying on Content-Negotiated Resources ... 58

9 Methods ... 59

9.1 Overview ... 59
9.2 Common Method Properties ... 60
9.2.1 Safe Methods ... 60
9.2.2 Idempotent Methods ... 60
9.2.3 Methods and Caching ... 61
9.3 Method Definitions ... 61
9.3.1 GET ... 61
9.3.2 HEAD ... 62
9.3.3 POST ... 62
9.3.4 PUT ... 63
9.3.5 DELETE ... 64
9.3.6 CONNECT ... 65
9.3.7 OPTIONS ... 66
9.3.8 TRACE ... 66

10 Message Context ... 68

10.1 Request Context Fields ... 68
10.1.1 Expect ... 68
10.1.2 From ... 69
10.1.3 Referer ... 69
10.1.4 TE ... 70
10.1.5 User-Agent ... 71
10.2 Response Context Fields ... 71
10.2.1 Allow ... 71

10.2.2 Location ... 72
10.2.3 Retry-After ... 72
10.2.4 Server ... 73

11 HTTP Authentication ... 74

11.1 Authentication Scheme ... 74
11.2 Authentication Parameters ... 74
11.3 Challenge and Response ... 74
11.4 Credentials ... 75
11.5 Establishing a Protection Space (Realm) ... 75
11.6 Authenticating Users to Origin Servers ... 76
11.6.1 WWW-Authenticate ... 76
11.6.2 Authorization ... 76
11.6.3 Authentication-Info ... 76
11.7 Authenticating Clients to Proxies ... 77
11.7.1 Proxy-Authenticate ... 77
11.7.2 Proxy-Authorization ... 77
11.7.3 Proxy-Authentication-Info ... 77

12 Content Negotiation ... 79

12.1 Proactive Negotiation ... 79
12.2 Reactive Negotiation ... 80
12.3 Request Content Negotiation ... 80
12.4 Content Negotiation Field Features ... 80
12.4.1 Absence ... 80
12.4.2 Quality Values ... 81
12.4.3 Wildcard Values ... 81
12.5 Content Negotiation Fields ... 81
12.5.1 Accept ... 81
12.5.2 Accept-Charset ... 82
12.5.3 Accept-Encoding ... 83
12.5.4 Accept-Language ... 84
12.5.5 Vary ... 85

13 Conditional Requests ... 86

13.1 Preconditions ... 86
13.1.1 If-Match ... 86
13.1.2 If-None-Match ... 87
13.1.3 If-Modified-Since ... 88
13.1.4 If-Unmodified-Since ... 89
13.1.5 If-Range ... 90
13.2 Evaluation of Preconditions ... 91
13.2.1 When to Evaluate ... 91
13.2.2 Precedence of Preconditions ... 92

14 Range Requests ... 93

14.1 Range Units ... 93
14.1.1 Range Specifiers ... 93

14.1.2 Byte Ranges ... 94
14.2 Range ... 95
14.3 Accept-Ranges ... 96
14.4 Content-Range ... 96
14.5 Partial PUT ... 98
14.6 Media Type multipart/byteranges ... 98

15 Status Codes ... 101

15.1 Overview of Status Codes ... 101
15.2 Informational 1xx ... 101
15.2.1 100 Continue ... 102
15.2.2 101 Switching Protocols ... 102
15.3 Successful 2xx ... 102
15.3.1 200 OK ... 102
15.3.2 201 Created ... 103
15.3.3 202 Accepted ... 103
15.3.4 203 Non-Authoritative Information ... 103
15.3.5 204 No Content ... 103
15.3.6 205 Reset Content ... 104
15.3.7 206 Partial Content ... 104
15.3.7.1 Single Part ... 104
15.3.7.2 Multiple Parts ... 105
15.3.7.3 Combining Parts ... 106
15.4 Redirection 3xx ... 106
15.4.1 300 Multiple Choices ... 107
15.4.2 301 Moved Permanently ... 107
15.4.3 302 Found ... 108
15.4.4 303 See Other ... 108
15.4.5 304 Not Modified ... 108
15.4.6 305 Use Proxy ... 109
15.4.7 306 (Unused) ... 109
15.4.8 307 Temporary Redirect ... 109
15.4.9 308 Permanent Redirect ... 109
15.5 Client Error 4xx ... 110
15.5.1 400 Bad Request ... 110
15.5.2 401 Unauthorized ... 110
15.5.3 402 Payment Required ... 110
15.5.4 403 Forbidden ... 110
15.5.5 404 Not Found ... 110
15.5.6 405 Method Not Allowed ... 110
15.5.7 406 Not Acceptable ... 111
15.5.8 407 Proxy Authentication Required ... 111
15.5.9 408 Request Timeout ... 111
15.5.10 409 Conflict ... 111
15.5.11 410 Gone ... 111
15.5.12 411 Length Required ... 111
15.5.13 412 Precondition Failed ... 112
15.5.14 413 Content Too Large ... 112
15.5.15 414 URI Too Long ... 112
15.5.16 415 Unsupported Media Type ... 112

15.5.17 416 Range Not Satisfiable ... 112
15.5.18 417 Expectation Failed ... 113
15.5.19 418 (Unused) ... 113
15.5.20 421 Misdirected Request ... 113
15.5.21 422 Unprocessable Content ... 113
15.5.22 426 Upgrade Required ... 113
15.6 Server Error 5xx ... 114
15.6.1 500 Internal Server Error ... 114
15.6.2 501 Not Implemented ... 114
15.6.3 502 Bad Gateway ... 114
15.6.4 503 Service Unavailable ... 114
15.6.5 504 Gateway Timeout ... 114
15.6.6 505 HTTP Version Not Supported ... 114
16 Extending HTTP ... 116
16.1 Method Extensibility ... 116
16.1.1 Method Registry ... 116
16.1.2 Considerations for New Methods ... 116
16.2 Status Code Extensibility ... 117
16.2.1 Status Code Registry ... 117
16.2.2 Considerations for New Status Codes ... 117
16.3 Field Extensibility ... 117
16.3.1 Field Name Registry ... 118
16.3.2 Considerations for New Fields ... 118
16.3.2.1 Considerations for New Field Names ... 119
16.3.2.2 Considerations for New Field Values ... 119
16.4 Authentication Scheme Extensibility ... 120
16.4.1 Authentication Scheme Registry ... 120
16.4.2 Considerations for New Authentication Schemes ... 120
16.5 Range Unit Extensibility ... 121
16.5.1 Range Unit Registry ... 121
16.5.2 Considerations for New Range Units ... 121
16.6 Content Coding Extensibility ... 121
16.6.1 Content Coding Registry ... 121
16.6.2 Considerations for New Content Codings ... 121
16.7 Upgrade Token Registry ... 122
17 Security Considerations ... 123
17.1 Establishing Authority ... 123
17.2 Risks of Intermediaries ... 123
17.3 Attacks Based on File and Path Names ... 124
17.4 Attacks Based on Command, Code, or Query Injection ... 124
17.5 Attacks via Protocol Element Length ... 124
17.6 Attacks Using Shared-Dictionary Compression ... 125
17.7 Disclosure of Personal Information ... 125
17.8 Privacy of Server Log Information ... 125
17.9 Disclosure of Sensitive Information in URIs ... 125

17.10 Application Handling of Field Names ... 126
17.11 Disclosure of Fragment after Redirects ... 126
17.12 Disclosure of Product Information ... 126
17.13 Browser Fingerprinting ... 127
17.14 Validator Retention ... 127
17.15 Denial-of-Service Attacks Using Range ... 127
17.16 Authentication Considerations ... 128
17.16.1 Confidentiality of Credentials ... 128
17.16.2 Credentials and Idle Clients ... 128
17.16.3 Protection Spaces ... 128
17.16.4 Additional Response Fields ... 129

18 IANA Considerations ... 130
18.1 URI Scheme Registration ... 130
18.2 Method Registration ... 130
18.3 Status Code Registration ... 130
18.4 Field Name Registration ... 131
18.5 Authentication Scheme Registration ... 133
18.6 Content Coding Registration ... 133
18.7 Range Unit Registration ... 133
18.8 Media Type Registration ... 133
18.9 Port Registration ... 133
18.10 Upgrade Token Registration ... 134

19 References ... 135
19.1 Normative References ... 135
19.2 Informative References ... 136

Appendix A Collected ABNF ... 140

Appendix B Changes from Previous RFCs ... 142
B.1 Changes from RFC 2818 ... 142
B.2 Changes from RFC 7230 ... 142
B.3 Changes from RFC 7231 ... 142
B.4 Changes from RFC 7232 ... 143
B.5 Changes from RFC 7233 ... 143
B.6 Changes from RFC 7235 ... 144
B.7 Changes from RFC 7538 ... 144
B.8 Changes from RFC 7615 ... 144
B.9 Changes from RFC 7694 ... 144

Index ... 146

Authors' Addresses **152**

1. Introduction

1.1. Purpose

The Hypertext Transfer Protocol (HTTP) is a family of stateless, application-level, request/response protocols that share a generic interface, extensible semantics, and self-descriptive messages to enable flexible interaction with network-based hypertext information systems.

HTTP hides 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: a request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. This allows general-purpose implementations to be used effectively in many different contexts, reduces interaction complexity, and enables independent evolution over time.

HTTP is also designed for use as an intermediation protocol, wherein proxies and gateways can translate non-HTTP information systems into a more generic interface.

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.

1.2. History and Evolution

HTTP has been the primary information transfer protocol for the World Wide Web since its introduction in 1990. It began as a trivial mechanism for low-latency requests, with a single method (GET) to request transfer of a presumed hypertext document identified by a given pathname. As the Web grew, HTTP was extended to enclose requests and responses within messages, transfer arbitrary data formats using MIME-like media types, and route requests through intermediaries. These protocols were eventually defined as HTTP/0.9 and HTTP/1.0 (see [HTTP/1.0]).

HTTP/1.1 was designed to refine the protocol's features while retaining compatibility with the existing text-based messaging syntax, improving its interoperability, scalability, and robustness across the Internet. This included length-based data delimiters for both fixed and dynamic (chunked) content, a consistent framework for content negotiation, opaque validators for conditional requests, cache controls for better cache consistency, range requests for partial updates, and default persistent connections. HTTP/1.1 was introduced in 1995 and published on the Standards Track in 1997 [RFC2068], revised in 1999 [RFC2616], and revised again in 2014 ([RFC7230] through [RFC7235]).

HTTP/2 ([HTTP/2]) introduced a multiplexed session layer on top of the existing TLS and TCP protocols for exchanging concurrent HTTP messages with efficient field compression and server push. HTTP/3 ([HTTP/3]) provides greater independence for concurrent messages by using QUIC as a secure multiplexed transport over UDP instead of TCP.

All three major versions of HTTP rely on the semantics defined by this document. They have not obsoleted each other because each one has specific benefits and limitations depending on the context of use. Implementations are expected to choose the most appropriate transport and messaging syntax for their particular context.

This revision of HTTP separates the definition of semantics (this document) and caching ([CACHING]) from the current HTTP/1.1 messaging syntax ([HTTP/1.1]) to allow each major protocol version to progress independently while referring to the same core semantics.

1.3. Core Semantics

HTTP provides a uniform interface for interacting with a resource (Section 3.1) — regardless of its type, nature, or implementation — by sending messages that manipulate or transfer representations (Section 3.2).

Each message is either a request or a response. A client constructs request messages that communicate its intentions and routes those messages toward an identified origin server. A server listens for requests, parses each message received, interprets the message semantics in relation to the identified target resource, and responds to that request with one or more response messages. The client examines received responses to see if its intentions were carried out, determining what to do next based on the status codes and content received.

HTTP semantics include the intentions defined by each request method (Section 9), extensions to those semantics that might be described in request header fields, status codes that describe the response (Section 15), and other control data and resource metadata that might be given in response fields.

Semantics also include representation metadata that describe how content is intended to be interpreted by a recipient, request header fields that might influence content selection, and the various selection algorithms that are collectively referred to as *content negotiation* (Section 12).

1.4. Specifications Obsoleted by This Document

Title	Reference	See
HTTP Over TLS	[RFC2818]	B.1
HTTP/1.1 Message Syntax and Routing [*]	[RFC7230]	B.2
HTTP/1.1 Semantics and Content	[RFC7231]	B.3
HTTP/1.1 Conditional Requests	[RFC7232]	B.4
HTTP/1.1 Range Requests	[RFC7233]	B.5
HTTP/1.1 Authentication	[RFC7235]	B.6
HTTP Status Code 308 (Permanent Redirect)	[RFC7538]	B.7
HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields	[RFC7615]	B.8
HTTP Client-Initiated Content-Encoding	[RFC7694]	B.9

Table 1

[*] This document only obsoletes the portions of RFC 7230 that are independent of the HTTP/1.1 messaging syntax and connection management; the remaining bits of RFC 7230 are obsoleted by "HTTP/1.1" [HTTP/1.1].

2. Conformance

2.1. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234], extended with the notation for case-sensitivity in strings defined in [RFC7405].

It also uses a list extension, defined in Section 5.6.1, 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).

Section 5.6 defines some generic syntactic components for field values.

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

2.2. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification targets conformance criteria according to the role of a participant in HTTP communication. Hence, 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 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 applies only to implementations that create the protocol element, rather than an implementation that forwards 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.

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).

Conformance to HTTP includes both conformance to the particular messaging syntax of the protocol version in use and conformance to the semantics of protocol elements sent. For example, a client that claims conformance to HTTP/1.1 but fails to recognize the features required of HTTP/1.1 recipients will fail to interoperate with servers that adjust their responses in accordance with those claims. Features that reflect user choices, such as content negotiation and user-selected extensions, can impact application behavior beyond the protocol stream; sending protocol elements that inaccurately reflect a user's choices will confuse the user and inhibit choice.

When an implementation fails semantic conformance, recipients of that implementation's messages will eventually develop workarounds to adjust their behavior accordingly. A recipient MAY employ such workarounds while remaining conformant to this protocol if the workarounds are limited to the implementations at fault. For example, servers often scan portions of the User-Agent field value, and user

agents often scan the Server field value, to adjust their own behavior with respect to known bugs or poorly chosen defaults.

2.3. Length Requirements

A recipient SHOULD parse a received protocol element defensively, with only marginal expectations that the element will conform to its ABNF grammar and fit within a reasonable buffer size.

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 three 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 target URI.

Many received protocol elements are only parsed to the extent necessary to identify and forward that element downstream. For example, an intermediary might parse a received field into its field name and field value components, but then forward the field without further parsing inside the field value.

2.4. 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.

Some requests can be automatically retried by a client in the event of an underlying connection failure, as described in Section 9.2.2.

2.5. Protocol Version

HTTP's version number consists of two decimal digits separated by a "." (period or decimal point). The first digit (major version) indicates the messaging syntax, whereas the second digit (minor version) indicates the highest minor version within that major version to which the sender is conformant (able to understand for future communication).

While HTTP's core semantics don't change between protocol versions, their expression "on the wire" can change, and so the HTTP version number changes when incompatible changes are made to the wire format. Additionally, HTTP allows incremental, backwards-compatible changes to be made to the protocol without changing its version through the use of defined extension points (Section 16).

The protocol version as a whole indicates the sender's conformance with the set of requirements laid out in that version's corresponding specification(s). For example, the version "HTTP/1.1" is defined by the combined specifications of this document, "HTTP Caching" [CACHING], and "HTTP/1.1" [HTTP/1.1].

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.

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).

When a major version of HTTP does not define any minor versions, the minor version "0" is implied. The "0" is used when referring to that protocol within elements that require a minor version identifier.

3. Terminology and Core Concepts

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 used to define HTTP.

3.1. 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. Most resources are identified by a Uniform Resource Identifier (URI), as described in Section 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 9) and a few request-modifying header fields. A resource cannot treat a request in a manner inconsistent with the semantics of the method of the request. For example, though the URI of a resource might imply semantics that are not safe, a client can expect the resource to avoid actions that are unsafe when processing a request with a safe method (see Section 9.2.1).

HTTP relies upon the Uniform Resource Identifier (URI) standard [URI] to indicate the target resource (Section 7.1) and relationships between resources.

3.2. Representations

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. A representation consists of a set of representation metadata and a potentially unbounded stream of representation data (Section 8).

HTTP allows "information hiding" behind its uniform interface by defining communication with respect to a transferable representation of the resource state, rather than transferring the resource itself. This allows the resource identified by a URI to be anything, including temporal functions like "the current weather in Laguna Beach", while potentially providing information that represents that resource at the time a message is generated [REST].

The uniform interface is similar to a window through which one can observe and act upon a thing only through the communication of messages to an independent actor on the other side. A shared abstraction is needed to represent ("take the place of") the current or desired state of that thing in our communications. When a representation is hypertext, it can provide both a representation of the resource state and processing instructions that help guide the recipient's future interactions.

A target resource might be provided with, or be capable of generating, multiple representations that are each intended to reflect the resource's current state. An algorithm, usually based on content negotiation (Section 12), would be used to select one of those representations as being most applicable to a given request. This *selected representation* provides the data and metadata for evaluating conditional requests (Section 13) and constructing the content for 200 (OK), 206 (Partial Content), and 304 (Not Modified) responses to GET (Section 9.3.1).

3.3. Connections, Clients, and Servers

HTTP is a client/server protocol that operates over a reliable transport- or session-layer *connection*.

An HTTP *client* is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP *server* is a program that accepts connections in order to service HTTP requests by sending HTTP responses.

The terms client and server refer only to the roles that these programs perform for a particular connection. The same program might act as a client on some connections and a server on others.

HTTP is defined as a stateless protocol, meaning that each request message's semantics can be understood in isolation, and that the relationship between connections and messages on them has no impact on the

interpretation of those messages. For example, a CONNECT request (Section 9.3.6) or a request with the Upgrade header field (Section 7.8) can occur at any time, not just in the first message on a connection. Many implementations depend on HTTP's stateless design in order to reuse proxied connections or dynamically load balance requests across multiple servers.

As a result, 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.

3.4. Messages

HTTP is a stateless request/response protocol for exchanging *messages* across a connection. The terms *sender* and *recipient* refer to any implementation that sends or receives a given message, respectively.

A client sends requests to a server in the form of a *request* message with a method (Section 9) and request target (Section 7.1). The request might also contain header fields (Section 6.3) for request modifiers, client information, and representation metadata, content (Section 6.4) intended for processing in accordance with the method, and trailer fields (Section 6.5) to communicate information collected while sending the content.

A server responds to a client's request by sending one or more *response* messages, each including a status code (Section 15). The response might also contain header fields for server information, resource metadata, and representation metadata, content to be interpreted in accordance with the status code, and trailer fields to communicate information collected while sending the content.

3.5. User Agents

The term *user agent* refers to any of the various client programs that initiate a request.

The most familiar form of user agent is the general-purpose Web browser, but that's only a small percentage of implementations. Other common user agents include spiders (web-traversing robots), command-line tools, billboard screens, household appliances, scales, light bulbs, firmware update scripts, mobile apps, and communication devices in a multitude of shapes and sizes.

Being a 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.

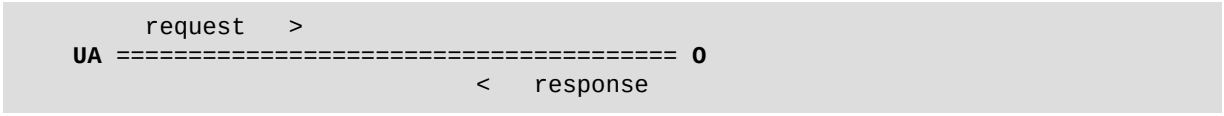
Many user agents cannot, or choose not to, make interactive suggestions to their user or provide adequate warning for security or privacy concerns. In the few cases where this specification requires reporting of errors to the user, it is acceptable for such reporting to only be observable in an error 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.

3.6. Origin Server

The term *origin server* refers to a program that can originate authoritative responses for a given target resource.

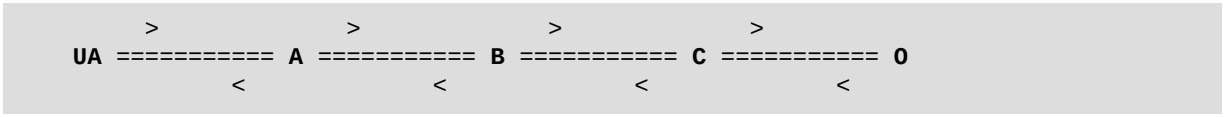
The most familiar form of origin server are large public websites. However, like user agents being equated with browsers, it is easy to be misled into thinking that all origin servers are alike. Common origin servers also include home automation units, configurable networking components, office machines, autonomous robots, news feeds, traffic cameras, real-time ad selectors, and video-on-demand platforms.

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).



3.7. Intermediaries

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



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

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

A *proxy* is a message-forwarding agent that is chosen 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 services, annotation services, or shared caching. Some proxies are designed to apply transformations to selected messages or content while they are being forwarded, as described in Section 7.7.

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 needs 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, [TLS13]) 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

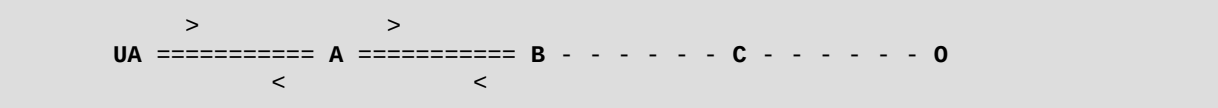
are indistinguishable (at a protocol level) from an on-path attacker, often introducing security flaws or interoperability problems due to mistakenly violating HTTP semantics.

For example, an *interception proxy* [RFC3040] (also commonly known as a *transparent proxy* [RFC1919]) differs from an HTTP proxy because it is not chosen 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.

3.8. 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 while acting as a tunnel.

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



A response is *cacheable* if a cache is allowed to store a copy of the response message for use in answering subsequent requests. Even when a response is cacheable, there might be additional constraints placed by the client or by the origin server on when that cached response can be used for a particular request. HTTP requirements for cache behavior and cacheable responses are defined in [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 bandwidth and reduce latency, content delivery networks that use gateway caching to optimize regional and global distribution of popular sites, 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.

3.9. Example Message Exchange

The following example illustrates a typical HTTP/1.1 message exchange for a GET request (Section 9.3.1) on the URI "http://www.example.com/hello.txt":

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.64.1
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 content includes a trailing CRLF.
```

4. Identifiers in HTTP

Uniform Resource Identifiers (URIs) [URI] are used throughout HTTP as the means for identifying resources (Section 3.1).

4.1. URI References

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", and "query" 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, 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 [URI], Section 4.1>
absolute-URI  = <absolute-URI, see [URI], Section 4.3>
relative-part = <relative-part, see [URI], Section 4.2>
authority     = <authority, see [URI], Section 3.2>
uri-host      = <host, see [URI], Section 3.2.2>
port          = <port, see [URI], Section 3.2.3>
path-abempty  = <path-abempty, see [URI], Section 3.3>
segment       = <segment, see [URI], Section 3.3>
query         = <query, see [URI], Section 3.4>

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 (partial-URI), or some combination of the above. Unless otherwise indicated, URI references are parsed relative to the target URI (Section 7.1).

It is RECOMMENDED that all senders and recipients support, at a minimum, URIs with lengths of 8000 octets in protocol elements. Note that this implies some structures and on-wire representations (for example, the request line in HTTP/1.1) will necessarily be larger in some cases.

4.2. HTTP-Related URI Schemes

IANA maintains the registry of URI Schemes [BCP35] at https://www.iana.org/assignments/uri-schemes/. Although requests might target any URI scheme, the following schemes are inherent to HTTP servers:

URI Scheme	Description	Section
http	Hypertext Transfer Protocol	4.2.1
https	Hypertext Transfer Protocol Secure	4.2.2

Table 2

Note that the presence of an "http" or "https" URI does not imply that there is always an HTTP server at the identified origin listening for connections. Anyone can mint a URI, whether or not a server exists and whether or not that server currently maps that identifier to a resource. The delegated nature of registered names and IP addresses creates a federated namespace whether or not an HTTP server is present.

4.2.1. http URI Scheme

The "http" URI scheme is hereby defined for minting identifiers within the hierarchical namespace governed by a potential HTTP origin server listening for TCP ([TCP]) connections on a given port.

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

The origin server for an "http" URI is identified by the authority component, which includes a host identifier ([URI], Section 3.2.2) and optional port number ([URI], Section 3.2.3). If the port subcomponent is empty or not given, TCP port 80 (the reserved port for WWW services) is the default. The origin determines who has the right to respond authoritatively to requests that target the identified resource, as defined in Section 4.3.2.

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.

The hierarchical path component and optional query component identify the target resource within that origin server's namespace.

4.2.2. https URI Scheme

The "https" URI scheme is hereby defined for minting identifiers within the hierarchical namespace governed by a potential origin server listening for TCP connections on a given port and capable of establishing a TLS ([TLS13]) connection that has been secured for HTTP communication. In this context, *secured* specifically means that the server has been authenticated as acting on behalf of the identified authority and all HTTP communication with that server has confidentiality and integrity protection that is acceptable to both client and server.

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

The origin server for an "https" URI is identified by the authority component, which includes a host identifier ([URI], Section 3.2.2) and optional port number ([URI], Section 3.2.3). If the port subcomponent is empty or not given, TCP port 443 (the reserved port for HTTP over TLS) is the default. The origin determines who has the right to respond authoritatively to requests that target the identified resource, as defined in Section 4.3.3.

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

The hierarchical path component and optional query component identify the target resource within that origin server's namespace.

A client MUST ensure that its HTTP requests for an "https" resource are secured, prior to being communicated, and that it only accepts secured responses to those requests. Note that the definition of what cryptographic mechanisms are acceptable to client and server are usually negotiated and can change over time.

Resources made available via the "https" scheme have no shared identity with the "http" scheme. They are distinct origins with separate namespaces. However, extensions to HTTP that are defined as applying to all origins with the same host, such as the Cookie protocol [COOKIE], allow information set by one service to impact communication with other services within a matching group of host domains. Such extensions ought to be designed with great care to prevent information obtained from a secured connection being inadvertently exchanged within an unsecured context.

4.2.3. http(s) Normalization and Comparison

URIs with an "http" or "https" scheme are normalized and compared according to the methods defined in Section 6 of [URI], using the defaults described above for each scheme.

HTTP does not require the use of a specific method for determining equivalence. For example, a cache key might be compared as a simple string, after syntax-based normalization, or after scheme-based normalization.

Scheme-based normalization (Section 6.2.3 of [URI]) of "http" and "https" URIs involves the following additional rules:

- 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 as the 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 [URI]).

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
```

Two HTTP URIs that are equivalent after normalization (using any method) can be assumed to identify the same resource, and any HTTP component MAY perform normalization. As a result, distinct resources SHOULD NOT be identified by HTTP URIs that are equivalent after normalization (using any method defined in Section 6.2 of [URI]).

4.2.4. Deprecation of userinfo in http(s) URIs

The URI generic syntax for authority also includes a userinfo subcomponent ([URI], Section 3.2.1) for including user authentication information in the URI. In that subcomponent, the use of the format "user:password" is deprecated.

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" or "https" URI reference is generated within a message as a target URI or field value.

Before making use of an "http" or "https" 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.

4.2.5. http(s) References with Fragment Identifiers

Fragment identifiers allow for indirect identification of a secondary resource, independent of the URI scheme, as defined in Section 3.5 of [URI]. Some protocol elements that refer to a URI allow inclusion of a fragment, while others do not. They are distinguished by use of the ABNF rule for elements where fragment is allowed; otherwise, a specific rule that excludes fragments is used.

> **Note:** The fragment identifier component is not part of the scheme definition for a URI scheme (see Section 4.3 of [URI]), thus does not appear in the ABNF definitions for the "http" and "https" URI schemes above.

4.3. Authoritative Access

Authoritative access refers to dereferencing a given identifier, for the sake of access to the identified resource, in a way that the client believes is authoritative (controlled by the resource owner). The process for determining whether access is granted is defined by the URI scheme and often uses data within the URI components, such as the authority component when the generic syntax is used. However, authoritative access is not limited to the identified mechanism.

Section 4.3.1 defines the concept of an origin as an aid to such uses, and the subsequent subsections explain how to establish that a peer has the authority to represent an origin.

See Section 17.1 for security considerations related to establishing authority.

4.3.1. URI Origin

The *origin* for a given URI is the triple of scheme, host, and port after normalizing the scheme and host to lowercase and normalizing the port to remove any leading zeros. If port is elided from the URI, the default port for that scheme is used. For example, the URI

```
https://Example.Com/happy.js
```

would have the origin

```
{ "https", "example.com", "443" }
```

which can also be described as the normalized URI prefix with port always present:

```
https://example.com:443
```

Each origin defines its own namespace and controls how identifiers within that namespace are mapped to resources. In turn, how the origin responds to valid requests, consistently over time, determines the semantics that users will associate with a URI, and the usefulness of those semantics is what ultimately transforms these mechanisms into a resource for users to reference and access in the future.

Two origins are distinct if they differ in scheme, host, or port. Even when it can be verified that the same entity controls two distinct origins, the two namespaces under those origins are distinct unless explicitly aliased by a server authoritative for that origin.

Origin is also used within HTML and related Web protocols, beyond the scope of this document, as described in [RFC6454].

4.3.2. http Origins

Although HTTP is independent of the transport protocol, the "http" scheme (Section 4.2.1) is specific to associating authority with whomever controls the origin server listening for TCP connections on the indicated port of whatever host is identified within the authority component. This is a very weak sense of authority because it depends on both client-specific name resolution mechanisms and communication that might not be secured from an on-path attacker. Nevertheless, it is a sufficient minimum for binding "http" identifiers to an origin server for consistent resolution within a trusted environment.

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 an appropriate origin server.

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 identifier to an IP address, establishing a TCP connection to that address on the indicated port, and sending over that connection an HTTP request message containing a request target that matches the client's target URI (Section 7.1).

If the server responds to such a request with a non-interim HTTP response message, as described in Section 15, then that response is considered an authoritative answer to the client's request.

Note, however, that the above is not the only means for obtaining an authoritative response, nor does it imply that an authoritative response is always necessary (see [CACHING]). For example, the Alt-Svc header field [ALTSVC] allows an origin server to identify other services that are also authoritative for that origin. Access to "http" identified resources might also be provided by protocols outside the scope of this document.

4.3.3. https Origins

The "https" scheme (Section 4.2.2) associates authority based on the ability of a server to use the private key corresponding to a certificate that the client considers to be trustworthy for the identified origin server. The client usually relies upon a chain of trust, conveyed from some prearranged or configured trust anchor, to deem a certificate trustworthy (Section 4.3.4).

In HTTP/1.1 and earlier, a client will only attribute authority to a server when they are communicating over a successfully established and secured connection specifically to that URI origin's host. The connection establishment and certificate verification are used as proof of authority.

In HTTP/2 and HTTP/3, a client will attribute authority to a server when they are communicating over a successfully established and secured connection if the URI origin's host matches any of the hosts present in the server's certificate and the client believes that it could open a connection to that host for that URI. In practice, a client will make a DNS query to check that the origin's host contains the same server IP address as the established connection. This restriction can be removed by the origin server sending an equivalent ORIGIN frame [RFC8336].

The request target's host and port value are passed within each HTTP request, identifying the origin and distinguishing it from other namespaces that might be controlled by the same server (Section 7.2). It is the origin's responsibility to ensure that any services provided with control over its certificate's private key are equally responsible for managing the corresponding "https" namespaces or at least prepared to reject requests that appear to have been misdirected (Section 7.4).

An origin server might be unwilling to process requests for certain target URIs even when they have the authority to do so. For example, when a host operates distinct services on different ports (e.g., 443 and 8000), checking the target URI at the origin server is necessary (even after the connection has been secured) because a network attacker might cause connections for one port to be received at some other port. Failing to check the target URI might allow such an attacker to replace a response to one target URI (e.g., "https://example.com/foo") with a seemingly authoritative response from the other port (e.g., "https://example.com:8000/foo").

Note that the "https" scheme does not rely on TCP and the connected port number for associating authority, since both are outside the secured communication and thus cannot be trusted as definitive. Hence, the HTTP communication might take place over any channel that has been secured, as defined in Section 4.2.2, including protocols that don't use TCP.

When an "https" URI is used within a context that calls for access to the indicated resource, a client MAY attempt access by resolving the host identifier to an IP address, establishing a TCP connection to that address on the indicated port, securing the connection end-to-end by successfully initiating TLS over TCP with confidentiality and integrity protection, and sending over that connection an HTTP request message containing a request target that matches the client's target URI (Section 7.1).

If the server responds to such a request with a non-interim HTTP response message, as described in Section 15, then that response is considered an authoritative answer to the client's request.

Note, however, that the above is not the only means for obtaining an authoritative response, nor does it imply that an authoritative response is always necessary (see [CACHING]).

4.3.4. https Certificate Verification

To establish a secured connection to dereference a URI, a client MUST verify that the service's identity is an acceptable match for the URI's origin server. Certificate verification is used to prevent server impersonation by an on-path attacker or by an attacker that controls name resolution. This process requires that a client be configured with a set of trust anchors.

In general, a client MUST verify the service identity using the verification process defined in Section 6 of [RFC6125]. The client MUST construct a reference identity from the service's host: if the host is a literal IP address (Section 4.3.5), the reference identity is an IP-ID, otherwise the host is a name and the reference identity is a DNS-ID.

A reference identity of type CN-ID MUST NOT be used by clients. As noted in Section 6.2.1 of [RFC6125], a reference identity of type CN-ID might be used by older clients.

A client might be specially configured to accept an alternative form of server identity verification. For example, a client might be connecting to a server whose address and hostname are dynamic, with an expectation that the service will present a specific certificate (or a certificate matching some externally defined reference identity) rather than one matching the target URI's origin.

In special cases, it might be appropriate for a client to simply ignore the server's identity, but it must be understood that this leaves a connection open to active attack.

If the certificate is not valid for the target URI's origin, a user agent MUST either obtain confirmation from the user before proceeding (see Section 3.5) or terminate the connection with a bad certificate error. Automated clients MUST log the error to an appropriate audit log (if available) and SHOULD terminate the connection (with a bad certificate error). Automated clients MAY provide a configuration setting that disables this check, but MUST provide a setting which enables it.

4.3.5. IP-ID Reference Identity

A server that is identified using an IP address literal in the "host" field of an "https" URI has a reference identity of type IP-ID. An IP version 4 address uses the "IPv4address" ABNF rule, and an IP version 6 address uses the "IP-literal" production with the "IPv6address" option; see Section 3.2.2 of [URI]. A reference identity of IP-ID contains the decoded bytes of the IP address.

An IP version 4 address is 4 octets, and an IP version 6 address is 16 octets. Use of IP-ID is not defined for any other IP version. The iPAddress choice in the certificate subjectAltName extension does not explicitly include the IP version and so relies on the length of the address to distinguish versions; see Section 4.2.1.6 of [RFC5280].

A reference identity of type IP-ID matches if the address is identical to an iPAddress value of the subjectAltName extension of the certificate.

5. Fields

HTTP uses *fields* to provide data in the form of extensible name/value pairs with a registered key namespace. Fields are sent and received within the header and trailer sections of messages (Section 6).

5.1. Field Names

A field name labels the corresponding field value as having the semantics defined by that name. For example, the Date header field is defined in Section 6.6.1 as containing the origination timestamp for the message in which it appears.

```
field-name     = token
```

Field names are case-insensitive and ought to be registered within the "Hypertext Transfer Protocol (HTTP) Field Name Registry"; see Section 16.3.1.

The interpretation of a 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, fields are defined for all versions of HTTP. In particular, the Host and Connection fields ought to be recognized by all HTTP implementations whether or not they advertise conformance with HTTP/1.1.

New 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; see Section 16.3.

A proxy MUST forward unrecognized header fields unless the field name is listed in the Connection header field (Section 7.6.1) or the proxy is specifically configured to block, or otherwise transform, such fields. Other recipients SHOULD ignore unrecognized header and trailer fields. Adhering to these requirements allows HTTP's functionality to be extended without updating or removing deployed intermediaries.

5.2. Field Lines and Combined Field Value

Field sections are composed of any number of *field lines*, each with a *field name* (see Section 5.1) identifying the field, and a *field line value* that conveys data for that instance of the field.

When a field name is only present once in a section, the combined *field value* for that field consists of the corresponding field line value. When a field name is repeated within a section, its combined field value consists of the list of corresponding field line values within that section, concatenated in order, with each field line value separated by a comma.

For example, this section:

```
Example-Field: Foo, Bar
Example-Field: Baz
```

contains two field lines, both with the field name "Example-Field". The first field line has a field line value of "Foo, Bar", while the second field line value is "Baz". The field value for "Example-Field" is the list "Foo, Bar, Baz".

5.3. Field Order

A recipient MAY combine multiple field lines within a field section that have the same field name into one field line, without changing the semantics of the message, by appending each subsequent field line value to the initial field line value in order, separated by a comma (",") and optional whitespace (OWS, defined in Section 5.6.3). For consistency, use comma SP.

The order in which field lines with the same name are received is therefore significant to the interpretation of the field value; a proxy MUST NOT change the order of these field line values when forwarding a message.

This means that, aside from the well-known exception noted below, a sender MUST NOT generate multiple field lines with the same name in a message (whether in the headers or trailers) or append a field line when a field line of the same name already exists in the message, unless that field's definition allows multiple field line values to be recombined as a comma-separated list (i.e., at least one alternative of the field's definition allows a comma-separated list, such as an ABNF rule of #(values) defined in Section 5.6.1).

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

The order in which field lines with differing field names are received in a section is not significant. However, it is good practice to send header fields that contain additional 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 it receives the entire request header section, since later header field lines might include conditionals, authentication credentials, or deliberately misleading duplicate header fields that could impact request processing.

5.4. Field Limits

HTTP does not place a predefined limit on the length of each field line, field value, or on the length of a header or trailer section as a whole, as described in Section 2. Various ad hoc limitations on individual lengths are found in practice, often depending on the specific field's semantics.

A server that receives a request header field line, field value, 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 [HTTP/1.1]).

A client MAY discard or truncate received field lines 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.

5.5. Field Values

HTTP field values consist of a sequence of characters in a format defined by the field's grammar. Each field's grammar is usually defined using ABNF ([RFC5234]).

```
field-value    = *field-content
field-content  = field-vchar
                 [ 1*( SP / HTAB / field-vchar ) field-vchar ]
field-vchar    = VCHAR / obs-text
obs-text       = %x80-FF
```

A field value does not include leading or trailing whitespace. When a specific version of HTTP allows such whitespace to appear in a message, a field parsing implementation MUST exclude such whitespace prior to evaluating the field value.

Field values are usually constrained to the range of US-ASCII characters [USASCII]. Fields needing a greater range of characters can use an encoding, such as the one defined in [RFC8187]. Historically, HTTP allowed field content with text in the ISO-8859-1 charset [ISO-8859-1], supporting other charsets only through use of [RFC2047] encoding. Specifications for newly defined fields SHOULD limit their values to visible US-ASCII octets (VCHAR), SP, and HTAB. A recipient SHOULD treat other allowed octets in field content (i.e., obs-text) as opaque data.

Field values containing CR, LF, or NUL characters are invalid and dangerous, due to the varying ways that implementations might parse and interpret those characters; a recipient of CR, LF, or NUL within a field value MUST either reject the message or replace each of those characters with SP before further processing

or forwarding of that message. Field values containing other CTL characters are also invalid; however, recipients MAY retain such characters for the sake of robustness when they appear within a safe context (e.g., an application-specific quoted string that will not be processed by any downstream HTTP parser).

Fields that only anticipate a single member as the field value are referred to as *singleton fields*.

Fields that allow multiple members as the field value are referred to as *list-based fields*. The list operator extension of Section 5.6.1 is used as a common notation for defining field values that can contain multiple members.

Because commas (",") are used as the delimiter between members, they need to be treated with care if they are allowed as data within a member. This is true for both list-based and singleton fields, since a singleton field might be erroneously sent with multiple members and detecting such errors improves interoperability. Fields that expect to contain a comma within a member, such as within an HTTP-date or URI-reference element, ought to be defined with delimiters around that element to distinguish any comma within that data from potential list separators.

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

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

Note that double-quote delimiters are almost always used with the quoted-string production (Section 5.6.4); using a different syntax inside double-quotes will likely cause unnecessary confusion.

Many fields (such as Content-Type, defined in Section 8.3) use a common syntax for parameters that allows both unquoted (token) and quoted (quoted-string) syntax for a parameter value (Section 5.6.6). Use of common syntax allows recipients to reuse existing parser components. When allowing both forms, the meaning of a parameter value ought to be the same whether it was received as a token or a quoted string.

> **Note:** For defining field value syntax, this specification uses an ABNF rule named after the field name to define the allowed grammar for that field's value (after said value has been extracted from the underlying messaging syntax and multiple instances combined into a list).

5.6. Common Rules for Defining Field Values

5.6.1. Lists (#rule ABNF Extension)

A #rule extension to the ABNF rules of [RFC5234] is used to improve readability in the definitions of some list-based 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, defined in Section 5.6.3).

5.6.1.1. Sender Requirements

In any production that uses the list construct, a sender MUST NOT generate empty list elements. In other words, a sender has to 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 )
```

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

5.6.1.2. Recipient Requirements

Empty elements do not contribute to the count of elements present. 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 ] )
```

Note that because of the potential presence of empty list elements, the RFC 5234 ABNF cannot enforce the cardinality of list elements, and consequently all cases are mapped as if there was no cardinality specified.

For example, given these ABNF productions:

```
example-list      = 1#example-list-elmt
example-list-elmt = token ; see Section 5.6.2
```

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:

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

5.6.2. Tokens

Tokens are short textual identifiers that do not include whitespace or delimiters.

```
token          = 1*tchar

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

Many HTTP field values are defined using common syntax components, 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 "(),/:;<=>?@[\]{}").

5.6.3. 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 overwrite 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.

OWS and RWS have the same semantics as a single SP. Any content known to be defined as OWS or RWS MAY be replaced with a single SP before interpreting it or forwarding the message downstream.

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.

BWS has no semantics. Any content known to be defined as BWS MAY be removed before interpreting it or forwarding the message downstream.

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

5.6.4. Quoted Strings

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
```

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.

5.6.5. Comments

Comments can be included in some HTTP 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
```

5.6.6. Parameters

Parameters are instances of name/value pairs; they are often used in field values as a common syntax for appending auxiliary information to an item. Each parameter is usually delimited by an immediately preceding semicolon.

```
parameters      = *( OWS ";" OWS [ parameter ] )
parameter       = parameter-name "=" parameter-value
parameter-name  = token
parameter-value = ( token / quoted-string )
```

Parameter names are case-insensitive. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name. Examples of parameters and some equivalent forms can be seen in media types (Section 8.3.1) and the Accept header field (Section 12.5.1).

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.

Note: Parameters do not allow whitespace (not even "bad" whitespace) around the "=" character.

5.6.7. 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 field MUST accept all three HTTP-date formats. When a sender generates a 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 *clock* is an implementation capable of providing a reasonable approximation of the current instant in UTC. A clock implementation ought to use NTP ([RFC5905]), or some similar protocol, to synchronize with 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     = %s"Mon" / %s"Tue" / %s"Wed"
             / %s"Thu" / %s"Fri" / %s"Sat" / %s"Sun"

date1        = day SP month SP year
             ; e.g., 02 Jun 1982

day          = 2DIGIT
month        = %s"Jan" / %s"Feb" / %s"Mar" / %s"Apr"
             / %s"May" / %s"Jun" / %s"Jul" / %s"Aug"
             / %s"Sep" / %s"Oct" / %s"Nov" / %s"Dec"
year         = 4DIGIT

GMT          = %s"GMT"

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   = %s"Monday" / %s"Tuesday" / %s"Wednesday"
             / %s"Thursday" / %s"Friday" / %s"Saturday"
             / %s"Sunday"

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. Note that Section 4.2 of [CACHING] relaxes this for cache recipients.

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 timestamp formats apply only to their usage within the protocol stream. Implementations are not required to use these formats for user presentation, request logging, etc.

6. Message Abstraction

Each major version of HTTP defines its own syntax for communicating messages. This section defines an abstract data type for HTTP messages based on a generalization of those message characteristics, common structure, and capacity for conveying semantics. This abstraction is used to define requirements on senders and recipients that are independent of the HTTP version, such that a message in one version can be relayed through other versions without changing its meaning.

A *message* consists of the following:

- control data to describe and route the message,
- a headers lookup table of name/value pairs for extending that control data and conveying additional information about the sender, message, content, or context,
- a potentially unbounded stream of content, and
- a trailers lookup table of name/value pairs for communicating information obtained while sending the content.

Framing and control data is sent first, followed by a header section containing fields for the headers table. When a message includes content, the content is sent after the header section, potentially followed by a trailer section that might contain fields for the trailers table.

Messages are expected to be processed as a stream, wherein the purpose of that stream and its continued processing is revealed while being read. Hence, control data describes what the recipient needs to know immediately, header fields describe what needs to be known before receiving content, the content (when present) presumably contains what the recipient wants or needs to fulfill the message semantics, and trailer fields provide optional metadata that was unknown prior to sending the content.

Messages are intended to be *self-descriptive*: everything a recipient needs to know about the message can be determined by looking at the message itself, after decoding or reconstituting parts that have been compressed or elided in transit, without requiring an understanding of the sender's current application state (established via prior messages). However, a client MUST retain knowledge of the request when parsing, interpreting, or caching a corresponding response. For example, responses to the HEAD method look just like the beginning of a response to GET but cannot be parsed in the same manner.

Note that this message abstraction is a generalization across many versions of HTTP, including features that might not be found in some versions. For example, trailers were introduced within the HTTP/1.1 chunked transfer coding as a trailer section after the content. An equivalent feature is present in HTTP/2 and HTTP/3 within the header block that terminates each stream.

6.1. Framing and Completeness

Message framing indicates how each message begins and ends, such that each message can be distinguished from other messages or noise on the same connection. Each major version of HTTP defines its own framing mechanism.

HTTP/0.9 and early deployments of HTTP/1.0 used closure of the underlying connection to end a response. For backwards compatibility, this implicit framing is also allowed in HTTP/1.1. However, implicit framing can fail to distinguish an incomplete response if the connection closes early. For that reason, almost all modern implementations use explicit framing in the form of length-delimited sequences of message data.

A message is considered *complete* when all of the octets indicated by its framing are available. Note that, when no explicit framing is used, a response message that is ended by the underlying connection's close is considered complete even though it might be indistinguishable from an incomplete response, unless a transport-level error indicates that it is not complete.

6.2. Control Data

Messages start with control data that describe its primary purpose. Request message control data includes a request method (Section 9), request target (Section 7.1), and protocol version (Section 2.5). Response message control data includes a status code (Section 15), optional reason phrase, and protocol version.

In HTTP/1.1 ([HTTP/1.1]) and earlier, control data is sent as the first line of a message. In HTTP/2 ([HTTP/2]) and HTTP/3 ([HTTP/3]), control data is sent as pseudo-header fields with a reserved name prefix (e.g., ":authority").

Every HTTP message has a protocol version. Depending on the version in use, it might be identified within the message explicitly or inferred by the connection over which the message is received. Recipients use that version information to determine limitations or potential for later communication with that sender.

When a message is forwarded by an intermediary, the protocol version is updated to reflect the version used by that intermediary. The Via header field (Section 7.6.3) is used to communicate upstream protocol information within a forwarded message.

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

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

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

A recipient that receives a message with a major version number that it implements and a minor version number higher than what it implements 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.

6.3. Header Fields

Fields (Section 5) that are sent or received before the content are referred to as "header fields" (or just "headers", colloquially).

The *header section* of a message consists of a sequence of header field lines. Each header field might modify or extend message semantics, describe the sender, define the content, or provide additional context.

> **Note:** We refer to named fields specifically as a "header field" when they are only allowed to be sent in the header section.

6.4. Content

HTTP messages often transfer a complete or partial representation as the message *content*: a stream of octets sent after the header section, as delineated by the message framing.

This abstract definition of content reflects the data after it has been extracted from the message framing. For example, an HTTP/1.1 message body (Section 6 of [HTTP/1.1]) might consist of a stream of data encoded with the chunked transfer coding — a sequence of data chunks, one zero-length chunk, and a trailer section — whereas the content of that same message includes only the data stream after the transfer coding has been decoded; it does not include the chunk lengths, chunked framing syntax, nor the trailer fields (Section 6.5).

> **Note:** Some field names have a "Content-" prefix. This is an informal convention; while some of these fields refer to the content of the message, as defined above, others are scoped to the selected representation (Section 3.2). See the individual field's definition to disambiguate.

6.4.1. Content Semantics

The purpose of content in a request is defined by the method semantics (Section 9).

For example, a representation in the content of a PUT request (Section 9.3.4) represents the desired state of the target resource after the request is successfully applied, whereas a representation in the content of a POST request (Section 9.3.3) represents information to be processed by the target resource.

In a response, the content's purpose is defined by the request method, response status code (Section 15), and response fields describing that content. For example, the content of a 200 (OK) response to GET (Section 9.3.1) represents the current state of the target resource, as observed at the time of the message origination date (Section 6.6.1), whereas the content 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.

The content of a 206 (Partial Content) response to GET contains either a single part of the selected representation or a multipart message body containing multiple parts of that representation, as described in Section 15.3.7.

Response messages with an error status code usually contain content that represents the error condition, such that the content describes the error state and what steps are suggested for resolving it.

Responses to the HEAD request method (Section 9.3.2) never include content; the associated response header fields indicate only what their values would have been if the request method had been GET (Section 9.3.1).

2xx (Successful) responses to a CONNECT request method (Section 9.3.6) switch the connection to tunnel mode instead of having content.

All 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include content.

All other responses do include content, although that content might be of zero length.

6.4.2. Identifying Content

When a complete or partial representation is transferred as message content, it is often desirable for the sender to supply, or the recipient to determine, an identifier for a resource corresponding to that specific representation. For example, a client making a GET request on a resource for "the current weather report" might want an identifier specific to the content returned (e.g., "weather report for Laguna Beach at 20210720T1711"). This can be useful for sharing or bookmarking content from resources that are expected to have changing representations over time.

For a request message:

- If the request has a Content-Location header field, then the sender asserts that the content 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 content is unidentified by HTTP, but a more specific identifier might be supplied within the content itself.

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

1. If the request method is HEAD or the response status code is 204 (No Content) or 304 (Not Modified), there is no content in the response.
2. If the request method is GET and the response status code is 200 (OK), the content is a representation of the target resource (Section 7.1).
3. If the request method is GET and the response status code is 203 (Non-Authoritative Information), the content is a potentially modified or enhanced representation of the target resource as provided by an intermediary.
4. If the request method is GET and the response status code is 206 (Partial Content), the content is one or more parts of a representation of the target resource.

5. If the response has a Content-Location header field and its field value is a reference to the same URI as the target URI, the content is a representation of the target resource.
6. If the response has a Content-Location header field and its field value is a reference to a URI different from the target URI, then the sender asserts that the content 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).
7. Otherwise, the content is unidentified by HTTP, but a more specific identifier might be supplied within the content itself.

6.5. Trailer Fields

Fields (Section 5) that are located within a *trailer section* are referred to as "trailer fields" (or just "trailers", colloquially). Trailer fields can be useful for supplying message integrity checks, digital signatures, delivery metrics, or post-processing status information.

Trailer fields ought to be processed and stored separately from the fields in the header section to avoid contradicting message semantics known at the time the header section was complete. The presence or absence of certain header fields might impact choices made for the routing or processing of the message as a whole before the trailers are received; those choices cannot be unmade by the later discovery of trailer fields.

6.5.1. Limitations on Use of Trailers

A trailer section is only possible when supported by the version of HTTP in use and enabled by an explicit framing mechanism. For example, the chunked transfer coding in HTTP/1.1 allows a trailer section to be sent after the content (Section 7.1.2 of [HTTP/1.1]).

Many fields cannot be processed outside the header section because their evaluation is necessary prior to receiving the content, such as those that describe message framing, routing, authentication, request modifiers, response controls, or content format. A sender MUST NOT generate a trailer field unless the sender knows the corresponding header field name's definition permits the field to be sent in trailers.

Trailer fields can be difficult to process by intermediaries that forward messages from one protocol version to another. If the entire message can be buffered in transit, some intermediaries could merge trailer fields into the header section (as appropriate) before it is forwarded. However, in most cases, the trailers are simply discarded. A recipient MUST NOT merge a trailer field into a header section unless the recipient understands the corresponding header field definition and that definition explicitly permits and defines how trailer field values can be safely merged.

The presence of the keyword "trailers" in the TE header field (Section 10.1.4) of a request indicates that the client is willing to accept trailer fields, on behalf of itself and any downstream clients. For requests from an intermediary, this implies that all downstream clients are willing to accept trailer fields in the forwarded response. Note that the presence of "trailers" does not mean that the client(s) will process any particular trailer field in the response; only that the trailer section(s) will not be dropped by any of the clients.

Because of the potential for trailer fields to be discarded in transit, a server SHOULD NOT generate trailer fields that it believes are necessary for the user agent to receive.

6.5.2. Processing Trailer Fields

The "Trailer" header field (Section 6.6.2) can be sent to indicate fields likely to be sent in the trailer section, which allows recipients to prepare for their receipt before processing the content. For example, this could be useful if a field name indicates that a dynamic checksum should be calculated as the content is received and then immediately checked upon receipt of the trailer field value.

Like header fields, trailer fields with the same name are processed in the order received; multiple trailer field lines with the same name have the equivalent semantics as appending the multiple values as a list of members. Trailer fields that might be generated more than once during a message MUST be defined as a list-based field even if each member value is only processed once per field line received.

At the end of a message, a recipient MAY treat the set of received trailer fields as a data structure of name/value pairs, similar to (but separate from) the header fields. Additional processing expectations, if any, can be defined within the field specification for a field intended for use in trailers.

6.6. Message Metadata

Fields that describe the message itself, such as when and how the message has been generated, can appear in both requests and responses.

6.6.1. 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 5.6.7.

```
Date = HTTP-date
```

An example is

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

A sender that generates a Date header field 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 generating the message content. In practice, a sender can generate the date value at any time during message origination.

An origin server with a clock (as defined in Section 5.6.7) MUST generate a Date header field in all 2xx (Successful), 3xx (Redirection), and 4xx (Client Error) responses, and MAY generate a Date header field in 1xx (Informational) and 5xx (Server Error) responses.

An origin server without a clock MUST NOT generate a Date header field.

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 recipient with a clock that receives a response with an invalid Date header field value MAY replace that value with the time that response was received.

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.

6.6.2. Trailer

The "Trailer" header field provides a list of field names that the sender anticipates sending as trailer fields within that message. This allows a recipient to prepare for receipt of the indicated metadata before it starts processing the content.

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

For example, a sender might indicate that a signature will be computed as the content is being streamed and provide the final signature as a trailer field. This allows a recipient to perform the same check on the fly as it receives the content.

A sender that intends to generate one or more trailer fields in a message SHOULD generate a Trailer header field in the header section of that message to indicate which fields might be present in the trailers.

If an intermediary discards the trailer section in transit, the Trailer field could provide a hint of what metadata was lost, though there is no guarantee that a sender of Trailer will always follow through by sending the named fields.

7. Routing HTTP Messages

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.

7.1. Determining the Target Resource

Although HTTP is used in a wide variety of applications, most clients rely on the same resource identification mechanism and configuration techniques as general-purpose Web browsers. Even when communication options are hard-coded in a client's configuration, we can think of their combined effect as a URI reference (Section 4.1).

A URI reference is resolved 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 ([URI], Section 3.5).

To perform an action on a *target resource*, the client sends a request message containing enough components of its parsed target URI to enable recipients to identify that same resource. For historical reasons, the parsed target URI components, collectively referred to as the *request target*, are sent within the message control data and the Host header field (Section 7.2).

There are two unusual cases for which the request target components are in a method-specific form:

- For CONNECT (Section 9.3.6), the request target is the host name and port number of the tunnel destination, separated by a colon.
- For OPTIONS (Section 9.3.7), the request target can be a single asterisk ("*").

See the respective method definitions for details. These forms MUST NOT be used with other methods.

Upon receipt of a client's request, a server reconstructs the target URI from the received components in accordance with their local configuration and incoming connection context. This reconstruction is specific to each major protocol version. For example, Section 3.3 of [HTTP/1.1] defines how a server determines the target URI of an HTTP/1.1 request.

> **Note:** Previous specifications defined the recomposed target URI as a distinct concept, the *effective request URI*.

7.2. Host and :authority

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.

In HTTP/2 [HTTP/2] and HTTP/3 [HTTP/3], the Host header field is, in some cases, supplanted by the ":authority" pseudo-header field of a request's control data.

```
Host = uri-host [ ":" port ] ; Section 4
```

The target URI's authority information is critical for handling a request. A user agent MUST generate a Host header field in a request unless it sends that information as an ":authority" pseudo-header field. A user agent that sends Host SHOULD send it as the first field in the header section of a request.

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
```

Since the host and port information 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 and port information 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.

7.3. Routing Inbound Requests

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

7.3.1. To a Cache

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

7.3.2. To a Proxy

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 an "http" or "https" proxy is applicable, the client connects inbound by establishing (or reusing) a connection to that proxy and then sending it an HTTP request message containing a request target that matches the client's target URI.

7.3.3. To the Origin

If no proxy is applicable, a typical client will invoke a handler routine (specific to the target URI's scheme) to obtain access to the identified resource. How that is accomplished is dependent on the target URI scheme and defined by its associated specification.

Section 4.3.2 defines how to obtain access to an "http" resource by establishing (or reusing) an inbound connection to the identified origin server and then sending it an HTTP request message containing a request target that matches the client's target URI.

Section 4.3.3 defines how to obtain access to an "https" resource by establishing (or reusing) an inbound secured connection to an origin server that is authoritative for the identified origin and then sending it an HTTP request message containing a request target that matches the client's target URI.

7.4. Rejecting Misdirected Requests

Once a request is received by a server and parsed sufficiently to determine its target URI, the server decides whether to process the request itself, forward the request to another server, redirect the client to a different resource, respond with an error, or drop the connection. This decision can be influenced by anything about the request or connection context, but is specifically directed at whether the server has been configured to process requests for that target URI and whether the connection context is appropriate for that request.

For example, a request might have been misdirected, deliberately or accidentally, such that the information within a received Host header field differs from the connection's host or port. If the connection is from a trusted gateway, such 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 17 for security considerations regarding message routing.

Unless the connection is from a trusted gateway, an origin server MUST reject a request if any scheme-specific requirements for the target URI are not met. In particular, a request for an "https" resource MUST be rejected unless it has been received over a connection that has been secured via a certificate valid for that target URI's origin, as defined by Section 4.2.2.

The 421 (Misdirected Request) status code in a response indicates that the origin server has rejected the request because it appears to have been misdirected (Section 15.5.20).

7.5. Response Correlation

A connection might be used for multiple request/response exchanges. The mechanism used to correlate between request and response messages is version dependent; some versions of HTTP use implicit ordering of messages, while others use an explicit identifier.

All responses, regardless of the status code (including interim responses) can be sent at any time after a request is received, even if the request is not yet complete. A response can complete before its corresponding request is complete (Section 6.1). Likewise, clients are not expected to wait any specific amount of time for a response. Clients (including intermediaries) might abandon a request if the response is not received within a reasonable period of time.

A client that receives a response while it is still sending the associated request SHOULD continue sending that request unless it receives an explicit indication to the contrary (see, e.g., Section 9.5 of [HTTP/1.1] and Section 6.4 of [HTTP/2]).

7.6. Message Forwarding

As described in Section 3.7, 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.

Intermediaries are expected to forward messages even when protocol elements are not recognized (e.g., new methods, status codes, or field names) since that preserves extensibility for downstream recipients.

An intermediary not acting as a tunnel MUST implement the Connection header field, as specified in Section 7.6.1, and exclude fields from being forwarded that are only intended for the incoming connection.

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

An HTTP message can be parsed as a stream for incremental processing or forwarding downstream. However, senders and 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 content transformations.

7.6.1. Connection

The "Connection" header field allows the sender to list desired control options for the current connection.

```
Connection        = #connection-option
connection-option = token
```

Connection options are case-insensitive.

When a field aside from Connection is used to supply control information for or about the current connection, the sender MUST list the corresponding field name within the Connection header field. Note that some versions of HTTP prohibit the use of fields for such information, and therefore do not allow the Connection field.

Intermediaries MUST parse a received Connection header field before a message is forwarded and, for each connection-option in this field, remove any header or trailer field(s) from the message with the same name as the connection-option, and then remove the Connection header field itself (or replace it with the intermediary's own control options for the forwarded message).

Hence, the Connection header field provides a declarative way of distinguishing fields that are only intended for the immediate recipient ("hop-by-hop") from those fields that are intended for all recipients on the chain ("end-to-end"), enabling the message to be self-descriptive and allowing future connection-specific extensions to be deployed without fear that they will be blindly forwarded by older intermediaries.

Furthermore, intermediaries SHOULD remove or replace fields that are known to require removal before forwarding, whether or not they appear as a connection-option, after applying those fields' semantics. This includes but is not limited to:

- Proxy-Connection (Appendix C.2.2 of [HTTP/1.1])
- Keep-Alive (Section 19.7.1 of [RFC2068])
- TE (Section 10.1.4)
- Transfer-Encoding (Section 6.1 of [HTTP/1.1])
- Upgrade (Section 7.8)

A sender MUST NOT send a connection option corresponding to a field that is intended for all recipients of the content. For example, Cache-Control is never appropriate as a connection option (Section 5.2 of [CACHING]).

Connection options do not always correspond to a field present in the message, since a connection-specific field might not be needed if there are no parameters associated with a connection option. In contrast, a connection-specific field received without a corresponding connection option usually indicates that the field has been improperly forwarded by an intermediary and ought to be ignored by the recipient.

When defining a new connection option that does not correspond to a field, specification authors ought to reserve the corresponding field name anyway in order to avoid later collisions. Such reserved field names are registered in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" (Section 16.3.1).

7.6.2. Max-Forwards

The "Max-Forwards" header field provides a mechanism with the TRACE (Section 9.3.8) and OPTIONS (Section 9.3.7) 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.

7.6.3. 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 = #( received-protocol RWS received-by [ RWS comment ] )

received-protocol = [ protocol-name "/" ] protocol-version
                  ; see Section 7.8
received-by       = pseudonym [ ":" port ]
pseudonym         = token
```

Each member of the Via field value represents a 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 2.5. For brevity, the protocol-name is omitted when the received protocol is HTTP.

The received-by portion 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 port, if any, for the received-protocol.

A sender MAY generate comments to identify the software of each recipient, analogous to the User-Agent and Server header fields. However, comments in Via 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 list members into a single member 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 list members unless they are all under the same organizational control and the hosts have already been replaced by pseudonyms. A sender MUST NOT combine members that have different received-protocol values.

7.7. Message Transformations

Some intermediaries include features for transforming messages and their content. 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 content 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 content 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) chose the proxy.

If a proxy receives a target URI 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 target URI contains a fully qualified domain name.

A proxy MUST NOT modify the "absolute-path" and "query" parts of the received target URI when forwarding it to the next inbound server except as required by that forwarding protocol. For example, a proxy forwarding a request to an origin server via HTTP/1.1 will replace an empty path with "/" (Section 3.2.1 of [HTTP/1.1]) or "*" (Section 3.2.4 of [HTTP/1.1]), depending on the request method.

A proxy MUST NOT transform the content (Section 6.4) of a response message that contains a no-transform cache directive (Section 5.2.2.6 of [CACHING]). Note that this does not apply to message transformations that do not change the content, such as the addition or removal of transfer codings (Section 7 of [HTTP/1.1]).

A proxy MAY transform the content of a message that does not contain a no-transform cache directive. A proxy that transforms the content of a 200 (OK) response can inform downstream recipients that a transformation has been applied by changing the response status code to 203 (Non-Authoritative Information) (Section 15.3.4).

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 content) unless the field's definition specifically allows such modification or the modification is deemed necessary for privacy or security.

7.8. Upgrade

The "Upgrade" header field is intended to provide a simple mechanism for transitioning from HTTP/1.1 to some other protocol on the same connection.

A client MAY send a list of protocol names in the Upgrade header field of a request to invite the server to switch to one or more of the named protocols, in order of descending preference, before sending the final response. A server MAY ignore a received Upgrade header field if it wishes to continue using the current protocol on that connection. Upgrade cannot be used to insist on a protocol change.

```
Upgrade          = #protocol

protocol         = protocol-name ["/" protocol-version]
protocol-name    = token
protocol-version = token
```

Although protocol names are registered with a preferred case, recipients SHOULD use case-insensitive comparison when matching each protocol-name to supported protocols.

A server that sends a 101 (Switching Protocols) response MUST send an Upgrade header field to indicate the new protocol(s) to which the connection is being switched; if multiple protocol layers are being switched, the sender MUST list the protocols in layer-ascending order. A server MUST NOT switch to a protocol that was

not indicated by the client in the corresponding request's Upgrade header field. A server MAY choose to ignore the order of preference indicated by the client and select the new protocol(s) based on other factors, such as the nature of the request or the current load on the server.

A server that sends a 426 (Upgrade Required) response MUST send an Upgrade header field to indicate the acceptable protocols, in order of descending preference.

A server MAY send an Upgrade header field in any other response to advertise that it implements support for upgrading to the listed protocols, in order of descending preference, when appropriate for a future request.

The following is a hypothetical example sent by a client:

```
GET /hello HTTP/1.1
Host: www.example.com
Connection: upgrade
Upgrade: websocket, IRC/6.9, RTA/x11

```

The capabilities and nature of the application-level communication after the protocol change is entirely dependent upon the new protocol(s) chosen. However, immediately after sending the 101 (Switching Protocols) response, the server is expected to continue responding to the original request as if it had received its equivalent within the new protocol (i.e., the server still has an outstanding request to satisfy after the protocol has been changed, and is expected to do so without requiring the request to be repeated).

For example, if the Upgrade header field is received in a GET request and the server decides to switch protocols, it first responds with a 101 (Switching Protocols) message in HTTP/1.1 and then immediately follows that with the new protocol's equivalent of a response to a GET on the target resource. This allows a connection to be upgraded to protocols with the same semantics as HTTP without the latency cost of an additional round trip. A server MUST NOT switch protocols unless the received message semantics can be honored by the new protocol; an OPTIONS request can be honored by any protocol.

The following is an example response to the above hypothetical request:

```
HTTP/1.1 101 Switching Protocols
Connection: upgrade
Upgrade: websocket

[... data stream switches to websocket with an appropriate response
(as defined by new protocol) to the "GET /hello" request ...]
```

A sender of Upgrade MUST also send an "Upgrade" connection option in the Connection header field (Section 7.6.1) to inform intermediaries not to forward this field. A server that receives an Upgrade header field in an HTTP/1.0 request MUST ignore that Upgrade field.

A client cannot begin using an upgraded protocol on the connection until it has completely sent the request message (i.e., the client can't change the protocol it is sending in the middle of a message). If a server receives both an Upgrade and an Expect header field with the "100-continue" expectation (Section 10.1.1), the server MUST send a 100 (Continue) response before sending a 101 (Switching Protocols) response.

The Upgrade header field only applies to switching protocols on top of the existing connection; it cannot be used to switch the underlying connection (transport) protocol, nor to switch the existing communication to a different connection. For those purposes, it is more appropriate to use a 3xx (Redirection) response (Section 15.4).

This specification only defines the protocol name "HTTP" for use by the family of Hypertext Transfer Protocols, as defined by the HTTP version rules of Section 2.5 and future updates to this specification. Additional protocol names ought to be registered using the registration procedure defined in Section 16.7.

8. Representation Data and Metadata

8.1. Representation Data

The representation data associated with an HTTP message is either provided as the content of the message or referred to by the message semantics and the target 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( data ) )
```

8.2. Representation Metadata

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

8.3. Content-Type

The "Content-Type" header field indicates the media type of the associated representation: either the representation enclosed in the message content 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 8.3.1. An example of the field is

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

A sender that generates a message containing content 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. Some user agents examine the content and, in certain cases, override the received type (for example, see [Sniffing]). This "MIME sniffing" risks drawing incorrect conclusions about the data, which might expose the user to additional security risks (e.g., "privilege escalation"). Furthermore, distinct media types often share a common data format, differing only in how the data is intended to be processed, which is impossible to distinguish by inspecting the data alone. When sniffing is implemented, implementers are encouraged to provide a means for the user to disable it.

Although Content-Type is defined as a singleton field, it is sometimes incorrectly generated multiple times, resulting in a combined field value that appears to be a list. Recipients often attempt to handle this error by using the last syntactically valid member of the list, leading to potential interoperability and security issues if different implementations have different error handling behaviors.

8.3.1. Media Type

HTTP uses media types [RFC2046] in the Content-Type (Section 8.3) and Accept (Section 12.5.1) 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 the message context.

```
media-type = type "/" subtype parameters
type       = token
subtype    = token
```

The type and subtype tokens are case-insensitive.

The type/subtype MAY be followed by semicolon-delimited parameters (Section 5.6.6) in the form of name/value pairs. 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. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name.

For example, the following media types are equivalent in describing HTML text data encoded in the UTF-8 character encoding scheme, but the first is preferred for consistency (the "charset" parameter value is defined as being case-insensitive in [RFC2046], Section 4.1.2):

```
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].

8.3.2. Charset

HTTP uses *charset* names to indicate or negotiate the character encoding scheme ([RFC6365], Section 2) of a textual representation. In the fields defined by this document, charset names appear either in parameters (Content-Type), or, for Accept-Encoding, in the form of a plain token. In both cases, charset names are matched case-insensitively.

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 Section 2 of [RFC2978].

> **Note:** In theory, charset names are defined by the "mime-charset" ABNF rule defined in Section 2.3 of [RFC2978] (as corrected in [Err1912]). That rule allows two characters that are not included in "token" ("{" and "}"), but no charset name registered at the time of this writing includes braces (see [Err5433]).

8.3.3. 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 content. For example, the "multipart/form-data" type is often used for carrying form data in a request, as described in [RFC7578], and the "multipart/byteranges" type is defined by this specification for use in some 206 (Partial Content) responses (see Section 15.3.7).

8.4. 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 = #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. Note that the coding named "identity" is reserved for its special role in Accept-Encoding and thus SHOULD NOT be included.

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 [HTTP/1.1]), 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.

8.4.1. Content Codings

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

```
content-coding   = token
```

All content codings are case-insensitive and ought to be registered within the "HTTP Content Coding Registry", as described in Section 16.6

Content-coding values are used in the Accept-Encoding (Section 12.5.3) and Content-Encoding (Section 8.4) header fields.

8.4.1.1. 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".

8.4.1.2. 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.

8.4.1.3. 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".

8.5. 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 = #language-tag
```

Language tags are defined in Section 8.5.1. 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.

8.5.1. 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 12.5.4, 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.

8.6. Content-Length

The "Content-Length" header field indicates the associated representation's data length as a decimal non-negative integer number of octets. When transferring a representation as content, Content-Length refers specifically to the amount of data enclosed so that it can be used to delimit framing (e.g., Section 6.2 of [HTTP/1.1]). In other cases, Content-Length indicates the selected representation's current length, which can be used by recipients to estimate transfer time or to compare with previously stored representations.

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

An example is

```
Content-Length: 3495
```

A user agent SHOULD send Content-Length in a request when the method defines a meaning for enclosed content and it is not sending Transfer-Encoding. For example, a user agent normally sends Content-Length in a POST request even when the value is 0 (indicating empty content). A user agent SHOULD NOT send a Content-Length header field when the request message does not contain content and the method semantics do not anticipate such data.

A server MAY send a Content-Length header field in a response to a HEAD request (Section 9.3.2); 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 content 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 15.4.5); 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 content 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 9.3.6).

Aside from the cases defined above, in the absence of Transfer-Encoding, an origin server SHOULD send a Content-Length header field when the content 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 content, a recipient MUST anticipate potentially large decimal numerals and prevent parsing errors due to integer conversion overflows or precision loss due to integer conversion (Section 17.5).

Because Content-Length is used for message delimitation in HTTP/1.1, its field value can impact how the message is parsed by downstream recipients even when the immediate connection is not using HTTP/1.1. If the message is forwarded by a downstream intermediary, a Content-Length field value that is inconsistent with the received message framing might cause a security failure due to request smuggling or response splitting.

As a result, a sender MUST NOT forward a message with a Content-Length header field value that is known to be incorrect.

Likewise, a sender MUST NOT forward a message with a Content-Length header field value that does not match the ABNF above, with one exception: a recipient of a Content-Length header field value consisting of

the same decimal value repeated as a comma-separated list (e.g, "Content-Length: 42, 42") MAY either reject the message as invalid or replace that invalid field value with a single instance of the decimal value, since this likely indicates that a duplicate was generated or combined by an upstream message processor.

8.7. 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 content. 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 content in this message.

```
Content-Location = absolute-URI / partial-URI
```

The field value is either an absolute-URI or a partial-URI. In the latter case (Section 4), the referenced URI is relative to the target URI ([URI], Section 5).

The Content-Location value is not a replacement for the target URI (Section 7.1). 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 target URI, then the recipient MAY consider the content to be a current representation of that resource at the time indicated by the message origination date. For a GET (Section 9.3.1) or HEAD (Section 9.3.2) 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 9.3.4) or POST (Section 9.3.3), 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 target 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 target 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 content is a current representation of the newly created resource.
- Otherwise, such a Content-Location indicates that this content 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 content 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.

8.8. Validator Fields

Resource metadata is referred to as a *validator* if it can be used within a precondition (Section 13.1) to make a conditional request (Section 13). Validator fields convey a current validator for the selected representation (Section 3.2).

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 method and status code semantics, the selected representation for a given response is not necessarily the same as the representation enclosed as response content.

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 field in a 201 (Created) response communicates the entity tag of the newly created resource's representation, so that the entity tag can be used as a validator in later conditional requests to prevent the "lost update" problem.

This specification defines two forms of metadata that are commonly used to observe resource state and test for preconditions: modification dates (Section 8.8.2) and opaque entity tags (Section 8.8.3). Additional metadata that reflects resource state has been defined by various extensions of HTTP, such as Web Distributed Authoring and Versioning [WEBDAV], that are beyond the scope of this specification.

8.8.1. 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 content 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 (e.g., 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.

8.8.2. 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
```

8.8.2.1. 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]) can substantially reduce unnecessary transfers and significantly improve service availability and scalability.

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.

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 (as defined in Section 5.6.7) MUST NOT generate a Last-Modified date that is later than the server's time of message origination (Date, Section 6.6.1). 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 generate a Last-Modified date for a response unless that date value was assigned to the resource by some other system (presumably one with a clock).

8.8.2.2. 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:

- The validator is being compared by an origin server to the actual current validator for the representation and,
- That origin server reliably knows that the associated representation did not change twice during the second covered by the presented validator;

or

- 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
- That cache entry includes a Date value which is at least one second after the Last-Modified value and the client has reason to believe that they were generated by the same clock or that there is enough difference between the Last-Modified and Date values to make clock synchronization issues unlikely;

or

- The validator is being compared by an intermediate cache to the validator stored in its cache entry for the representation, and
- That cache entry includes a Date value which is at least one second after the Last-Modified value and the cache has reason to believe that they were generated by the same clock or that there is enough difference between the Last-Modified and Date values to make clock synchronization issues unlikely.

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.

8.8.3. ETag

The "ETag" 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       = %s"W/"
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 8.8.1), then the origin server MUST mark the entity tag as weak by prefixing its opaque value with "W/" (case-sensitive).

A sender MAY send the ETag field in a trailer section (see Section 6.5). However, since trailers are often ignored, it is preferable to send ETag as a header field unless the entity tag is generated while sending the content.

8.8.3.1. 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 substantially reduce unnecessary transfers and significantly improve service availability, scalability, and reliability.

8.8.3.2. 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

ETag 1	ETag 2	Strong Comparison	Weak Comparison
"1"	"1"	match	match

Table 3

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

Consider a resource that is subject to content negotiation (Section 12), and where the representations sent in response to a GET request vary based on the Accept-Encoding request header field (Section 12.5.3):

>> 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 [HTTP/1.1]) apply only during message transfer and do not result in distinct entity tags.

9. Methods

9.1. 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 if those additional semantics do not conflict with the method. For example, a client can send conditional request header fields (Section 13.1) to make the requested action conditional on the current state of the target resource.

HTTP is designed to be usable as an interface to distributed object systems. The request method invokes an action to be applied to a target resource in much the same way that a remote method invocation can be sent to an identified object.

```
method = token
```

The method token is case-sensitive because it might be used as a gateway to object-based systems with case-sensitive method names. By convention, standardized methods are defined in all-uppercase US-ASCII letters.

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.

Method Name	Description	Section
GET	Transfer a current representation of the target resource.	9.3.1
HEAD	Same as GET, but do not transfer the response content.	9.3.2
POST	Perform resource-specific processing on the request content.	9.3.3
PUT	Replace all current representations of the target resource with the request content.	9.3.4
DELETE	Remove all current representations of the target resource.	9.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	9.3.6
OPTIONS	Describe the communication options for the target resource.	9.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	9.3.8

Table 4

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 10.2.1). However, the set of allowed methods can change dynamically. An origin server that receives a request method that is unrecognized or not implemented SHOULD respond with the 501 (Not Implemented) status code. An origin server that receives a request method that is recognized and implemented, but not allowed for the target resource, SHOULD respond with the 405 (Method Not Allowed) status code.

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", as described in Section 16.1.

9.2. Common Method Properties

9.2.1. 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 cause the server to fail. 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 target 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.

9.2.2. 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.

A client SHOULD NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied.

For example, a user agent can repeat a POST request automatically if it knows (through design or configuration) that the request is safe for that resource. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

Some clients take a riskier approach and attempt to guess when an automatic retry is possible. For example, a client might automatically retry a POST request if the underlying transport connection closed before any part of a response is received, particularly if an idle persistent connection was used.

A proxy MUST NOT automatically retry non-idempotent requests. A client SHOULD NOT automatically retry a failed automatic retry.

9.2.3. Methods and Caching

For a cache to store and use a response, the associated method needs to explicitly allow caching and to detail under what conditions a response can be used to satisfy subsequent requests; a method definition that does not do so cannot be cached. For additional requirements see [CACHING].

This specification defines caching semantics for GET, HEAD, and POST, although the overwhelming majority of cache implementations only support GET and HEAD.

9.3. Method Definitions

9.3.1. GET

The GET method requests transfer of a current selected representation for the target resource. A successful response reflects the quality of "sameness" identified by the target URI (Section 1.2.2 of [URI]). Hence, retrieving identifiable information via HTTP is usually performed by making a GET request on an identifier associated with the potential for providing that information in a 200 (OK) response.

GET is the primary mechanism of information retrieval and the focus of almost all performance optimizations. Applications that produce a URI for each important resource can benefit from those optimizations while enabling their reuse by other applications, creating a network effect that promotes further expansion of the Web.

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 17.3 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 resource identifier corresponds to an implementation and how that implementation manages to select and send a current representation of the target resource.

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 14.2).

Although request message framing is independent of the method used, content received in a GET request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling

attack (Section 11.2 of [HTTP/1.1]). A client SHOULD NOT generate content in a GET request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

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]).

When information retrieval is performed with a mechanism that constructs a target URI from user-provided information, such as the query fields of a form using GET, potentially sensitive data might be provided that would not be appropriate for disclosure within a URI (see Section 17.9). In some cases, the data can be filtered or transformed such that it would not reveal such information. In others, particularly when there is no benefit from caching a response, using the POST method (Section 9.3.3) instead of GET can transmit such information in the request content rather than within the target URI.

9.3.2. HEAD

The HEAD method is identical to GET except that the server MUST NOT send content in the response. HEAD is used to obtain metadata about the selected representation without transferring its representation data, often for the sake of testing hypertext links or finding recent modifications.

The server SHOULD send the same header fields in response to a HEAD request as it would have sent if the request method had been GET. However, a server MAY omit header fields for which a value is determined only while generating the content. For example, some servers buffer a dynamic response to GET until a minimum amount of data is generated so that they can more efficiently delimit small responses or make late decisions with regard to content selection. Such a response to GET might contain Content-Length and Vary fields, for example, that are not generated within a HEAD response. These minor inconsistencies are considered preferable to generating and discarding the content for a HEAD request, since HEAD is usually requested for the sake of efficiency.

Although request message framing is independent of the method used, content received in a HEAD request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack (Section 11.2 of [HTTP/1.1]). A client SHOULD NOT generate content in a HEAD request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

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 affect previously cached responses to GET; see Section 4.3.5 of [CACHING].

9.3.3. 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 could 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 10.2.2) 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]) and a Content-Location header field that has the same value as the POST's target URI (Section 8.7). A cached POST response can be reused to satisfy a later GET or HEAD request. In contrast, a POST request cannot be satisfied by a cached POST response because POST is potentially unsafe; see Section 4 of [CACHING].

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.

9.3.4. 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 content. 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 verify that the PUT representation is consistent with its configured constraints for the target resource. For example, if an origin server determines a resource's representation metadata based on the URI, then the origin server needs to ensure that the content received in a successful PUT request is consistent with that metadata. 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.

This extends to how header and trailer fields are stored; while common header fields like Content-Type will typically be stored and returned upon subsequent GET requests, header and trailer field handling is specific to the resource that received the request. As a result, an origin server SHOULD ignore unrecognized header and trailer fields received in a PUT request (i.e., not save them as part of the resource state).

An origin server MUST NOT send a validator field (Section 8.8), 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 content (i.e., the resource's new representation data is identical to the content 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 it sent (and retains in memory) is the result of the PUT, and thus it doesn't need to be retrieved again from the origin server. The new validator(s) received in the response can be used for future conditional requests in order to prevent accidental overwrites (Section 13.1).

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.

Some origin servers support use of the Content-Range header field (Section 14.4) as a request modifier to perform a partial PUT, as described in Section 14.5.

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 target URI, those stored responses will be invalidated (see Section 4.4 of [CACHING]).

9.3.5. 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.

Although request message framing is independent of the method used, content received in a DELETE request has no generally defined semantics, cannot alter the meaning or target of the request, and might lead some implementations to reject the request and close the connection because of its potential as a request smuggling attack (Section 11.2 of [HTTP/1.1]). A client SHOULD NOT generate content in a DELETE request unless it is made directly to an origin server that has previously indicated, in or out of band, that such a request has a purpose and will be adequately supported. An origin server SHOULD NOT rely on private agreements to receive content, since participants in HTTP communication are often unaware of intermediaries along the request chain.

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

9.3.6. 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 data, 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, [TLS13]).

CONNECT uses a special form of request target, unique to this method, consisting of only the host and port number of the tunnel destination, separated by a colon. There is no default port; a client MUST send the port number even if the CONNECT request is based on a URI reference that contains an authority component with an elided port (Section 4.1). For example,

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

A server MUST reject a CONNECT request that targets an empty or invalid port number, typically by responding with a 400 (Bad Request) status code.

Because CONNECT changes the request/response nature of an HTTP connection, specific HTTP versions might have different ways of mapping its semantics into the protocol's wire format.

CONNECT is intended for use in requests to a proxy. The recipient can establish a tunnel either by directly connecting to the server identified by the request target or, if configured to use another proxy, by forwarding the CONNECT request to the next inbound proxy. An origin server MAY accept a CONNECT request, but most origin servers do not implement CONNECT.

Any 2xx (Successful) response indicates that the sender (and all inbound proxies) will switch to tunnel mode immediately after the response header section; data received after that header section 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.

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:443 HTTP/1.1
Host: server.example.com:443
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 "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 list 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 CONNECT request message does not have content. The interpretation of data sent after the header section of the CONNECT request message is specific to the version of HTTP in use.

Responses to the CONNECT method are not cacheable.

9.3.7. 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 7.1) 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 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 content, 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 client MAY send a Max-Forwards header field in an OPTIONS request to target a specific recipient in the request chain (see Section 7.6.2). 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 content MUST send a valid Content-Type header field describing the representation media type. Note that this specification does not define any use for such content.

Responses to the OPTIONS method are not cacheable.

9.3.8. 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 content of a 200 (OK) response. The "message/http" format (Section 10.1 of [HTTP/1.1]) is one way to do so. 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 7.6.2).

A client MUST NOT generate 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 11) or cookies [COOKIE] in a TRACE request. The final recipient of the request SHOULD exclude any request fields that are likely to contain sensitive data when that recipient generates the response content.

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 7.6.3) 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 content in a TRACE request.

Responses to the TRACE method are not cacheable.

10. Message Context

10.1. Request Context Fields

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

10.1.1. 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.

```
Expect =      #expectation
expectation = token [ "=" ( token / quoted-string ) parameters ]
```

The Expect field value is case-insensitive.

The only expectation defined by this specification is "100-continue" (with no defined parameters).

A server that receives an Expect field value containing a member 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 (presumably large) content in this request and wishes to receive a 100 (Continue) interim response if the method, target URI, 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 content before actually doing so, which can improve efficiency when the data 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 content.
- A client that will wait for a 100 (Continue) response before sending the request content 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 content 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 content.
- 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 content for the corresponding request, or if the framing indicates that there is no content.
- A server that sends a 100 (Continue) response MUST ultimately send a final status code, once it receives and processes the request content, unless the connection is closed prematurely.
- A server that responds with a final status code before reading the entire request content SHOULD indicate whether it intends to close the connection (e.g., see Section 9.6 of [HTTP/1.1]) or continue reading the request content.

Upon receiving an HTTP/1.1 (or later) request that has a method, target URI, and complete header section that contains a 100-continue expectation and an indication that request content will follow, an origin server MUST send either:

- an immediate response with a final status code, if that status can be determined by examining just the method, target URI, and header fields, or
- an immediate 100 (Continue) response to encourage the client to send the request content.

The origin server MUST NOT wait for the content before sending the 100 (Continue) response.

Upon receiving an HTTP/1.1 (or later) request that has a method, target URI, and complete header section that contains a 100-continue expectation and indicates a request content will follow, a proxy MUST either:

- send an immediate response with a final status code, if that status can be determined by examining just the method, target URI, and header fields, or
- forward 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 content.

10.1.2. 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: spider-admin@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 its value is expected to be visible to anyone receiving or observing the request and is often recorded within logfiles and error reports without any expectation of privacy.

10.1.3. 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 [URI], if any, when generating the Referer field value.

```
Referer = absolute-URI / partial-URI
```

The field value is either an absolute-URI or a partial-URI. In the latter case (Section 4), the referenced URI is relative to the target URI ([URI], Section 5).

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 header field or send it with a value of "about:blank".

The Referer header field value need not convey the full URI of the referring resource; a user agent MAY truncate parts other than the referring origin.

The Referer header 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 SHOULD NOT send a Referer header field if the referring resource was accessed with a secure protocol and the request target has an origin differing from that of the referring resource, unless the referring resource explicitly allows Referer to be sent. A user agent MUST NOT send a Referer header field in an unsecured HTTP request if the referring resource was accessed with a secure protocol. See Section 17.9 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 target URI.

10.1.4. TE

The "TE" header field describes capabilities of the client with regard to transfer codings and trailer sections.

As described in Section 6.5, a TE field with a "trailers" member sent in a request indicates that the client will not discard trailer fields.

TE is also used within HTTP/1.1 to advise servers about which transfer codings the client is able to accept in a response. As of publication, only HTTP/1.1 uses transfer codings (see Section 7 of [HTTP/1.1]).

The TE field value is a list of members, with each member (aside from "trailers") consisting of a transfer coding name token with an optional weight indicating the client's relative preference for that transfer coding (Section 12.4.2) and optional parameters for that transfer coding.

```
TE                = #t-codings
t-codings         = "trailers" / ( transfer-coding [ weight ] )
transfer-coding   = token *( OWS ";" OWS transfer-parameter )
transfer-parameter = token BWS "=" BWS ( token / quoted-string )
```

A sender of TE MUST also send a "TE" connection option within the Connection header field (Section 7.6.1) to inform intermediaries not to forward this field.

10.1.5. 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 header 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 5.6.5), 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 header 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.

10.2. Response Context Fields

The response header fields below provide additional information about the response, beyond what is implied by the status code, including information about the server, about the target resource, or about related resources.

10.2.1. 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 header 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.

10.2.2. 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 ([URI], Section 4.2), the final value is computed by resolving it against the target URI ([URI], 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 target URI (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 header 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. A Location field value cannot allow a list of members because the comma list separator is a valid data character within a URI-reference. If an invalid message is sent with multiple Location field lines, a recipient along the path might combine those field lines into one value. Recovery of a valid Location field value from that situation is difficult and not interoperable across implementations.
>
> **Note:** The Content-Location header field (Section 8.7) 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.

10.2.3. 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 Retry-After field value can be either an HTTP-date or a number of seconds to delay after receiving the response.

```
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.

10.2.4. 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 header field in its responses.

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

The Server header field value consists of one or more product identifiers, each followed by zero or more comments (Section 5.6.5), 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 10.1.5.

Example:

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

An origin server SHOULD NOT generate a Server header 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. HTTP Authentication

11.1. Authentication Scheme

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. It uses a case-insensitive token to identify the authentication scheme:

```
auth-scheme    = token
```

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 [RFC7617] and [RFC7616], respectively.

11.2. Authentication Parameters

The authentication scheme is followed by additional information necessary for achieving authentication via that scheme as either a comma-separated list of parameters or a single sequence of characters capable of holding base64-encoded information.

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

The token68 syntax allows the 66 unreserved URI characters ([URI]), 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]).

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

Parameter values can be expressed either as "token" or as "quoted-string" (Section 5.6). Authentication scheme definitions need to accept both notations, both for senders and recipients, to allow recipients to use generic parsing components regardless of the authentication scheme.

For backwards compatibility, authentication scheme definitions can restrict the format for senders to one of the two variants. This can be important when it is known that deployed implementations will fail when encountering one of the two formats.

11.3. Challenge and Response

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.

11.4. Credentials

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 17.16.1.

```
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 15.5.4).

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.

Note that various custom mechanisms for user authentication use the Set-Cookie and Cookie header fields, defined in [COOKIE], for passing tokens related to authentication.

11.5. Establishing a 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 origin (see Section 4.3.1) 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).

The extent of a protection space, and therefore the requests to which credentials might be automatically applied, is not necessarily known to clients without additional information. An authentication scheme might define parameters that describe the extent of a protection space. 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.

11.6. Authenticating Users to Origin Servers

11.6.1. WWW-Authenticate

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

```
WWW-Authenticate = #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 header 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: Basic realm="simple", Newauth realm="apps",
                  type=1, title="Login to \"apps\""
```

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

Some user agents do not recognize this form, however. As a result, sending a WWW-Authenticate field value with more than one member on the same field line might not be interoperable.

> **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.

11.6.2. 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 header fields in that request. See Section 3.5 of [CACHING] for details of and requirements pertaining to handling of the Authorization header field by HTTP caches.

11.6.3. Authentication-Info

HTTP authentication schemes can use the "Authentication-Info" response field to communicate information after the client's authentication credentials have been accepted. This information can include a finalization message from the server (e.g., it can contain the server authentication).

The field value is a list of parameters (name/value pairs), using the "auth-param" syntax defined in Section 11.3. This specification only describes the generic format; authentication schemes using Authentication-Info will define the individual parameters. The "Digest" Authentication Scheme, for instance, defines multiple parameters in Section 3.5 of [RFC7616].

```
Authentication-Info = #auth-param
```

The Authentication-Info field can be used in any HTTP response, independently of request method and status code. Its semantics are defined by the authentication scheme indicated by the Authorization header field (Section 11.6.2) of the corresponding request.

A proxy forwarding a response is not allowed to modify the field value in any way.

Authentication-Info can be sent as a trailer field (Section 6.5) when the authentication scheme explicitly allows this.

11.7. Authenticating Clients to Proxies

11.7.1. 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 request. A proxy MUST send at least one Proxy-Authenticate header field in each 407 (Proxy Authentication Required) response that it generates.

```
Proxy-Authenticate = #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 11.6.1 for details.

11.7.2. 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 header 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.

11.7.3. Proxy-Authentication-Info

The "Proxy-Authentication-Info" response header field is equivalent to Authentication-Info, except that it applies to proxy authentication (Section 11.3) and its semantics are defined by the authentication scheme indicated by the Proxy-Authorization header field (Section 11.7.2) of the corresponding request:

```
Proxy-Authentication-Info = #auth-param
```

However, unlike Authentication-Info, the Proxy-Authentication-Info 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-Authentication-Info is being forwarded because each proxy will send the same field value.

Proxy-Authentication-Info can be sent as a trailer field (Section 6.5) when the authentication scheme explicitly allows this.

12. Content Negotiation

When responses convey content, 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 three patterns of content negotiation that can be made visible within the protocol: "proactive" negotiation, where the server selects the representation based upon the user agent's stated preferences; "reactive" negotiation, where the server provides a list of representations for the user agent to choose from; and "request content" negotiation, where the user agent selects the representation for a future request based upon the server's stated preferences in past responses.

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.

12.1. 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 header fields below 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 (when that "best guess" is good enough for the user, this avoids the round-trip delay of a subsequent request). 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 12.5.5) is often sent in a response subject to proactive negotiation to indicate what parts of the request information were used in the selection algorithm.

The request header fields Accept, Accept-Charset, Accept-Encoding, and Accept-Language are defined below for a user agent to engage in proactive negotiation of the response content. 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.

12.2. Reactive Negotiation

With *reactive negotiation* (a.k.a., *agent-driven negotiation*), selection of content (regardless of the status code) is performed by the user agent after receiving an initial response. The mechanism for reactive negotiation might be as simple as a list of references to alternative representations.

If the user agent is not satisfied by the initial response content, it can perform a GET request on one or more of the alternative resources to obtain a different representation. Selection of such alternatives might be performed automatically (by the user agent) or manually (e.g., by the user selecting from a hypertext menu).

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 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.

12.3. Request Content Negotiation

When content negotiation preferences are sent in a server's response, the listed preferences are called *request content negotiation* because they intend to influence selection of an appropriate content for subsequent requests to that resource. For example, the Accept (Section 12.5.1) and Accept-Encoding (Section 12.5.3) header fields can be sent in a response to indicate preferred media types and content codings for subsequent requests to that resource.

Similarly, Section 3.1 of [RFC5789] defines the "Accept-Patch" response header field, which allows discovery of which content types are accepted in PATCH requests.

12.4. Content Negotiation Field Features

12.4.1. Absence

For each of the content negotiation fields, a request that does not contain the field implies that the sender has no preference on that dimension of negotiation.

If a content negotiation header field is present in a request and none of the available representations for the response can be considered acceptable according to it, 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 for that request header field. This does not imply, however, that the client will be able to use the representation.

> **Note:** A user agent sending these header fields makes it easier for a server to identify an individual by virtue of the user agent's request characteristics (Section 17.13).

12.4.2. Quality Values

The content negotiation fields defined by this specification 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.

12.4.3. Wildcard Values

Most of these header fields, where indicated, define a wildcard value ("*") to select unspecified values. If no wildcard is present, values that are not explicitly mentioned in the field are considered unacceptable. Within Vary, the wildcard value means that the variance is unlimited.

> **Note:** In practice, using wildcards in content negotiation has limited practical value because it is seldom useful to say, for example, "I prefer image/* more or less than (some other specific value)". By sending Accept: */*;q=0, clients can explicitly request a 406 (Not Acceptable) response if a more preferred format is not available, but they still need to be able to handle a different response since the server is allowed to ignore their preference.

12.5. Content Negotiation Fields

12.5.1. Accept

The "Accept" header field can be used by user agents to specify their preferences regarding response media types. For example, 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.

When sent by a server in a response, Accept provides information about which content types are preferred in the content of a subsequent request to the same resource.

```
Accept = #( media-range [ weight ] )

media-range    = ( "*/*"
                 / ( type "/" "*" )
                 / ( type "/" subtype )
                ) parameters
```

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 optional applicable media type parameters (e.g., charset), followed by an optional "q" parameter for indicating a relative weight (Section 12.4.2).

Previous specifications allowed additional extension parameters to appear after the weight parameter. The accept extension grammar (accept-params, accept-ext) has been removed because it had a complicated definition, was not being used in practice, and is more easily deployed through new header fields. Senders

using weights SHOULD send "q" last (after all media-range parameters). Recipients SHOULD process any parameter named "q" as weight, regardless of parameter ordering.

> **Note:** Use of the "q" parameter name to control content negotiation would interfere with any media type parameter having the same name. Hence, the media type registry disallows parameters 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 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/plain;q=0.7, text/plain;format=flowed,
       text/plain;format=fixed;q=0.4, */*;q=0.5
```

would cause the following values to be associated:

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

Table 5

> **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.

12.5.2. Accept-Charset

The "Accept-Charset" header field can be sent by a user agent to indicate its preferences for charsets in textual response content. For example, 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 = #( ( token / "*" ) [ weight ] )
```

Charset names are defined in Section 8.3.2. A user agent MAY associate a quality value with each charset to indicate the user's relative preference for that charset, as defined in Section 12.4.2. An example is

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

The special value "*", if present in the Accept-Charset header field, matches every charset that is not mentioned elsewhere in the field.

> **Note:** Accept-Charset is deprecated because UTF-8 has become nearly ubiquitous and sending a detailed list of user-preferred charsets wastes bandwidth, increases latency, and makes passive fingerprinting far too easy (Section 17.13). Most general-purpose user agents do not send Accept-Charset unless specifically configured to do so.

12.5.3. Accept-Encoding

The "Accept-Encoding" header field can be used to indicate preferences regarding the use of content codings (Section 8.4.1).

When sent by a user agent in a request, Accept-Encoding indicates the content codings acceptable in a response.

When sent by a server in a response, Accept-Encoding provides information about which content codings are preferred in the content of a subsequent request to the same resource.

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 (weight) representing the preference for that encoding, as defined in Section 12.4.2. The asterisk "*" symbol in an Accept-Encoding field matches any available content coding not explicitly listed in the field.

Examples:

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

1. If no Accept-Encoding header 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 header 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 value, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in Section 12.4.2, a qvalue of 0 means "not acceptable".)

A representation could be encoded with multiple content codings. However, most content codings are alternative ways to accomplish the same purpose (e.g., data compression). When selecting between multiple content codings that have the same purpose, the acceptable content coding with the highest non-zero qvalue is preferred.

An Accept-Encoding header field with a field value that is empty implies that the user agent does not want any content coding in response. If a non-empty 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 unless the identity coding is indicated as unacceptable.

When the Accept-Encoding header field is present in a response, it indicates what content codings the resource was willing to accept in the associated request. The field value is evaluated the same way as in a request.

Note that this information is specific to the associated request; the set of supported encodings might be different for other resources on the same server and could change over time or depend on other aspects of the request (such as the request method).

Servers that fail a request due to an unsupported content coding ought to respond with a 415 (Unsupported Media Type) status and include an Accept-Encoding header field in that response, allowing clients to distinguish between issues related to content codings and media types. In order to avoid confusion with issues related to media types, servers that fail a request with a 415 status for reasons unrelated to content codings MUST NOT include the Accept-Encoding header field.

The most common use of Accept-Encoding is in responses with a 415 (Unsupported Media Type) status code, in response to optimistic use of a content coding by clients. However, the header field can also be used to indicate to clients that content codings are supported in order to optimize future interactions. For example, a resource might include it in a 2xx (Successful) response when the request content was big enough to justify use of a compression coding but the client failed do so.

12.5.4. 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 8.5.1.

```
Accept-Language = #( 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 12.4.2. 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".

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 17.13).

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.

12.5.5. Vary

The "Vary" header field in a response describes what parts of a request message, aside from the method and target URI, might have influenced the origin server's process for selecting the content of this response.

```
Vary = #( "*" / field-name )
```

A Vary field value is either the wildcard member "*" or a list of request field names, known as the selecting header fields, that might have had a role in selecting the representation for this response. Potential selecting header fields are not limited to fields defined by this specification.

A list containing the member "*" signals that other aspects of the request might have played a role in selecting the response representation, possibly including aspects 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 "*" in a Vary field value.

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 header fields (or lack thereof) as determining factors while choosing the content for this response.

A Vary field containing a list of field names has 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 header fields as the original request (Section 4.1 of [CACHING]) or reuse of the response has been validated by the origin server. 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 was subject to content negotiation (Section 12) and a different representation might be sent in a subsequent request if other values are provided in the listed header fields (proactive negotiation).

An origin server SHOULD generate a Vary header field on a cacheable response when it wishes that response to be selectively reused for subsequent requests. Generally, that is the case when the response content has been tailored to better fit the preferences expressed by those selecting header fields, such as when an origin server has selected the response's language based on the request's Accept-Language header field.

Vary might be elided when an origin server considers variance in content selection to be less significant than Vary's performance impact on caching, particularly when reuse is already limited by cache response directives (Section 5.2 of [CACHING]).

There is no need to send the Authorization field name in Vary because reuse of that response for a different user is prohibited by the field definition (Section 11.6.2). Likewise, if the response content has been selected or influenced by network region, but the origin server wants the cached response to be reused even if recipients move from one region to another, then there is no need for the origin server to indicate such variance in Vary.

13. Conditional Requests

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 13.2 defines when to evaluate preconditions and their order of precedence 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.

13.1. Preconditions

Preconditions are usually defined with respect to a 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). If a resource has multiple current representations, each with its own observable state, a precondition will assume that the mapping of each request to a selected representation (Section 3.2) is consistent over time. Regardless, if the mapping is inconsistent or the server is unable to select an appropriate representation, then no harm will result when the precondition evaluates to false.

Each precondition defined below 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 8.8). 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 13.2.

Other preconditions, defined by other specifications as extension fields, 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 ([WEBDAV], Section 10.4).

Extensibility of preconditions is only possible when the precondition can be safely ignored if unknown (like If-Modified-Since), when deployment can be assumed for a given use case, or when implementation is signaled by some other property of the target resource. This encourages a focus on mutually agreed deployment of common standards.

13.1.1. 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 8.8.3.2), 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 = "*" / #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). In general, it can be used with any method that involves the selection or modification of a

representation to abort the request if the selected representation's current entity tag is not a member within the If-Match field value.

When an origin server receives a request that selects a representation and that request includes an If-Match header field, the origin server MUST evaluate the If-Match condition per Section 13.2 prior to performing the method.

To evaluate a received If-Match header field:

1. If the field value is "*", the condition is true if the origin server has a current representation for the target resource.
2. If the field value is a list of entity tags, the condition is true if any of the listed tags match the entity tag of the selected representation.
3. Otherwise, the condition is false.

An origin server that evaluates an If-Match condition MUST NOT perform the requested method if the condition evaluates to false. Instead, the origin server MAY indicate that the conditional request failed by responding with a 412 (Precondition Failed) status code. Alternatively, if the request is a state-changing operation that appears to have already been applied to the selected representation, the origin server MAY respond with a 2xx (Successful) status code (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 an equivalent change was made by some other user agent).

Allowing an origin server to send a success response when a change request appears to have already been applied is more efficient for many authoring use cases, but comes with some risk if multiple user agents are making change requests that are very similar but not cooperative. For example, multiple user agents writing to a common resource as a semaphore (e.g., a nonatomic increment) are likely to collide and potentially lose important state transitions. For those kinds of resources, an origin server is better off being stringent in sending 412 for every failed precondition on an unsafe method. In other cases, excluding the ETag field from a success response might encourage the user agent to perform a GET as its next request to eliminate confusion about the resource's current state.

A client MAY send an If-Match header field in a GET request to indicate that it would prefer a 412 (Precondition Failed) response if the selected representation does not match. However, this is only useful in range requests (Section 14) for completing a previously received partial representation when there is no desire for a new representation. If-Range (Section 13.1.5) is better suited for range requests when the client prefers to receive a new representation.

A cache or intermediary MAY ignore If-Match because its interoperability features are only necessary for an origin server.

Note that an If-Match header field with a list value containing "*" and other values (including other instances of "*") is syntactically invalid (therefore not allowed to be generated) and furthermore is unlikely to be interoperable.

13.1.2. 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 8.8.3.2), since weak entity tags can be used for cache validation even if there have been changes to the representation data.

```
If-None-Match = "*" / #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 9.2.1). 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.

When an origin server receives a request that selects a representation and that request includes an If-None-Match header field, the origin server MUST evaluate the If-None-Match condition per Section 13.2 prior to performing the method.

To evaluate a received If-None-Match header field:

1. If the field value is "*", the condition is false if the origin server has a current representation for the target resource.
2. If the field value is a list of entity tags, the condition is false if one of the listed tags matches the entity tag of the selected representation.
3. Otherwise, the condition is true.

An origin server that evaluates an If-None-Match condition 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].

Note that an If-None-Match header field with a list value containing "*" and other values (including other instances of "*") is syntactically invalid (therefore not allowed to be generated) and furthermore is unlikely to be interoperable.

13.1.3. 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, the field value has more than one member, or if the request method is neither GET nor HEAD.

A recipient MUST ignore the If-Modified-Since header field if the resource does not have a modification date available.

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 header 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 clock time at which the message was received, particularly when the cached message does not contain a Last-Modified header 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 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 header field will not be able to help the user agent limit its data transfers to only those changed during the specified window.

When an origin server receives a request that selects a representation and that request includes an If-Modified-Since header field without an If-None-Match header field, the origin server SHOULD evaluate the If-Modified-Since condition per Section 13.2 prior to performing the method.

To evaluate a received If-Modified-Since header field:

1. If the selected representation's last modification date is earlier or equal to the date provided in the field value, the condition is false.
2. Otherwise, the condition is true.

An origin server that evaluates an If-Modified-Since condition SHOULD NOT perform the requested method if the condition evaluates to false; 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].

13.1.4. 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 (including when the field value appears to be a list of dates).

A recipient MUST ignore the If-Unmodified-Since header field if the resource does not have a modification date available.

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). In general, it can be used with any method that involves the selection or modification of a representation to abort the request if the selected representation's last modification date has changed since the date provided in the If-Unmodified-Since field value.

When an origin server receives a request that selects a representation and that request includes an If-Unmodified-Since header field without an If-Match header field, the origin server MUST evaluate the If-Unmodified-Since condition per Section 13.2 prior to performing the method.

To evaluate a received If-Unmodified-Since header field:

1. If the selected representation's last modification date is earlier than or equal to the date provided in the field value, the condition is true.
2. Otherwise, the condition is false.

An origin server that evaluates an If-Unmodified-Since condition MUST NOT perform the requested method if the condition evaluates to false. Instead, the origin server MAY indicate that the conditional request failed by responding with a 412 (Precondition Failed) status code. Alternatively, if the request is a state-changing operation that appears to have already been applied to the selected representation, the origin server MAY respond with a 2xx (Successful) status code (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 an equivalent change was made by some other user agent).

Allowing an origin server to send a success response when a change request appears to have already been applied is more efficient for many authoring use cases, but comes with some risk if multiple user agents are making change requests that are very similar but not cooperative. In those cases, an origin server is better off being stringent in sending 412 for every failed precondition on an unsafe method.

A client MAY send an If-Unmodified-Since header field in a GET request to indicate that it would prefer a 412 (Precondition Failed) response if the selected representation has been modified. However, this is only useful in range requests (Section 14) for completing a previously received partial representation when there is no desire for a new representation. If-Range (Section 13.1.5) is better suited for range requests when the client prefers to receive a new representation.

A cache or intermediary MAY ignore If-Unmodified-Since because its interoperability features are only necessary for an origin server.

13.1.5. 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 valid entity-tag can be distinguished from a valid HTTP-date by examining the first three characters for a DQUOTE.

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 8.8.2.2.

A server that receives an If-Range header field on a Range request MUST evaluate the condition per Section 13.2 prior to performing the method.

To evaluate a received If-Range header field containing an HTTP-date:

1. If the HTTP-date validator provided is not a strong validator in the sense defined by Section 8.8.2.2, the condition is false.
2. If the HTTP-date validator provided exactly matches the Last-Modified field value for the selected representation, the condition is true.
3. Otherwise, the condition is false.

To evaluate a received If-Range header field containing an entity-tag:

1. If the entity-tag validator provided exactly matches the ETag field value for the selected representation using the strong comparison function (Section 8.8.3.2), the condition is true.
2. Otherwise, the condition is false.

A recipient of an If-Range header field MUST ignore the Range header field if the If-Range condition evaluates to false. Otherwise, the recipient SHOULD process the Range header field as requested.

Note that the If-Range comparison is by exact match, including when the validator is an HTTP-date, and so it differs from the "earlier than or equal to" comparison used when evaluating an If-Unmodified-Since conditional.

13.2. Evaluation of Preconditions

13.2.1. When to Evaluate

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 process the request content (if any) or 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, prior to processing the request content, would have been a status code other than a 2xx (Successful) or 412 (Precondition Failed). In other words, redirects and failures that can be detected before significant processing occurs take precedence over the evaluation of preconditions.

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.

Note that protocol extensions can modify the conditions under which preconditions are evaluated or the consequences of their evaluation. For example, the immutable cache directive (defined by [RFC8246]) instructs caches to forgo forwarding conditional requests when they hold a fresh response.

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.

13.2.2. Precedence of Preconditions

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:
2. When recipient is the origin server, If-Match is not present, and If-Unmodified-Since is present, evaluate the If-Unmodified-Since precondition:
3. When If-None-Match is present, evaluate the If-None-Match precondition:
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:
5. When the method is GET and both Range and If-Range are present, evaluate the If-Range precondition:
6. Otherwise,

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

14. Range Requests

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.

14.1. Range Units

Representation data can be partitioned into subranges when there are addressable structural units inherent to that data's content coding or media type. For example, octet (a.k.a. byte) boundaries are a structural unit common to all representation data, allowing partitions of the data to be identified as a range of bytes at some offset from the start or end of that data.

This general notion of a *range unit* is used in the Accept-Ranges (Section 14.3) response header field to advertise support for range requests, the Range (Section 14.2) request header field to delineate the parts of a representation that are requested, and the Content-Range (Section 14.4) header field to describe which part of a representation is being transferred.

```
range-unit       = token
```

All range unit names are case-insensitive and ought to be registered within the "HTTP Range Unit Registry", as defined in Section 16.5.1.

Range units are intended to be extensible, as described in Section 16.5.

14.1.1. Range Specifiers

Ranges are expressed in terms of a range unit paired with a set of range specifiers. The range unit name determines what kinds of range-spec are applicable to its own specifiers. Hence, the following grammar is generic: each range unit is expected to specify requirements on when int-range, suffix-range, and other-range are allowed.

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

```
ranges-specifier = range-unit "=" range-set
range-set        = 1#range-spec
range-spec       = int-range
                 / suffix-range
                 / other-range
```

An int-range is a range expressed as two non-negative integers or as one non-negative integer through to the end of the representation data. The range unit specifies what the integers mean (e.g., they might indicate unit offsets from the beginning, inclusive numbered parts, etc.).

```
int-range     = first-pos "-" [ last-pos ]
first-pos     = 1*DIGIT
last-pos      = 1*DIGIT
```

An int-range is invalid if the last-pos value is present and less than the first-pos.

A suffix-range is a range expressed as a suffix of the representation data with the provided non-negative integer maximum length (in range units). In other words, the last N units of the representation data.

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

To provide for extensibility, the other-range rule is a mostly unconstrained grammar that allows application-specific or future range units to define additional range specifiers.

```
other-range   = 1*( %x21-2B / %x2D-7E )
              ; 1*(VCHAR excluding comma)
```

A ranges-specifier is invalid if it contains any range-spec that is invalid or undefined for the indicated range-unit.

A valid ranges-specifier is *satisfiable* if it contains at least one range-spec that is satisfiable, as defined by the indicated range-unit. Otherwise, the ranges-specifier is *unsatisfiable*.

14.1.2. Byte Ranges

The "bytes" range unit is used to express subranges of a representation data's octet sequence. Each byte range is expressed as an integer range at some offset, relative to either the beginning (int-range) or end (suffix-range) of the representation data. Byte ranges do not use the other-range specifier.

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

If the representation data has a content coding applied, each byte range is calculated with respect to the encoded sequence of bytes, not the sequence of underlying bytes that would be obtained after decoding.

Examples of bytes range specifiers:

- 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 client can limit the number of bytes requested without knowing the size of the selected representation. If the last-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-pos with a value that is one less than the current length of the selected representation).

A client can refer to the last N bytes (N > 0) of the selected representation using a suffix-range. 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
```

- The first, middle, and last 1000 bytes:

```
bytes= 0-999, 4500-5499, -1000
```

- 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
```

For a GET request, a valid bytes range-spec is satisfiable if it is either:

- an int-range with a first-pos that is less than the current length of the selected representation or
- a suffix-range with a non-zero suffix-length.

When a selected representation has zero length, the only satisfiable form of range-spec in a GET request is a suffix-range with a non-zero suffix-length.

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

14.2. 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 (Section 8.1), rather than the entire selected representation.

```
Range = ranges-specifier
```

A server MAY ignore the Range header field. However, origin servers and intermediate caches ought to support byte ranges when possible, since they support 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 that is unrecognized or for which range handling is not defined. For this specification, GET is the only method for which range handling is defined.

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 contains an invalid ranges-specifier (Section 14.1.1), a ranges-specifier with more than two overlapping ranges, or a set of many small ranges that are not listed in ascending order, since these are indications of either a broken client or a deliberate denial-of-service attack (Section 17.15). 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 server that supports range requests MAY ignore a Range header field when the selected representation has no content (i.e., the selected representation's data is of zero length).

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 13.1, 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 13.1.5) 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, the received Range field-value contains a valid ranges-specifier with a range-unit supported for that target resource, and that ranges-specifier is satisfiable with respect to the selected representation, the server SHOULD send a 206 (Partial Content) response with content containing one or more partial representations that correspond to the satisfiable range-spec(s) requested.

The above does not imply that a server will send all requested ranges. In some cases, it may only be possible (or efficient) to send a portion of the requested ranges first, while expecting the client to re-request the remaining portions later if they are still desired (see Section 15.3.7).

If all of the preconditions are true, the server supports the Range header field for the target resource, the received Range field-value contains a valid ranges-specifier, and either the range-unit is not supported for that target resource or the ranges-specifier is unsatisfiable with respect to the selected representation, the server SHOULD send a 416 (Range Not Satisfiable) response.

14.3. Accept-Ranges

The "Accept-Ranges" field in a response indicates whether an upstream server supports range requests for the target resource.

```
Accept-Ranges     = acceptable-ranges
acceptable-ranges = 1#range-unit
```

For example, a server that supports byte-range requests (Section 14.1.2) can send the field

```
Accept-Ranges: bytes
```

to indicate that it supports byte range requests for that target resource, thereby encouraging its use by the client for future partial requests on the same request path. Range units are defined in Section 14.1.

A client MAY generate range requests regardless of having received an Accept-Ranges field. The information only provides advice for the sake of improving performance and reducing unnecessary network transfers.

Conversely, a client MUST NOT assume that receiving an Accept-Ranges field means that future range requests will return partial responses. The content might change, the server might only support range requests at certain times or under certain conditions, or a different intermediary might process the next request.

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 on the same request path. The range unit "none" is reserved for this purpose.

The Accept-Ranges field MAY be sent in a trailer section, but is preferred to be sent as a header field because the information is particularly useful for restarting large information transfers that have failed in mid-content (before the trailer section is received).

14.4. 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 content, sent in each part of a multipart 206

response to indicate the range enclosed within each body part (Section 14.6), and sent in 416 (Range Not Satisfiable) responses to provide information about the selected representation.

```
Content-Range       = range-unit SP
                      ( range-resp / unsatisfied-range )

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

complete-length     = 1*DIGIT
```

If a 206 (Partial Content) response contains a Content-Range header field with a range unit (Section 14.1) 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.

Content-Range might also be sent as a request modifier to request a partial PUT, as described in Section 14.5, based on private agreements between client and origin server. A server MUST ignore a Content-Range header field received in a request with a method for which Content-Range support is not defined.

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 range-resp that has a last-pos value less than its first-pos value, or a complete-length value less than or equal to its last-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
```

14.5. Partial PUT

Some origin servers support PUT of a partial representation when the user agent sends a Content-Range header field (Section 14.4) in the request, though such support is inconsistent and depends on private agreements with user agents. In general, it requests that the state of the target resource be partly replaced with the enclosed content at an offset and length indicated by the Content-Range value, where the offset is relative to the current selected representation.

An origin server SHOULD respond with a 400 (Bad Request) status code if it receives Content-Range on a PUT for a target resource that does not support partial PUT requests.

Partial PUT is not backwards compatible with the original definition of PUT. It may result in the content being written as a complete replacement for the current representation.

Partial resource updates are also possible by targeting a separately identified resource with state that overlaps or extends 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]).

14.6. 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:	byterange
Required parameters:	boundary
Optional parameters:	N/A
Encoding considerations:	only "7bit", "8bit", or "binary" are permitted
Security considerations:	see Section 17
Interoperability considerations:	N/A
Published specification:	RFC 9110 (see Section 14.6)
Applications that use this media type:	HTTP componen supportin multiple ranges in a

	single request
Fragment identifier considerations:	N/A
Additional information:	Deprecate
	Magic nu
	File exten
	Macintosh
Person and email address to contact for further information:	See Authors' Addresses section.
Intended usage:	COMMO
Restrictions on usage:	N/A
Author:	See Authors' Addresses section.
Change controller:	IESG

15. Status Codes

The status code of a response is a three-digit integer code that describes the result of the request and the semantics of the response, including whether the request was successful and what content is enclosed (if any). All valid status codes are within the range of 100 to 599, inclusive.

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

HTTP status codes are extensible. A client is 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.

For example, if a client receives an unrecognized status code of 471, it can see from the first digit 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.

Values outside the range 100..599 are invalid. Implementations often use three-digit integer values outside of that range (i.e., 600..999) for internal communication of non-HTTP status (e.g., library errors). A client that receives a response with an invalid status code SHOULD process the response as if it had a 5xx (Server Error) status code.

A single request can have multiple associated responses: zero or more *interim* (non-final) responses with status codes in the "informational" (1xx) range, followed by exactly one *final* response with a status code in one of the other ranges.

15.1. 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 or left out altogether without affecting the protocol.

Responses with status codes that are defined as heuristically cacheable (e.g., 200, 203, 204, 206, 300, 301, 308, 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 heuristically cacheable.

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", as described in Section 16.2.

15.2. 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. 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 1xx response is terminated by the end of the header section; it cannot contain content or trailers.

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

15.2.1. 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 content, as described in Section 10.1.1. 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.

15.2.2. 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 7.8), 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 in effect after this 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.

15.3. Successful 2xx

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

15.3.1. 200 OK

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

Request Method	Response content is a representation of:
GET	the target resource
HEAD	the target resource, like GET, but without transferring the representation data
POST	the status of, or results obtained from, the action
PUT, DELETE	the status of the action
OPTIONS	communication options for the target resource
TRACE	the request message as received by the server returning the trace

Table 6

Aside from responses to CONNECT, a 200 response is expected to contain message content unless the message framing explicitly indicates that the content has zero length. If some aspect of the request indicates a preference for no content upon success, the origin server ought to send a 204 (No Content) response instead. For CONNECT, there is no content because the successful result is a tunnel, which begins immediately after the 200 response header section.

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

In 200 responses to GET or HEAD, an origin server SHOULD send any available validator fields (Section 8.8) for the selected representation, with both a strong entity tag and a Last-Modified date being preferred.

In 200 responses to state-changing methods, any validator fields (Section 8.8) sent in the response convey the current validators for the new representation formed as a result of successfully applying the request semantics. Note that the PUT method (Section 9.3.4) has additional requirements that might preclude sending such validators.

15.3.2. 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 header field is received, by the target URI.

The 201 response content typically describes and links to the resource(s) created. Any validator fields (Section 8.8) sent in the response convey the current validators for a new representation created by the request. Note that the PUT method (Section 9.3.4) has additional requirements that might preclude sending such validators.

15.3.3. 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.

15.3.4. 203 Non-Authoritative Information

The 203 (Non-Authoritative Information) status code indicates that the request was successful but the enclosed content has been modified from that of the origin server's 200 (OK) response by a transforming proxy (Section 7.7). 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).

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

15.3.5. 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 content. 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 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 end of the header section; it cannot contain content or trailers.

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

15.3.6. 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 content in a 205 response.

15.3.7. 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.

A server that supports range requests (Section 14) will usually attempt to satisfy all of the requested ranges, since sending less data will likely result in another client request for the remainder. However, a server might want to send only a subset of the data requested for reasons of its own, such as temporary unavailability, cache efficiency, load balancing, etc. Since a 206 response is self-descriptive, the client can still understand a response that only partially satisfies its range request.

A client MUST inspect a 206 response's Content-Type and Content-Range field(s) to determine what parts are enclosed and whether additional requests are needed.

A server that generates a 206 response 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.

A Content-Length header field present in a 206 response indicates the number of octets in the content of this message, which is usually not the complete length of the selected representation. Each Content-Range header field includes information about the selected representation's complete length.

A sender that generates a 206 response to a request with an If-Range header field SHOULD NOT generate other representation header fields beyond those required because the client already has a prior response containing those header fields. Otherwise, a 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 heuristically cacheable; i.e., unless otherwise indicated by explicit cache controls (see Section 4.2.2 of [CACHING]).

15.3.7.1. 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 content 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 ...
```

15.3.7.2. Multiple Parts

If multiple parts are being transferred, the server generating the 206 response MUST generate "multipart/byteranges" content, as defined in Section 14.6, 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 content, 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 header 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 range-spec appeared in the received Range header field. Since the typical overhead between each part of a "multipart/byteranges" 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" response 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.

A server that generates a multipart response SHOULD send the parts in the same order that the corresponding 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.

15.3.7.3. 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 8.8.1).

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 content consists of the union of partial content ranges within the new response and all of the matching stored 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 "multipart/byteranges" content, or multiple 206 (Partial Content) responses, each with one continuous range that is indicated by a Content-Range header field.

15.4. 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. There are several types of redirects:

1. Redirects that indicate this resource might be available at a different URI, as provided by the Location header field, as in the status codes 301 (Moved Permanently), 302 (Found), 307 (Temporary Redirect), and 308 (Permanent Redirect).
2. Redirection that offers a choice among matching resources capable of representing this resource, as in the 300 (Multiple Choices) status code.
3. Redirection to a different resource, identified by the Location header field, that can represent an indirect response to the request, as in the 303 (See Other) status code.
4. Redirection to a previously stored result, as in the 304 (Not Modified) status code.

   Note: In HTTP/1.0, the status codes 301 (Moved Permanently) and 302 (Found) were originally defined as method-preserving ([HTTP/1.0], Section 9.3) to match their implementation at CERN; 303 (See Other) was defined for a redirection that changed its method to GET. However, early user agents split on whether to redirect POST requests as POST (according to then-current specification) or as GET (the safer alternative when redirected to a different site). Prevailing practice eventually converged on changing the method to GET. 307 (Temporary Redirect) and 308 (Permanent Redirect) [RFC7538] were later added to unambiguously indicate method-preserving redirects, and status codes 301 and 302 have been adjusted to allow a POST request to be redirected as GET.

If a Location header field (Section 10.2.2) 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 be done with care for methods not known to be safe, as defined in Section 9.2.1, since the user might not wish to redirect an unsafe request.

When automatically following a redirected request, the user agent SHOULD resend the original request message with the following modifications:

1. Replace the target URI with the URI referenced by the redirection response's Location header field value after resolving it relative to the original request's target URI.
2. Remove header fields that were automatically generated by the implementation, replacing them with updated values as appropriate to the new request. This includes:
3. Consider removing header fields that were not automatically generated by the implementation (i.e., those present in the request because they were added by the calling context) where there are security implications; this includes but is not limited to Authorization and Cookie.
4. Change the request method according to the redirecting status code's semantics, if applicable.
5. If the request method has been changed to GET or HEAD, remove content-specific header fields, including (but not limited to) Content-Encoding, Content-Language, Content-Location, Content-Type, Content-Length, Digest, Last-Modified.

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.

15.4.1. 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 12).

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 content 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 content. 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 heuristically cacheable; 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 as a Link header field value [RFC8288] whose members have a relationship of "alternate", though deployment is a chicken-and-egg problem.

15.4.2. 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. The server is suggesting that a user agent with link-editing capability can permanently replace references to the target URI with one of the new references sent by the server. However, this suggestion is usually ignored unless the user agent is actively editing references (e.g., engaged in authoring content), the connection is secured, and the origin server is a trusted authority for the content being edited.

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 content 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 308 (Permanent Redirect) status code can be used instead.

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

15.4.3. 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 target 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 content 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.

15.4.4. 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 target 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 different resource, since doing so provides the information corresponding to the POST response as a resource that can be separately identified, bookmarked, and cached.

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.

15.4.5. 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 content 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:

- Content-Location, Date, ETag, and Vary
- Cache-Control and Expires (see [CACHING])

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 is terminated by the end of the header section; it cannot contain content or trailers.

15.4.6. 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]).

15.4.7. 306 (Unused)

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

15.4.8. 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 target 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 content usually contains a short hypertext note with a hyperlink to the different URI(s).

15.4.9. 308 Permanent Redirect

The 308 (Permanent Redirect) 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. The server is suggesting that a user agent with link-editing capability can permanently replace references to the target URI with one of the new references sent by the server. However, this suggestion is usually ignored unless the user agent is actively editing references (e.g., engaged in authoring content), the connection is secured, and the origin server is a trusted authority for the content being edited.

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 content usually contains a short hypertext note with a hyperlink to the new URI(s).

A 308 response is heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

> **Note:** This status code is much younger (June 2014) than its sibling codes and thus might not be recognized everywhere. See Section 4 of [RFC7538] for deployment considerations.

15.5. 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.

15.5.1. 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).

15.5.2. 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 11.6.1) 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 11.6.2). If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user agent SHOULD present the enclosed representation to the user, since it usually contains relevant diagnostic information.

15.5.3. 402 Payment Required

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

15.5.4. 403 Forbidden

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

15.5.5. 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 heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

15.5.6. 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 heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

15.5.7. 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 12.1), and the server is unwilling to supply a default representation.

The server SHOULD generate content 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 15.4.1.

15.5.8. 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 for this request. The proxy MUST send a Proxy-Authenticate header field (Section 11.7.1) containing a challenge applicable to that proxy for the request. The client MAY repeat the request with a new or replaced Proxy-Authorization header field (Section 11.7.2).

15.5.9. 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.

If the client has an outstanding request in transit, it MAY repeat that request. If the current connection is not usable (e.g., as it would be in HTTP/1.1 because request delimitation is lost), a new connection will be used.

15.5.10. 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 content 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.

15.5.11. 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 heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

15.5.12. 411 Length Required

The 411 (Length Required) status code indicates that the server refuses to accept the request without a defined Content-Length (Section 8.6). The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the request content.

15.5.13. 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 (Section 13). This response status 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.

15.5.14. 413 Content Too Large

The 413 (Content Too Large) status code indicates that the server is refusing to process a request because the request content is larger than the server is willing or able to process. The server MAY terminate the request, if the protocol version in use allows it; otherwise, the server MAY close the connection.

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.

15.5.15. 414 URI Too Long

The 414 (URI Too Long) status code indicates that the server is refusing to service the request because the target URI 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 an infinite loop 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 heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

15.5.16. 415 Unsupported Media Type

The 415 (Unsupported Media Type) status code indicates that the origin server is refusing to service the request because the content 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.

If the problem was caused by an unsupported content coding, the Accept-Encoding response header field (Section 12.5.3) ought to be used to indicate which (if any) content codings would have been accepted in the request.

On the other hand, if the cause was an unsupported media type, the Accept response header field (Section 12.5.1) can be used to indicate which media types would have been accepted in the request.

15.5.17. 416 Range Not Satisfiable

The 416 (Range Not Satisfiable) status code indicates that the set of ranges in the request's Range header field (Section 14.2) has been rejected either because none of the requested ranges are satisfiable or because the client has requested an excessive number of small or overlapping ranges (a potential denial of service attack).

Each range unit defines what is required for its own range sets to be satisfiable. For example, Section 14.1.2 defines what makes a bytes range set satisfiable.

A server that generates a 416 response to a byte-range request SHOULD generate a Content-Range header field specifying the current length of the selected representation (Section 14.4).

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 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 range 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.

15.5.18. 417 Expectation Failed

The 417 (Expectation Failed) status code indicates that the expectation given in the request's Expect header field (Section 10.1.1) could not be met by at least one of the inbound servers.

15.5.19. 418 (Unused)

[RFC2324] was an April 1 RFC that lampooned the various ways HTTP was abused; one such abuse was the definition of an application-specific 418 status code, which has been deployed as a joke often enough for the code to be unusable for any future use.

Therefore, the 418 status code is reserved in the IANA HTTP Status Code Registry. This indicates that the status code cannot be assigned to other applications currently. If future circumstances require its use (e.g., exhaustion of 4NN status codes), it can be re-assigned to another use.

15.5.20. 421 Misdirected Request

The 421 (Misdirected Request) status code indicates that the request was directed at a server that is unable or unwilling to produce an authoritative response for the target URI. An origin server (or gateway acting on behalf of the origin server) sends 421 to reject a target URI that does not match an origin for which the server has been configured (Section 4.3.1) or does not match the connection context over which the request was received (Section 7.4).

A client that receives a 421 (Misdirected Request) response MAY retry the request, whether or not the request method is idempotent, over a different connection, such as a fresh connection specific to the target resource's origin, or via an alternative service [ALTSVC].

A proxy MUST NOT generate a 421 response.

15.5.21. 422 Unprocessable Content

The 422 (Unprocessable Content) status code indicates that the server understands the content type of the request content (hence a 415 (Unsupported Media Type) status code is inappropriate), and the syntax of the request content is correct, but it was unable to process the contained instructions. For example, this status code can be sent if an XML request content contains well-formed (i.e., syntactically correct), but semantically erroneous XML instructions.

15.5.22. 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 7.8).

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.
```

15.6. 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 status codes are applicable to any request method.

15.6.1. 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.

15.6.2. 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 heuristically cacheable; i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [CACHING]).

15.6.3. 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.

15.6.4. 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 10.2.3) 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.

15.6.5. 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.

15.6.6. 505 HTTP Version Not Supported

The 505 (HTTP Version Not Supported) status code indicates that the server does not support, or refuses to support, the major version of HTTP that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, as described in Section

2.5, 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.

16. Extending HTTP

HTTP defines a number of generic extension points that can be used to introduce capabilities to the protocol without introducing a new version, including methods, status codes, field names, and further extensibility points within defined fields, such as authentication schemes and cache directives (see Cache-Control extensions in Section 5.2.3 of [CACHING]). Because the semantics of HTTP are not versioned, these extension points are persistent; the version of the protocol in use does not affect their semantics.

Version-independent extensions are discouraged from depending on or interacting with the specific version of the protocol in use. When this is unavoidable, careful consideration needs to be given to how the extension can interoperate across versions.

Additionally, specific versions of HTTP might have their own extensibility points, such as transfer codings in HTTP/1.1 (Section 6.1 of [HTTP/1.1]) and HTTP/2 SETTINGS or frame types ([HTTP/2]). These extension points are specific to the version of the protocol they occur within.

Version-specific extensions cannot override or modify the semantics of a version-independent mechanism or extension point (like a method or header field) without explicitly being allowed by that protocol element. For example, the CONNECT method (Section 9.3.6) allows this.

These guidelines assure that the protocol operates correctly and predictably, even when parts of the path implement different versions of HTTP.

16.1. Method Extensibility

16.1.1. 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 9)
- Safe ("yes" or "no", see Section 9.2.1)
- Idempotent ("yes" or "no", see Section 9.2.2)
- Pointer to specification text

Values to be added to this namespace require IETF Review (see [RFC8126], Section 4.8).

16.1.2. 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) 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 content on either the request or the response message. Definitions of new methods can specify that only a zero-length content is allowed by requiring a Content-Length header field with a value of "0".

Likewise, new methods cannot use the special host:port and asterisk forms of request target that are allowed for CONNECT and OPTIONS, respectively (Section 7.1). A full URI in absolute form is needed for the target URI, which means either the request target needs to be sent in absolute form or the target URI will be reconstructed from the request context in the same way it is for other methods.

A new method definition needs to indicate whether it is safe (Section 9.2.1), idempotent (Section 9.2.2), cacheable (Section 9.2.3), what semantics are to be associated with the request content (if any), 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 13.1) 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 14.2), 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].

16.2. Status Code Extensibility

16.2.1. 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:

- Status Code (3 digits)
- Short Description
- Pointer to specification text

Values to be added to the HTTP status code namespace require IETF Review (see [RFC8126], Section 4.8).

16.2.2. 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 15. To allow existing parsers to process the response message, new status codes cannot disallow content, although they can mandate a zero-length content.

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 field semantics are further refined when used with the new status code).

By default, a status code applies only to the request corresponding to the response it occurs within. If a status code applies to a larger scope of applicability — for example, all requests to the resource in question or all requests to a server — this must be explicitly specified. When doing so, it should be noted that not all clients can be expected to consistently apply a larger scope because they might not understand the new status code.

The definition of a new final status code ought to specify whether or not it is heuristically cacheable. Note that any response with a final status code can be cached if the response has explicit freshness information. A status code defined as heuristically cacheable is allowed to be cached without explicit freshness information. Likewise, the definition of a status code can place constraints upon cache behavior if the must-understand cache directive is used. See [CACHING] for more information.

Finally, the definition of a new status code ought to indicate whether the content has any implied association with an identified resource (Section 6.4.2).

16.3. Field Extensibility

HTTP's most widely used extensibility point is the definition of new header and trailer fields.

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

However, defining a field doesn't guarantee its deployment or recognition by recipients. Most fields are designed with the expectation that a recipient can safely ignore (but forward downstream) any field not recognized. In other cases, the sender's ability to understand a given field might be indicated by its prior communication, perhaps in the protocol version or fields that it sent in prior messages, or its use of a specific media type. Likewise, direct inspection of support might be possible through an OPTIONS request or by interacting with a defined well-known URI [RFC8615] if such inspection is defined along with the field being introduced.

16.3.1. Field Name Registry

The "Hypertext Transfer Protocol (HTTP) Field Name Registry" defines the namespace for HTTP field names.

Any party can request registration of an HTTP field. See Section 16.3.2 for considerations to take into account when creating a new HTTP field.

The "Hypertext Transfer Protocol (HTTP) Field Name Registry" is located at https://www.iana.org/assignments/http-fields/. Registration requests can be made by following the instructions located there or by sending an email to the "ietf-http-wg@w3.org" mailing list.

Field names are registered on the advice of a designated expert (appointed by the IESG or their delegate). Fields with the status 'permanent' are Specification Required ([RFC8126], Section 4.6).

Registration requests consist of the following information:

Field name:	The requested field name. It MUST conform to the field-name syntax defined in Section 5.1, and it SHOULD be restricted to just letters, digits, and hyphen ('-') characters, with the first character being a letter.
Status:	"permanent", "provisional", "deprecated", or "obsoleted".
Specification document(s):	Reference to the document that specifies the field, preferably including a URI that can be used to retrieve a copy of the document. Optional but encouraged for provisional registrations. An indication of the relevant section(s) can also be included, but is not required.

And optionally:

Comments: Additional information, such as about reserved entries.

The expert(s) can define additional fields to be collected in the registry, in consultation with the community.

Standards-defined names have a status of "permanent". Other names can also be registered as permanent if the expert(s) finds that they are in use, in consultation with the community. Other names should be registered as "provisional".

Provisional entries can be removed by the expert(s) if — in consultation with the community — the expert(s) find that they are not in use. The expert(s) can change a provisional entry's status to permanent at any time.

Note that names can be registered by third parties (including the expert(s)) if the expert(s) determines that an unregistered name is widely deployed and not likely to be registered in a timely manner otherwise.

16.3.2. Considerations for New Fields

HTTP header and trailer fields are a widely used extension point for the protocol. While they can be used in an ad hoc fashion, fields that are intended for wider use need to be carefully documented to ensure interoperability.

In particular, authors of specifications defining new fields are advised to consider and, where appropriate, document the following aspects:

- Under what conditions the 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 semantics are further refined by their context, such as their use with certain request methods or status codes.
- The scope of applicability for the information conveyed. By default, fields apply only to the message they are associated with, but some response fields are designed to apply to all representations of a resource, the resource itself, or an even broader scope. Specifications that expand the scope of a response field will need to carefully consider issues such as content negotiation, the time period of applicability, and (in some cases) multi-tenant server deployments.
- Under what conditions intermediaries are allowed to insert, delete, or modify the field's value.
- If the field is allowable in trailers; by default, it will not be (see Section 6.5.1).
- Whether it is appropriate or even required to list the field name in the Connection header field (i.e., if the field is to be hop-by-hop; see Section 7.6.1).
- Whether the field introduces any additional security considerations, such as disclosure of privacy-related data.

Request header fields have additional considerations that need to be documented if the default behavior is not appropriate:

- If 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 12.5.5).
- If the field is intended to be stored when received in a PUT request (see Section 9.3.4).
- If the field ought to be removed when automatically redirecting a request due to security concerns (see Section 15.4).

16.3.2.1. Considerations for New Field Names

Authors of specifications defining new fields are advised to choose a short but descriptive field name. Short names avoid needless data transmission; descriptive names avoid confusion and "squatting" on names that might have broader uses.

To that end, limited-use fields (such as a header confined to a single application or use case) are encouraged to use a name that includes that use (or an abbreviation) as a prefix; for example, if the Foo Application needs a Description field, it might use "Foo-Desc"; "Description" is too generic, and "Foo-Description" is needlessly long.

While the field-name syntax is defined to allow any token character, in practice some implementations place limits on the characters they accept in field-names. To be interoperable, new field names SHOULD constrain themselves to alphanumeric characters, "-", and ".", and SHOULD begin with a letter. For example, the underscore ("_") character can be problematic when passed through non-HTTP gateway interfaces (see Section 17.10).

Field names ought not be prefixed with "X-"; see [BCP178] for further information.

Other prefixes are sometimes used in HTTP field names; for example, "Accept-" is used in many content negotiation headers, and "Content-" is used as explained in Section 6.4. These prefixes are only an aid to recognizing the purpose of a field and do not trigger automatic processing.

16.3.2.2. Considerations for New Field Values

A major task in the definition of a new HTTP field is the specification of the field value syntax: what senders should generate, and how recipients should infer semantics from what is received.

Authors are encouraged (but not required) to use either the ABNF rules in this specification or those in [RFC8941] to define the syntax of new field values.

Authors are advised to carefully consider how the combination of multiple field lines will impact them (see Section 5.3). Because senders might erroneously send multiple values, and both intermediaries and HTTP libraries can perform combination automatically, this applies to all field values — even when only a single value is anticipated.

Therefore, authors are advised to delimit or encode values that contain commas (e.g., with the quoted-string rule of Section 5.6.4, the String data type of [RFC8941], or a field-specific encoding). This ensures that commas within field data are not confused with the commas that delimit a list value.

For example, the Content-Type field value only allows commas inside quoted strings, which can be reliably parsed even when multiple values are present. The Location field value provides a counter-example that should not be emulated: because URIs can include commas, it is not possible to reliably distinguish between a single value that includes a comma from two values.

Authors of fields with a singleton value (see Section 5.5) are additionally advised to document how to treat messages where the multiple members are present (a sensible default would be to ignore the field, but this might not always be the right choice).

16.4. Authentication Scheme Extensibility

16.4.1. 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 [RFC8126], Section 4.8).

16.4.2. 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 3.3).
- The authentication parameter "realm" is reserved for defining protection spaces as described in Section 11.5. 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 response directive (Section 5.2.2.7 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 cache response directives (e.g., "private").

- Schemes using Authentication-Info, Proxy-Authentication-Info, or any other authentication related response header field need to consider and document the related security considerations (see Section 17.16.4).

16.5. Range Unit Extensibility

16.5.1. 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 [RFC8126], Section 4.8).

16.5.2. Considerations for New Range Units

Other range units, such as format-specific boundaries like pages, sections, records, rows, or time, are potentially usable in HTTP for application-specific purposes, but are not commonly used in practice. Implementors of alternative range units ought to consider how they would work with content codings and general-purpose intermediaries.

16.6. Content Coding Extensibility

16.6.1. 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 (per the "HTTP Transfer Coding Registry" located at https://www.iana.org/assignments/http-parameters/) unless the encoding transformation is identical (as is the case for the compression codings defined in Section 8.4.1).

Values to be added to this namespace require IETF Review (see Section 4.8 of [RFC8126]) and MUST conform to the purpose of content coding defined in Section 8.4.1.

16.6.2. Considerations for New Content Codings

New content codings ought to be self-descriptive whenever possible, with optional parameters discoverable within the coding format itself, rather than rely on external metadata that might be lost during transit.

16.7. Upgrade Token Registry

The "Hypertext Transfer Protocol (HTTP) Upgrade Token Registry" defines the namespace for protocol-name tokens used to identify protocols in the Upgrade header field. The registry is maintained at https://www.iana.org/assignments/http-upgrade-tokens.

Each registered protocol name is associated with contact information and an optional set of specifications that details how the connection will be processed after it has been upgraded.

Registrations happen on a "First Come First Served" basis (see Section 4.4 of [RFC8126]) and are subject to the following rules:

1. A protocol-name token, once registered, stays registered forever.
2. A protocol-name token is case-insensitive and registered with the preferred case to be generated by senders.
3. The registration MUST name a responsible party for the registration.
4. The registration MUST name a point of contact.
5. The registration MAY name a set of specifications associated with that token. Such specifications need not be publicly available.
6. The registration SHOULD name a set of expected "protocol-version" tokens associated with that token at the time of registration.
7. The responsible party MAY change the registration at any time. The IANA will keep a record of all such changes, and make them available upon request.
8. The IESG MAY reassign responsibility for a protocol token. This will normally only be used in the case when a responsible party cannot be contacted.

17. 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 caching are discussed in Section 7 of [CACHING], and considerations related to HTTP/1.1 message syntax and parsing are discussed in Section 11 of [HTTP/1.1].

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 content received via HTTP, or secure use of the Internet in general, rather than security of the protocol. The security considerations for URIs, which are fundamental to HTTP operation, are discussed in Section 7 of [URI]. Various organizations maintain topical information and links to current research on Web application security (e.g., [OWASP]).

17.1. Establishing Authority

HTTP relies on the notion of an *authoritative response*: a response that has been determined by (or at the direction of) the origin server 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.

When a registered name is used in the authority component, the "http" URI scheme (Section 4.2.1) 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 for "http" URIs. 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, as are the various mechanisms for making DNS requests over more secure transfer protocols.

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 4.2.2) is intended to prevent (or at least reveal) many of these potential attacks on establishing authority, provided that the negotiated connection is secured and the client properly verifies that the communicating server's identity matches the target URI's authority component (Section 4.3.4). Correctly implementing such verification can be difficult (see [Georgiev]).

Authority for a given origin server can be delegated through protocol extensions; for example, [ALTSVC]. Likewise, the set of servers for which a connection is considered authoritative can be changed with a protocol extension like [RFC8336].

Providing a response from a non-authoritative source, such as a shared proxy 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, communicating authority to users 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 4.2.1). 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.

17.2. Risks of Intermediaries

HTTP intermediaries are inherently situated for on-path 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).

Intermediaries are no more trustworthy than the people and policies under which they operate; HTTP cannot solve this problem.

17.3. Attacks Based on File and Path Names

Origin servers frequently make use of their local file system to manage the mapping from target 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 target resource 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.

17.4. 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, target URI, header fields, or content) 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 target URI 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.

17.5. 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 2.3).

To promote interoperability, specific recommendations are made for minimum size limits on fields (Section 5.4). 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 target URI that is too long (Section 15.5.15) or request content that is too large (Section 15.5.14). 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, field names, numeric values, and chunk lengths. Failure to limit such processing can result in arbitrary code execution due to buffer or arithmetic overflows, and increased vulnerability to denial-of-service attacks.

17.6. Attacks Using Shared-Dictionary Compression

Some attacks on encrypted protocols use the differences in size created by dynamic compression to reveal confidential information; for example, [BREACH]. These attacks rely on creating a redundancy between attacker-controlled content and the confidential information, such that a dynamic compression algorithm using the same dictionary for both content will compress more efficiently when the attacker-controlled content matches parts of the confidential content.

HTTP messages can be compressed in a number of ways, including using TLS compression, content codings, transfer codings, and other extension or version-specific mechanisms.

The most effective mitigation for this risk is to disable compression on sensitive data, or to strictly separate sensitive data from attacker-controlled data so that they cannot share the same compression dictionary. With careful design, a compression scheme can be designed in a way that is not considered exploitable in limited use cases, such as HPACK ([HPACK]).

17.7. 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.

17.8. Privacy of Server Log Information

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

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

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

17.9. 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. Many servers, proxies, and user agents log or display the target URI in places where it might be visible to third parties. It is therefore unwise to include information within a URI that is sensitive, personally identifiable, or a risk to disclose.

When an application uses client-side mechanisms to construct a target URI out of user-provided information, such as the query fields of a form using GET, potentially sensitive data might be provided that would not

be appropriate for disclosure within a URI. POST is often preferred in such cases because it usually doesn't construct a URI; instead, POST of a form transmits the potentially sensitive data in the request content. However, this hinders caching and uses an unsafe method for what would otherwise be a safe request. Alternative workarounds include transforming the user-provided data prior to constructing the URI or filtering the data to only include common values that are not sensitive. Likewise, redirecting the result of a query to a different (server-generated) URI can remove potentially sensitive data from later links and provide a cacheable response for later reuse.

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 10.1.3 to address some of its security considerations.

17.10. Application Handling of Field Names

Servers often use non-HTTP gateway interfaces and frameworks to process a received request and produce content for the response. For historical reasons, such interfaces often pass received field names as external variable names, using a name mapping suitable for environment variables.

For example, the Common Gateway Interface (CGI) mapping of protocol-specific meta-variables, defined by Section 4.1.18 of [RFC3875], is applied to received header fields that do not correspond to one of CGI's standard variables; the mapping consists of prepending "HTTP_" to each name and changing all instances of hyphen ("-") to underscore ("_"). This same mapping has been inherited by many other application frameworks in order to simplify moving applications from one platform to the next.

In CGI, a received Content-Length field would be passed as the meta-variable "CONTENT_LENGTH" with a string value matching the received field's value. In contrast, a received "Content_Length" header field would be passed as the protocol-specific meta-variable "HTTP_CONTENT_LENGTH", which might lead to some confusion if an application mistakenly reads the protocol-specific meta-variable instead of the default one. (This historical practice is why Section 16.3.2.1 discourages the creation of new field names that contain an underscore.)

Unfortunately, mapping field names to different interface names can lead to security vulnerabilities if the mapping is incomplete or ambiguous. For example, if an attacker were to send a field named "Transfer_Encoding", a naive interface might map that to the same variable name as the "Transfer-Encoding" field, resulting in a potential request smuggling vulnerability (Section 11.2 of [HTTP/1.1]).

To mitigate the associated risks, implementations that perform such mappings are advised to make the mapping unambiguous and complete for the full range of potential octets received as a name (including those that are discouraged or forbidden by the HTTP grammar). For example, a field with an unusual name character might result in the request being blocked, the specific field being removed, or the name being passed with a different prefix to distinguish it from other fields.

17.11. 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 10.2.2), 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.

17.12. Disclosure of Product Information

The User-Agent (Section 10.1.5), Via (Section 7.6.3), and Server (Section 10.2.4) 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.

17.13. 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 how it uses the underlying transport protocol, 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 ([Bujlow]) 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 12.1), 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 explicitly permitted, 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.

17.14. Validator Retention

The validators defined by this specification are not intended to ensure the validity of a representation, guard against malicious changes, or detect on-path 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.

17.15. 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.

17.16. 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.

17.16.1. 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 fields. Services that depend on individual user authentication require a secured connection prior to exchanging credentials (Section 4.2.2).

17.16.2. 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.

17.16.3. 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 origin (Section 11.5). 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.

17.16.4. Additional Response Fields

Adding information to responses that are sent over an unencrypted channel can affect security and privacy. The presence of the Authentication-Info and Proxy-Authentication-Info header fields alone indicates that HTTP authentication is in use. Additional information could be exposed by the contents of the authentication-scheme specific parameters; this will have to be considered in the definitions of these schemes.

18. IANA Considerations

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

18.1. URI Scheme Registration

IANA has updated the "Uniform Resource Identifier (URI) Schemes" registry [BCP35] at https://www.iana.org/assignments/uri-schemes/ with the permanent schemes listed in Table 2 in Section 4.2.

18.2. Method Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Method Registry" at https://www.iana.org/assignments/http-methods with the registration procedure of Section 16.1.1 and the method names summarized in the following table.

Method	Safe	Idempotent	Section
CONNECT	no	no	9.3.6
DELETE	no	yes	9.3.5
GET	yes	yes	9.3.1
HEAD	yes	yes	9.3.2
OPTIONS	yes	yes	9.3.7
POST	no	no	9.3.3
PUT	no	yes	9.3.4
TRACE	yes	yes	9.3.8
*	no	no	18.2

Table 7

The method name "*" is reserved because using "*" as a method name would conflict with its usage as a wildcard in some fields (e.g., "Access-Control-Request-Method").

18.3. Status Code Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Status Code Registry" at https://www.iana.org/assignments/http-status-codes with the registration procedure of Section 16.2.1 and the status code values summarized in the following table.

Value	Description	Section
100	Continue	15.2.1
101	Switching Protocols	15.2.2
200	OK	15.3.1
201	Created	15.3.2
202	Accepted	15.3.3
203	Non-Authoritative Information	15.3.4
204	No Content	15.3.5
205	Reset Content	15.3.6
206	Partial Content	15.3.7
300	Multiple Choices	15.4.1
301	Moved Permanently	15.4.2
302	Found	15.4.3
303	See Other	15.4.4
304	Not Modified	15.4.5
305	Use Proxy	15.4.6
306	(Unused)	15.4.7
307	Temporary Redirect	15.4.8

Value	Description	Section
308	Permanent Redirect	15.4.9
400	Bad Request	15.5.1
401	Unauthorized	15.5.2
402	Payment Required	15.5.3
403	Forbidden	15.5.4
404	Not Found	15.5.5
405	Method Not Allowed	15.5.6
406	Not Acceptable	15.5.7
407	Proxy Authentication Required	15.5.8
408	Request Timeout	15.5.9
409	Conflict	15.5.10
410	Gone	15.5.11
411	Length Required	15.5.12
412	Precondition Failed	15.5.13
413	Content Too Large	15.5.14
414	URI Too Long	15.5.15
415	Unsupported Media Type	15.5.16
416	Range Not Satisfiable	15.5.17
417	Expectation Failed	15.5.18
418	(Unused)	15.5.19
421	Misdirected Request	15.5.20
422	Unprocessable Content	15.5.21
426	Upgrade Required	15.5.22
500	Internal Server Error	15.6.1
501	Not Implemented	15.6.2
502	Bad Gateway	15.6.3
503	Service Unavailable	15.6.4
504	Gateway Timeout	15.6.5
505	HTTP Version Not Supported	15.6.6

Table 8

18.4. Field Name Registration

This specification updates the HTTP-related aspects of the existing registration procedures for message header fields defined in [RFC3864]. It replaces the old procedures as they relate to HTTP by defining a new registration procedure and moving HTTP field definitions into a separate registry.

IANA has created a new registry titled "Hypertext Transfer Protocol (HTTP) Field Name Registry" as outlined in Section 16.3.1.

IANA has moved all entries in the "Permanent Message Header Field Names" and "Provisional Message Header Field Names" registries (see https://www.iana.org/assignments/message-headers/) with the protocol 'http' to this registry and has applied the following changes:

1. The 'Applicable Protocol' field has been omitted.
2. Entries that had a status of 'standard', 'experimental', 'reserved', or 'informational' have been made to have a status of 'permanent'.
3. Provisional entries without a status have been made to have a status of 'provisional'.
4. Permanent entries without a status (after confirmation that the registration document did not define one) have been made to have a status of 'provisional'. The expert(s) can choose to update the entries' status if there is evidence that another is more appropriate.

IANA has annotated the "Permanent Message Header Field Names" and "Provisional Message Header Field Names" registries with the following note to indicate that HTTP field name registrations have moved:

Note

HTTP field name registrations have been moved to [https://www.iana.org/assignments/http-fields] per [RFC9110].

IANA has updated the "Hypertext Transfer Protocol (HTTP) Field Name Registry" with the field names listed in the following table.

Field Name	Status	Section	Comments
Accept	permanent	12.5.1	
Accept-Charset	deprecated	12.5.2	
Accept-Encoding	permanent	12.5.3	
Accept-Language	permanent	12.5.4	
Accept-Ranges	permanent	14.3	
Allow	permanent	10.2.1	
Authentication-Info	permanent	11.6.3	
Authorization	permanent	11.6.2	
Connection	permanent	7.6.1	
Content-Encoding	permanent	8.4	
Content-Language	permanent	8.5	
Content-Length	permanent	8.6	
Content-Location	permanent	8.7	
Content-Range	permanent	14.4	
Content-Type	permanent	8.3	
Date	permanent	6.6.1	
ETag	permanent	8.8.3	
Expect	permanent	10.1.1	
From	permanent	10.1.2	
Host	permanent	7.2	
If-Match	permanent	13.1.1	
If-Modified-Since	permanent	13.1.3	
If-None-Match	permanent	13.1.2	
If-Range	permanent	13.1.5	
If-Unmodified-Since	permanent	13.1.4	
Last-Modified	permanent	8.8.2	
Location	permanent	10.2.2	
Max-Forwards	permanent	7.6.2	
Proxy-Authenticate	permanent	11.7.1	
Proxy-Authentication-Info	permanent	11.7.3	
Proxy-Authorization	permanent	11.7.2	
Range	permanent	14.2	
Referer	permanent	10.1.3	
Retry-After	permanent	10.2.3	
Server	permanent	10.2.4	
TE	permanent	10.1.4	
Trailer	permanent	6.6.2	
Upgrade	permanent	7.8	
User-Agent	permanent	10.1.5	
Vary	permanent	12.5.5	
Via	permanent	7.6.3	
WWW-Authenticate	permanent	11.6.1	
*	permanent	12.5.5	(reserved)

Table 9

The field name "*" is reserved because using that name as an HTTP header field might conflict with its special semantics in the Vary header field (Section 12.5.5).

IANA has updated the "Content-MD5" entry in the new registry to have a status of 'obsoleted' with references to Section 14.15 of [RFC2616] (for the definition of the header field) and Appendix B of [RFC7231] (which removed the field definition from the updated specification).

18.5. Authentication Scheme Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" at https://www.iana.org/assignments/http-authschemes with the registration procedure of Section 16.4.1. No authentication schemes are defined in this document.

18.6. Content Coding Registration

IANA has updated the "HTTP Content Coding Registry" at https://www.iana.org/assignments/http-parameters/ with the registration procedure of Section 16.6.1 and the content coding names summarized in the table below.

Name	Description	Section
compress	UNIX "compress" data format [Welch]	8.4.1.1
deflate	"deflate" compressed data ([RFC1951]) inside the "zlib" data format ([RFC1950])	8.4.1.2
gzip	GZIP file format [RFC1952]	8.4.1.3
identity	Reserved	12.5.3
x-compress	Deprecated (alias for compress)	8.4.1.1
x-gzip	Deprecated (alias for gzip)	8.4.1.3

Table 10

18.7. Range Unit Registration

IANA has updated the "HTTP Range Unit Registry" at https://www.iana.org/assignments/http-parameters/ with the registration procedure of Section 16.5.1 and the range unit names summarized in the table below.

Range Unit Name	Description	Section
bytes	a range of octets	14.1.2
none	reserved as keyword to indicate range requests are not supported	14.3

Table 11

18.8. Media Type Registration

IANA has updated the "Media Types" registry at https://www.iana.org/assignments/media-types with the registration information in Section 14.6 for the media type "multipart/byteranges".

IANA has updated the registry note about "q" parameters with a link to Section 12.5.1 of this document.

18.9. Port Registration

IANA has updated the "Service Name and Transport Protocol Port Number Registry" at https://www.iana.org/assignments/service-names-port-numbers/ for the services on ports 80 and 443 that use UDP or TCP to:

1. use this document as "Reference", and
2. when currently unspecified, set "Assignee" to "IESG" and "Contact" to "IETF_Chair".

18.10. Upgrade Token Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Upgrade Token Registry" at https://www.iana.org/assignments/http-upgrade-tokens with the registration procedure described in Section 16.7 and the upgrade token names summarized in the following table.

Name	Description	Expected Version Tokens	Section
HTTP	Hypertext Transfer Protocol	any DIGIT.DIGIT (e.g., "2.0")	2.5

Table 12

19. References

19.1. Normative References

[CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", RFC 9111, DOI 10.17487/RFC9111, June 2022.

[RFC1950] Deutsch, P. 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/rfc>.

[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/rfc>.

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc>.

[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/rfc>.

[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/rfc>.

[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/rfc>.

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

[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/rfc>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc>.

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

[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/rfc>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc>.

[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/rfc>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc>.

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

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc>.

[URI] 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/rfc>.

[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), DOI 10.1109/MC.1984.1659158, June 1984, <https://ieeexplore.ieee.org/document/1659158/>.

19.2. Informative References

[ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc>.

[BCP13] Freed, N. and J. Klensin, "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures", BCP 13, RFC 4289, DOI 10.17487/RFC4289, December 2005.
Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, 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, DOI 10.17487/RFC6648, June 2012.
https://www.rfc-editor.org/info/bcp178>

[BCP35] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015.
https://www.rfc-editor.org/info/bcp35>

[BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>.

[Bujlow] Bujlow, T., Carela-Español, V., Solé-Pareta, J., and P. Barlet-Ros, "A Survey on Web Tracking: Mechanisms, Implications, and Defenses", DOI 10.1109/JPROC.2016.2637878, In Proceedings of the IEEE 105(8), August 2017.

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

[Err1912] RFC Errata, "Erratum ID 1912", RFC 2978, <https://www.rfc-editor.org/errata/eid1912>.

[Err5433] RFC Errata, "Erratum ID 5433", RFC 2978, <https://www.rfc-editor.org/errata/eid5433>.

[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", DOI 10.1145/2382196.2382204, In Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12), pp. 38-49, October 2012.

[HPACK] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc>.

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

[HTTP/1.1] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", RFC 9112, DOI 10.17487/RFC9112, June 2022.

[HTTP/2] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/info/rfc>.

[HTTP/3] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/info/rfc>.

[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] "The Open Web Application Security Project", <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/rfc>.

[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/rfc>.

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Frystyk, 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/rfc>.

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

[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/rfc>.

[RFC2324] Masinter, L., "Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)", RFC 2324, DOI 10.17487/RFC2324, 1 April 1998, <https://www.rfc-editor.org/info/rfc>.

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

[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/rfc>.

[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/rfc>.

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

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

[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/rfc>.

[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/rfc>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc>.

[RFC3875] Robinson, D. and K. Coar, "The Common Gateway Interface (CGI) Version 1.1", RFC 3875, DOI 10.17487/RFC3875, October 2004, <https://www.rfc-editor.org/info/rfc>.

[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/rfc>.

[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/rfc>.

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

[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/rfc>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc>.

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

[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/rfc>.

[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/rfc>.

[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/rfc>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <https://www.rfc-editor.org/info/rfc>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc>.

[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/rfc>.

[RFC7538] Reschke, J., "The Hypertext Transfer Protocol Status Code 308 (Permanent Redirect)", RFC 7538, DOI 10.17487/RFC7538, April 2015, <https://www.rfc-editor.org/info/rfc>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc>.

[RFC7578] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 7578, DOI 10.17487/RFC7578, July 2015, <https://www.rfc-editor.org/info/rfc>.

[RFC7615] Reschke, J., "HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields", RFC 7615, DOI 10.17487/RFC7615, September 2015, <https://www.rfc-editor.org/info/rfc>.

[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/rfc>.

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

[RFC7694] Reschke, J., "Hypertext Transfer Protocol (HTTP) Client-Initiated Content-Encoding", RFC 7694, DOI 10.17487/RFC7694, November 2015, <https://www.rfc-editor.org/info/rfc>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc>.

[RFC8187] Reschke, J., "Indicating Character Encoding and Language for HTTP Header Field Parameters", RFC 8187, DOI 10.17487/RFC8187, September 2017, <https://www.rfc-editor.org/info/rfc>.

[RFC8246] McManus, P., "HTTP Immutable Responses", RFC 8246, DOI 10.17487/RFC8246, September 2017, <https://www.rfc-editor.org/info/rfc>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc>.

[RFC8336] Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", RFC 8336, DOI 10.17487/RFC8336, March 2018, <https://www.rfc-editor.org/info/rfc>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/info/rfc>.

[RFC8941] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/info/rfc>.

[Sniffing] WHATWG, "MIME Sniffing", <https://mimesniff.spec.whatwg.org>.

[WEBDAV] 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/rfc>.

Appendix A. Collected ABNF

In the collected ABNF below, list rules are expanded per Section 5.6.1.

```
Accept = [ ( media-range [ weight ] ) *( OWS "," OWS ( media-range [
 weight ] ) ) ]
Accept-Charset = [ ( ( token / "*" ) [ weight ] ) *( OWS "," OWS ( (
 token / "*" ) [ weight ] ) ) ]
Accept-Encoding = [ ( codings [ weight ] ) *( OWS "," OWS ( codings [
 weight ] ) ) ]
Accept-Language = [ ( language-range [ weight ] ) *( OWS "," OWS (
 language-range [ weight ] ) ) ]
Accept-Ranges = acceptable-ranges
Allow = [ method *( OWS "," OWS method ) ]
Authentication-Info = [ auth-param *( OWS "," OWS auth-param ) ]
Authorization = credentials

BWS = OWS

Connection = [ connection-option *( OWS "," OWS connection-option )
 ]
Content-Encoding = [ content-coding *( OWS "," OWS content-coding )
 ]
Content-Language = [ language-tag *( OWS "," OWS language-tag ) ]
Content-Length = 1*DIGIT
Content-Location = absolute-URI / partial-URI
Content-Range = range-unit SP ( range-resp / unsatisfied-range )
Content-Type = media-type

Date = HTTP-date

ETag = entity-tag
Expect = [ expectation *( OWS "," OWS expectation ) ]

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 = "*" / [ entity-tag *( OWS "," OWS entity-tag ) ]
If-Modified-Since = HTTP-date
If-None-Match = "*" / [ 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 = [ challenge *( OWS "," OWS challenge ) ]
Proxy-Authentication-Info = [ auth-param *( OWS "," OWS auth-param )
 ]
Proxy-Authorization = credentials

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

Appendix B. Changes from Previous RFCs

B.1. Changes from RFC 2818

None.

B.2. Changes from RFC 7230

The sections introducing HTTP's design goals, history, architecture, conformance criteria, protocol versioning, URIs, message routing, and header fields have been moved here.

The requirement on semantic conformance has been replaced with permission to ignore or work around implementation-specific failures. (Section 2.2)

The description of an origin and authoritative access to origin servers has been extended for both "http" and "https" URIs to account for alternative services and secured connections that are not necessarily based on TCP. (Sections 4.2.1, 4.2.2, 4.3.1, and 7.3.3)

Explicit requirements have been added to check the target URI scheme's semantics and reject requests that don't meet any associated requirements. (Section 7.4)

Parameters in media type, media range, and expectation can be empty via one or more trailing semicolons. (Section 5.6.6)

"Field value" now refers to the value after multiple field lines are combined with commas — by far the most common use. To refer to a single header line's value, use "field line value". (Section 6.3)

Trailer field semantics now transcend the specifics of chunked transfer coding. The use of trailer fields has been further limited to allow generation as a trailer field only when the sender knows the field defines that usage and to allow merging into the header section only if the recipient knows the corresponding field definition permits and defines how to merge. In all other cases, implementations are encouraged either to store the trailer fields separately or to discard them instead of merging. (Section 6.5.1)

The priority of the absolute form of the request URI over the Host header field by origin servers has been made explicit to align with proxy handling. (Section 7.2)

The grammar definition for the Via field's "received-by" was expanded in RFC 7230 due to changes in the URI grammar for host [URI] that are not desirable for Via. For simplicity, we have removed uri-host from the received-by production because it can be encompassed by the existing grammar for pseudonym. In particular, this change removed comma from the allowed set of characters for a host name in received-by. (Section 7.6.3)

B.3. Changes from RFC 7231

Minimum URI lengths to be supported by implementations are now recommended. (Section 4.1)

The following have been clarified: CR and NUL in field values are to be rejected or mapped to SP, and leading and trailing whitespace needs to be stripped from field values before they are consumed. (Section 5.5)

Parameters in media type, media range, and expectation can be empty via one or more trailing semicolons. (Section 5.6.6)

An abstract data type for HTTP messages has been introduced to define the components of a message and their semantics as an abstraction across multiple HTTP versions, rather than in terms of the specific syntax form of HTTP/1.1 in [HTTP/1.1], and reflect the contents after the message is parsed. This makes it easier to distinguish between requirements on the content (what is conveyed) versus requirements on the messaging syntax (how it is conveyed) and avoids baking limitations of early protocol versions into the future of HTTP. (Section 6)

The terms "payload" and "payload body" have been replaced with "content", to better align with its usage elsewhere (e.g., in field names) and to avoid confusion with frame payloads in HTTP/2 and HTTP/3. (Section 6.4)

The term "effective request URI" has been replaced with "target URI". (Section 7.1)

Restrictions on client retries have been loosened to reflect implementation behavior. (Section 9.2.2)

The fact that request bodies on GET, HEAD, and DELETE are not interoperable has been clarified. (Sections 9.3.1, 9.3.2, and 9.3.5)

The use of the Content-Range header field (Section 14.4) as a request modifier on PUT is allowed. (Section 9.3.4)

A superfluous requirement about setting Content-Length has been removed from the description of the OPTIONS method. (Section 9.3.7)

The normative requirement to use the "message/http" media type in TRACE responses has been removed. (Section 9.3.8)

List-based grammar for Expect has been restored for compatibility with RFC 2616. (Section 10.1.1)

Accept and Accept-Encoding are allowed in response messages; the latter was introduced by [RFC7694]. (Section 12.3)

"Accept Parameters" (accept-params and accept-ext ABNF production) have been removed from the definition of the Accept field. (Section 12.5.1)

The Accept-Charset field is now deprecated. (Section 12.5.2)

The semantics of "*" in the Vary header field when other values are present was clarified. (Section 12.5.5)

Range units are compared in a case-insensitive fashion. (Section 14.1)

The use of the Accept-Ranges field is not restricted to origin servers. (Section 14.3)

The process of creating a redirected request has been clarified. (Section 15.4)

Status code 308 (previously defined in [RFC7538]) has been added so that it's defined closer to status codes 301, 302, and 307. (Section 15.4.9)

Status code 421 (previously defined in Section 9.1.2 of [RFC7540]) has been added because of its general applicability. 421 is no longer defined as heuristically cacheable since the response is specific to the connection (not the target resource). (Section 15.5.20)

Status code 422 (previously defined in Section 11.2 of [WEBDAV]) has been added because of its general applicability. (Section 15.5.21)

B.4. Changes from RFC 7232

Previous revisions of HTTP imposed an arbitrary 60-second limit on the determination of whether Last-Modified was a strong validator to guard 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. This specification has relaxed that to allow reasonable discretion. (Section 8.8.2.2)

An edge-case requirement on If-Match and If-Unmodified-Since has been removed that required a validator not to be sent in a 2xx response if validation fails because the change request has already been applied. (Sections 13.1.1 and 13.1.4)

The fact that If-Unmodified-Since does not apply to a resource without a concept of modification time has been clarified. (Section 13.1.4)

Preconditions can now be evaluated before the request content is processed rather than waiting until the response would otherwise be successful. (Section 13.2)

B.5. Changes from RFC 7233

Refactored the range-unit and ranges-specifier grammars to simplify and reduce artificial distinctions between bytes and other (extension) range units, removing the overlapping grammar of other-range-unit by defining

range units generically as a token and placing extensions within the scope of a range-spec (other-range). This disambiguates the role of list syntax (commas) in all range sets, including extension range units, for indicating a range-set of more than one range. Moving the extension grammar into range specifiers also allows protocol specific to byte ranges to be specified separately.

It is now possible to define Range handling on extension methods. (Section 14.2)

Described use of the Content-Range header field (Section 14.4) as a request modifier to perform a partial PUT. (Section 14.5)

B.6. Changes from RFC 7235

None.

B.7. Changes from RFC 7538

None.

B.8. Changes from RFC 7615

None.

B.9. Changes from RFC 7694

This specification includes the extension defined in [RFC7694] but leaves out examples and deployment considerations.

Acknowledgements

Aside from the current editors, the following individuals deserve special recognition for their contributions to early aspects of HTTP and its core specifications: Marc Andreessen, Tim Berners-Lee, Robert Cailliau, Daniel W. Connolly, Bob Denny, John Franks, Jim Gettys, Jean-François Groff, Phillip M. Hallam-Baker, Koen Holtman, Jeffery L. Hostetler, Shel Kaphan, Dave Kristol, Yves Lafon, Scott D. Lawrence, Paul J. Leach, Håkon W. Lie, Ari Luotonen, Larry Masinter, Rob McCool, Jeffrey C. Mogul, Lou Montulli, David Morris, Henrik Frystyk Nielsen, Dave Raggett, Eric Rescorla, Tony Sanders, Lawrence C. Stewart, Marc VanHeyningen, and Steve Zilles.

This document builds on the many contributions that went into past specifications of HTTP, including [HTTP/1.0], [RFC2068], [RFC2145], [RFC2616], [RFC2617], [RFC2818], [RFC7230], [RFC7231], [RFC7232], [RFC7233], [RFC7234], and [RFC7235]. The acknowledgements within those documents still apply.

Since 2014, the following contributors have helped improve this specification by reporting bugs, asking smart questions, drafting or reviewing text, and evaluating issues:

Alan Egerton, Alex Rousskov, Amichai Rothman, Amos Jeffries, Anders Kaseorg, Andreas Gebhardt, Anne van Kesteren, Armin Abfalterer, Aron Duby, Asanka Herath, Asbjørn Ulsberg, Asta Olofsson, Attila Gulyas, Austin Wright, Barry Pollard, Ben Burkert, Benjamin Kaduk, Björn Höhrmann, Brad Fitzpatrick, Chris Pacejo, Colin Bendell, Cory Benfield, Cory Nelson, Daisuke Miyakawa, Dale Worley, Daniel Stenberg, Danil Suits, David Benjamin, David Matson, David Schinazi, ##### ######## (Dilyan Palauzov), Eric Anderson, Eric Rescorla, Éric Vyncke, Erik Kline, Erwin Pe, Etan Kissling, Evert Pot, Evgeny Vrublevsky, Florian Best, Francesca Palombini, Igor Lubashev, James Callahan, James Peach, Jeffrey Yasskin, Kalin Gyokov, Kannan Goundan, # ## (Kazuho Oku), Ken Murchison, Krzysztof Maczy#ski, Lars Eggert, Lucas Pardue, Martin Duke, Martin Dürst, Martin Thomson, Martynas Jusevi#ius, Matt Menke, Matthias Pigulla, Mattias Grenfeldt, Michael Osipov, Mike Bishop, Mike Pennisi, Mike Taylor, Mike West, Mohit Sethi, Murray Kucherawy, Nathaniel J. Smith, Nicholas Hurley, Nikita Prokhorov, Patrick McManus, Piotr Sikora, Poul-Henning Kamp, Rick van Rein, Robert Wilton, Roberto Polli, Roman Danyliw, Samuel Williams, Semyon Kholodnov, Simon Pieters, Simon Schüppel, Stefan Eissing, Taylor Hunt, Todd Greer, Tommy Pauly, Vasiliy Faronov, Vladimir Lashchev, Wenbo Zhu, William A. Rowe Jr., Willy Tarreau, Xingwei Liu, Yishuai Li, and Zaheduzzaman Sarker.

Index

1

100 Continue (status code) **102**, 130
100-continue (expect value) **68**
101 Switching Protocols (status code) **102**, 130
1xx Informational (status code class) **101**

2

200 OK (status code) **102**, 130
201 Created (status code) **103**, 130
202 Accepted (status code) **103**, 130
203 Non-Authoritative Information (status code) 46, **103**, 130
204 No Content (status code) **103**, 130
205 Reset Content (status code) **104**, 130
206 Partial Content (status code) 37, 49, 96, **104**, 130
2xx Successful (status code class) **102**

3

300 Multiple Choices (status code) **107**, 111, 130
301 Moved Permanently (status code) **107**, 130
302 Found (status code) **108**, 130
303 See Other (status code) **108**, 130
304 Not Modified (status code) 52, **108**, 130
305 Use Proxy (status code) **109**, 130
306 (Unused) (status code) **109**, 130
307 Temporary Redirect (status code) **109**, 130
308 Permanent Redirect (status code) **109**, 130, 143
3xx Redirection (status code class) 47, **106**, 119, 143

4

400 Bad Request (status code) **110**, 130
401 Unauthorized (status code) **110**, 130
402 Payment Required (status code) **110**, 130
403 Forbidden (status code) 75, **110**, 130
404 Not Found (status code) **110**, 130
405 Method Not Allowed (status code) **110**, 130
406 Not Acceptable (status code) **111**, 130
407 Proxy Authentication Required (status code) **111**, 130
408 Request Timeout (status code) **111**, 130
409 Conflict (status code) **111**, 130
410 Gone (status code) **111**, 130
411 Length Required (status code) **112**, 130
412 Precondition Failed (status code) **112**, 130
413 Content Too Large (status code) **112**, 125, 130
414 URI Too Long (status code) **112**, 125, 130
415 Unsupported Media Type (status code) **112**, 130
416 Range Not Satisfiable (status code) **112**, 130
417 Expectation Failed (status code) **113**, 130
418 (Unused) (status code) **113**, 130
421 Misdirected Request (status code) 43, **113**, 130, 143
422 Unprocessable Content (status code) **113**, 130, 143
426 Upgrade Required (status code) **113**, 130
4xx Client Error (status code class) **110**

5

500 Internal Server Error (status code) **114**, 130
501 Not Implemented (status code) **114**, 130
502 Bad Gateway (status code) **114**, 130
503 Service Unavailable (status code) **114**, 130
504 Gateway Timeout (status code) **114**, 130
505 HTTP Version Not Supported (status code) **114**, 130
5xx Server Error (status code class) **114**

A

accelerator **18**
Accept header field 32, 49, 80, **81**, 112, 132, 133, 143
Accept-Charset header field **82**, 132, 143
Accept-Encoding header field 50, 58, 80, **83**, 112, 132, 133
Accept-Language header field 51, **84**, 132
Accept-Ranges header field 93, **96**, 132, 133, 143
Allow header field 60, **71**, 132
ALTSVC 24, 113, 123, **136**
Authentication-Info header field **76**, 132
authoritative response **123**
Authorization header field **76**, 77, 85, 110, 132

B

BREACH 125, **136**
browser **17**
Bujlow 127, **136**

C

cache **19**
cacheable **19**
CACHING 11, 14, 19, 24, 25, 33, 42, 44, 46, 55, 57, 61, 62, 62, 62, 63, 63, 64, 65, 76, 85, 85, 86, 88, 89, 101, 103, 103, 104, 104, 107, 108, 109, 109, 109, 110, 111, 111, 112, 114, 116, 117, 121, 123, 124, **135**
Section 3.5 76
Section 4 63
Section 4.1 85
Section 4.2 33
Section 4.2.1 63
Section 4.2.2 103, 103, 104, 104, 107, 108, 109, 110, 111, 111, 112, 114
Section 4.3.2 88, 89
Section 4.3.4 109
Section 4.3.5 62
Section 4.4 64, 65
Section 5.2 44, 62, 62, 85
Section 5.2.2.6 46
Section 5.2.2.7 121
Section 5.2.3 116
Section 7 123, 124
client **16**
clock **32**
complete **35**
compress (Coding Format) 50
compress (content coding) **50**
conditional request **86**
CONNECT method 17, 37, 41, 52, 59, **65**, 116, 130
connection **16**
Connection header field 27, 43, **43**, 47, 71, 119, 132
content 36
content coding **50**
content negotiation 12
Content-Encoding header field **49**, 50, 132
Content-Language header field **51**, 132

Content-Length header field **52**, 112, 132
Content-Location header field **53**, 63, 72, 132
Content-MD5 header field **133**
Content-Range header field 64, 93, **96**, 98, 98, 112, 132, 143, 144
Content-Type header field 29, **48**, 49, 132
control data **35**
COOKIE 22, 28, 67, 75, **136**

D

Date header field 27, 37, **39**, 56, 132
deflate (Coding Format) 50
deflate (content coding) **50**
DELETE method 59, **64**, 130, 143
Delimiters 30
downstream **18**

E

effective request URI **41**
Err1912 49, **136**
Err5433 49, **136**
ETag field 54, **56**, 132
Expect header field 47, **68**, 102, 113, 132, 143

F

field **27**, 36
field line 27
field line value 27
field name 27
field value 27
Fields
* **133**
Accept 32, 49, 80, **81**, 112, 132, 133, 143
Accept-Charset **82**, 132, 143
Accept-Encoding 50, 58, 80, **83**, 112, 132, 133
Accept-Language 51, **84**, 132
Accept-Ranges 93, **96**, 132, 133, 143
Allow 60, **71**, 132
Authentication-Info **76**, 132
Authorization **76**, 77, 85, 110, 132
Connection 27, 43, **43**, 47, 71, 119, 132
Content-Encoding **49**, 50, 132
Content-Language **51**, 132
Content-Length **52**, 112, 132
Content-Location **53**, 63, 72, 132
Content-MD5 **133**
Content-Range 64, 93, **96**, 98, 98, 112, 132, 143, 144
Content-Type 29, **48**, 49, 132
Date 27, 37, **39**, 56, 132
ETag 54, **56**, 132
Expect 47, **68**, 102, 113, 132, 143
From **69**, 132
Host 25, 41, **41**, 132, 142
If-Match **86**, 132, 143
If-Modified-Since **88**, 132
If-None-Match **87**, 132
If-Range 87, 90, **90**, 96, 132
If-Unmodified-Since **89**, 132, 143, 143
Last-Modified 54, **55**, 132
Location 63, **72**, 107, 126, 132
Max-Forwards **44**, 66, 67, 132
Proxy-Authenticate **77**, 111, 132
Proxy-Authentication-Info **77**, 132
Proxy-Authorization **77**, 78, 111, 132
Range 61, 93, **95**, 112, 117, 132, 144
Referer **69**, 126, 132
Retry-After **72**, 114, 132
Server **73**, 126, 132
TE 38, 44, **70**, 132
Trailer 38, **39**, 132
Upgrade 17, 44, 45, **46**, 102, 113, 132
User-Agent **71**, 73, 126, 132
Vary 80, **85**, 119, 132, 132, 133, 143
Via 36, **44**, 67, 126, 132, 142
WWW-Authenticate **76**, 77, 110, 132
Fragment Identifiers 23
From header field **69**, 132

G

gateway **18**
Georgiev 123, **136**
GET method 16, 19, 37, 37, 53, 59, **61**, 130, 143
`Grammar`
`absolute-path` **21**
`absolute-URI` **21**
`Accept` **81**
`Accept-Charset` **83**
`Accept-Encoding` **83**
`Accept-Language` **84**
`Accept-Ranges` **96**
`acceptable-ranges` **96**
`Allow` **71**
ALPHA **13**
`asctime-date` **33**
`auth-param` **74**
`auth-scheme` **74**
`Authentication-Info` **77**
`authority` **21**
`Authorization` **76**
`BWS` **31**
`challenge` **74**
`codings` **83**
`comment` **31**
`complete-length` **97**
`Connection` **43**
`connection-option` **43**
`content-coding` **50**
`Content-Encoding` **50**
`Content-Language` **51**
`Content-Length` **52**
`Content-Location` **53**
`Content-Range` **97**
`Content-Type` **48**
CR **13**
`credentials` **75**
CRLF **13**
`ctext` **31**
CTL **13**
`Date` **39**
`date1` **33**

day **33**
day-name **33**
day-name-l **33**
delay-seconds **73**
DIGIT **13**
DQUOTE **13**
entity-tag **56**
ETag **56**
etagc **56**
Expect **68**
field-content **28**
field-name **27**, 39
field-value **28**
field-vchar **28**
first-pos **93**, 97
From **69**
GMT **33**
HEXDIG **13**
Host **41**
hour **33**
HTAB **13**
HTTP-date **32**
http-URI **22**
https-URI **22**
If-Match **86**
If-Modified-Since **88**
If-None-Match **87**
If-Range **91**
If-Unmodified-Since **89**
IMF-fixdate **33**
incl-range **97**
int-range **93**
language-range **84**
language-tag **51**
Last-Modified **55**
last-pos **93**, 97
LF **13**
Location **72**
Max-Forwards **44**
media-range **81**
media-type **49**
method **59**
minute **33**
month **33**
obs-date **33**
obs-text **28**
OCTET **13**
opaque-tag **56**
other-range **94**
OWS **31**
parameter **32**
parameter-name **32**
parameter-value **32**
parameters **32**
partial-URI **21**
port **21**
product **71**
product-version **71**
protocol-name **45**
protocol-version **45**
Proxy-Authenticate **77**
Proxy-Authentication-Info **78**
Proxy-Authorization **77**
pseudonym **45**
qdtext **31**
query **21**
quoted-pair **31**
quoted-string **31**
qvalue **81**
Range **95**
range-resp **97**
range-set **93**
range-spec **93**
range-unit **93**
ranges-specifier **93**
received-by **45**
received-protocol **45**
Referer **70**
Retry-After **73**
rfc850-date **33**
RWS **31**
second **33**
segment **21**
Server **73**
SP **13**
subtype **49**
suffix-length **94**
suffix-range **94**
t-codings **70**
tchar **30**
TE **70**
time-of-day **33**
token **30**
token68 **74**
Trailer **39**
transfer-coding **70**
transfer-parameter **70**
type **49**
unsatisfied-range **97**
Upgrade **46**
uri-host **21**
URI-reference **21**
User-Agent **71**
Vary **85**
VCHAR **13**
Via **45**
weak **56**
weight **81**
WWW-Authenticate **76**
year **33**
gzip (Coding Format) 51
gzip (content coding) **50**

H

HEAD method 37, 52, 53, 59, **62**, 130, 143
Header Fields
Accept 32, 49, 80, **81**, 112, 132, 133, 143
Accept-Charset **82**, 132, 143
Accept-Encoding 50, 58, 80, **83**, 112, 132, 133
Accept-Language 51, **84**, 132

Accept-Ranges 93, **96**, 132, 133, 143
Allow 60, **71**, 132
Authentication-Info **76**, 132
Authorization **76**, 77, 85, 110, 132
Connection 27, 43, **43**, 47, 71, 119, 132
Content-Encoding **49**, 50, 132
Content-Language **51**, 132
Content-Length **52**, 112, 132
Content-Location **53**, 63, 72, 132
Content-MD5 **133**
Content-Range 64, 93, **96**, 98, 98, 112, 132, 143, 144
Content-Type 29, **48**, 49, 132
Date 27, 37, **39**, 56, 132
ETag 54, **56**, 132
Expect 47, **68**, 102, 113, 132, 143
From **69**, 132
Host 25, 41, **41**, 132, 142
If-Match **86**, 132, 143
If-Modified-Since **88**, 132
If-None-Match **87**, 132
If-Range 87, 90, **90**, 96, 132
If-Unmodified-Since **89**, 132, 143, 143
Last-Modified 54, **55**, 132
Location 63, **72**, 107, 126, 132
Max-Forwards **44**, 66, 67, 132
Proxy-Authenticate **77**, 111, 132
Proxy-Authentication-Info **77**, 132
Proxy-Authorization **77**, 78, 111, 132
Range 61, 93, **95**, 112, 117, 132, 144
Referer **69**, 126, 132
Retry-After **72**, 114, 132
Server **73**, 126, 132
TE 38, 44, **70**, 132
Trailer 38, **39**, 132
Upgrade 17, 44, 45, **46**, 102, 113, 132
User-Agent **71**, 73, 126, 132
Vary 80, **85**, 119, 132, 132, 133, 143
Via 36, **44**, 67, 126, 132, 142
WWW-Authenticate **76**, 77, 110, 132
header section **36**
Host header field 25, 41, **41**, 132, 142
HPACK 125, **136**
http URI scheme **22**
HTTP/1.0 11, 106, **136**, 145
Section 9.3 106
HTTP/1.1 11, 12, 14, 28, 36, 36, 38, 41, 43, 44, 44, 46, 46, 46, 50, 52, 58, 62, 62, 65, 67, 69, 70, 116, 123, 126, **136**, 142
Appendix
Section 3.2.1 46
Section 3.2.4 46
Section 6 36
Section 6.1 44, 50, 116
Section 6.2 52
Section 7 46, 58, 70
Section 7.1.2 38
Section 9.5 43
Section 9.6 69
Section 10.1 67
Section 11 123
Section 11.2 28, 62, 62, 65, 126
Appendix C.2.2 44
HTTP/2 11, 36, 41, 43, 116, **136**
HTTP/3 11, 36, 41, **136**
https URI scheme **22**

I

idempotent **60**
If-Match header field **86**, 132, 143
If-Modified-Since header field **88**, 132
If-None-Match header field **87**, 132
If-Range header field 87, 90, **90**, 96, 132
If-Unmodified-Since header field **89**, 132, 143, 143
inbound **18**
incomplete **35**
interception proxy **19**
intermediary **18**
ISO-8859-1 28, **137**

K

Kri2001 28, **137**

L

Last-Modified header field 54, **55**, 132
list-based field 29
Location header field 63, **72**, 107, 126, 132

M

Max-Forwards header field **44**, 66, 67, 132
Media Type
multipart/byteranges **98**
multipart/x-byteranges 98
message 17, **35**
message abstraction **35**
messages **17**
metadata **54**
Method
* **130**
CONNECT 17, 37, 41, 52, 59, **65**, 116, 130
DELETE 59, **64**, 130, 143
GET 16, 19, 37, 37, 53, 59, **61**, 130, 143
HEAD 37, 52, 53, 59, **62**, 130, 143
OPTIONS 41, 44, 59, **66**, 130, 143
POST 37, 53, 59, 62, **62**, 130
PUT 37, 53, 59, **63**, 103, 103, 119, 130, 143
TRACE 44, 59, **67**, 130, 143
multipart/byteranges Media Type **98**
multipart/x-byteranges Media Type 98

N

non-transforming proxy **45**

O

OPTIONS method 41, 44, 59, **66**, 130, 143
origin **24**, 75
origin server **17**
outbound **18**
OWASP 123, 128, **137**

P

phishing **123**

POST method 37, 53, 59, 62, **62**, 130
Protection Space 75
proxy **18**
Proxy-Authenticate header field **77**, 111, 132
Proxy-Authentication-Info header field **77**, 132
Proxy-Authorization header field **77**, 78, 111, 132
PUT method 37, 53, 59, **63**, 103, 103, 119, 130, 143

R

Range header field 61, 93, **95**, 112, 117, 132, 144
Realm 75
recipient **17**
Referer header field **69**, 126, 132
representation **16**
request **17**
request target **41**
resource **16**, 21
response **17**
REST 16, 59, **137**
Retry-After header field **72**, 114, 132
reverse proxy **18**
RFC1919 19, **137**
RFC1950 51, 133, **135**
RFC1951 51, 133, **135**
RFC1952 51, 133, **135**
RFC2046 48, 49, 49, 49, 98, 98, **135**
Section 4.1.2 49
Section 4.5.1 48
Section 5.1 98
Section 5.1.1 49
RFC2047 28, **137**
RFC2068 11, 44, 107, **137**, 145
Section 10.3 107
Section 19.7.1 44
RFC2119 13, **135**
RFC2145 **137**, 145
RFC2295 79, **137**
RFC2324 113, **137**
RFC2557 53, **137**
Section 4 53
RFC2616 11, 57, 84, 133, **137**, 145
Section 3.11 57
Section 14.4 84
Section 14.15 133
RFC2617 **137**, 145
RFC2774 117, **137**
RFC2818 12, **137**, 145
RFC2978 49, 49, **137**
Section 2 49
Section 2.3 49
RFC3040 19, **137**
RFC3864 131, **137**
RFC3875 126, **137**
Section 4.1.18 126
RFC4033 123, **138**
RFC4289
RFC4559 17, **138**
RFC4647 84, 84, 84, 84, **135**, 141
Section 2.1 84, 141
Section 2.3 84
Section 3 84
Section 3.3.1 84
RFC4648 74, **135**
RFC5234 13, 13, 28, 29, **135**
Appendix B.1 13
RFC5280 26, **135**
Section 4.2.1.6 26
RFC5322 32, 33, 33, 39, 44, 69, 69, **135**, 141
Section 3.3 33, 33
Section 3.4 69, 69, 141
Section 3.6.1 39
Section 3.6.7 44
RFC5646 51, 51, 52, **135**, 141
Section 2.1 51, 141
RFC5789 80, 98, **138**
Section 3.1 80
RFC5905 32, **138**
RFC6125 25, 26, **135**
Section 6 25
Section 6.2.1 26
RFC6365 13, 49, **135**
RFC6454 24, **138**
RFC6585 125, **138**
RFC6648
RFC6838
RFC7230 11, 12, 12, 12, **138**, 145
RFC7231 12, 109, 133, **138**, 145
Appendix B 109, 133
RFC7232 12, **138**, 145
RFC7233 12, **138**, 145
RFC7234 **138**, 145
RFC7235 11, 12, **138**, 145
RFC7405 13, **135**
RFC7538 12, 106, 109, **138**, 143
Section 4 109
RFC7540 **138**, 143
RFC7578 49, **138**
RFC7595
RFC7615 12, **138**
RFC7616 74, 77, **138**
Section 3.5 77
RFC7617 74, **138**
RFC7694 12, **139**, 143, 144
RFC8126 116, 117, 118, 120, 121, 121, 122, **139**
Section 4.4 122
Section 4.6 118
Section 4.8 116, 117, 120, 121, 121
RFC8174 13, **135**
RFC8187 28, **139**
RFC8246 92, **139**
RFC8288 107, **139**
RFC8336 25, 123, **139**
RFC8615 118, **139**
RFC8941 119, 120, **139**

S

safe **60**
satisfiable range **93**
secured **22**
selected representation **16**, 54, 86

self-descriptive **35**
sender **17**
server **16**
Server header field **73**, 126, 132
singleton field 29
Sniffing 48, **139**
spider **17**
Status Code 101
Status Codes
Final 101
Informational 101
Interim 101
Status Codes Classes
1xx Informational **101**
2xx Successful **102**
3xx Redirection 47, **106**, 119, 143
4xx Client Error **110**
5xx Server Error **114**

T
target resource **41**
target URI **41**
TCP 22, **135**
TE header field 38, 44, **70**, 132
TLS13 18, 22, 65, **135**
TRACE method 44, 59, **67**, 130, 143
Trailer Fields **38**
ETag 54, **56**, 132
Trailer header field 38, **39**, 132
trailer section **38**
trailers **38**
transforming proxy **45**
transparent proxy **19**
tunnel **18**

U
unsatisfiable range **93**
Upgrade header field 17, 44, 45, **46**, 102, 113, 132
upstream **18**
URI **21**
origin **24**
URI 16, 21, 21, 21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23, 23, 26, 41, 53, 61, 70, 70, 72, 72, 74, 123, **136**, 141, 141, 141, 141, 141, 141, 141, 141, 141, 142
Section 2.1 23
Section 2.2 23
Section 3.2 21, 141
Section 3.2.1 23
Section 3.2.2 21, 22, 22, 26, 141
Section 3.2.3 21, 22, 22, 141
Section 3.3 21, 21, 141, 141
Section 3.4 21, 141
Section 3.5 23, 41
Section 4.1 21, 141
Section 4.2 21, 72, 141
Section 4.3 21, 23, 141
Section 5 53, 70, 72
Section 6 22
Section 6.2 23
Section 6.2.3 23
Section 7 123
URI reference **21**
URI scheme
http **22**
https **22**
USASCII 28, **136**
user agent **17**
User-Agent header field **71**, 73, 126, 132

V
validator **54**
strong **54**
weak **54**
Vary header field 80, **85**, 119, 132, 132, 133, 143
Via header field 36, **44**, 67, 126, 132, 142

W
WEBDAV 54, 86, **139**, 143
Section 10.4 86
Section 11.2 143
Welch 50, 133, **136**
WWW-Authenticate header field **76**, 77, 110, 132

X
x-compress (content coding) **50**
x-gzip (content coding) **50**

Authors' Addresses

Roy T. Fielding (editor)
Adobe
345 Park Ave
San Jose, CA 95110
United States of America
Email: fielding@gbiv.com
URI: https://roy.gbiv.com/

Mark Nottingham (editor)
Fastly
Prahran
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/

Julian Reschke (editor)
greenbytes GmbH
Hafenweg 16
48155 Münster
Germany
Email: julian.reschke@greenbytes.de
URI: https://greenbytes.de/tech/webdav/