Internet Engineering Task Force (IETF)
Request for Comments: 7231
Obsoletes: 2616
Updates: 2817
Category: Standards Track
ISSN: 2070-1721

R. Fielding, Editor
Adobe
J. Reschke, Editor
greenbytes

June 2014

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines the semantics of HTTP/1.1 messages, as expressed by request methods, request header fields, response status codes, and response header fields, along with the payload of messages (metadata and body content) and mechanisms for content negotiation.

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 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7231[1].

Copyright Notice

[1] http://www.rfc-editor.org/info/rfc7231
[2] http://trustee.ietf.org/license-info

created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1 Introduction....7
1.1 Conformance and Error Handling....7
1.2 Syntax Notation....7
2 Resources....8
3 Representations....9
3.1 Representation Metadata....9
3.1.1 Processing Representation Data....9
3.1.2 Encoding for Compression or Integrity....11
3.1.3 Audience Language....12
3.1.4 Identification....13
3.2 Representation Data....14
3.3 Payload Semantics....15
3.4 Content Negotiation....15
3.4.1 Proactive Negotiation....15
3.4.2 Reactive Negotiation....16
4 Request Methods....18
4.1 Overview....18
4.2 Common Method Properties....19
4.2.1 Safe Methods....19
4.2.2 Idempotent Methods....19
4.2.3 Cacheable Methods....19
4.3 Method Definitions....20
4.3.1 GET....20
4.3.2 HEAD....20
4.3.3 POST....20
4.3.4 PUT....21
4.3.5 DELETE....22
4.3.6 CONNECT....23
4.3.7 OPTIONS....24
4.3.8 TRACE....24
5 Request Header Fields....26
5.1 Controls....26
5.1.1 Expect....26
5.1.2 Max-Forwards....27
5.2 Conditionals....28
5.3 Content Negotiation....28
5.3.1 Quality Values....28
5.3.2 Accept....28
5.3.3 Accept-Charset....30
5.3.4 Accept-Encoding....30
5.3.5 Accept-Language....31
5.4 Authentication Credentials....32

5.5 Request Context....32
5.5.1 From....32
5.5.2 Referer....33
5.5.3 User-Agent....33

6 Response Status Codes....35

6.1 Overview of Status Codes....35

6.2 Informational 1xx....36
6.2.1 100 Continue....36
6.2.2 101 Switching Protocols....36

6.3 Successful 2xx....37
6.3.1 200 OK....37
6.3.2 201 Created....37
6.3.3 202 Accepted....37
6.3.4 203 Non-Authoritative Information....38
6.3.5 204 No Content....38
6.3.6 205 Reset Content....38

6.4 Redirection 3xx....38
6.4.1 300 Multiple Choices....39
6.4.2 301 Moved Permanently....40
6.4.3 302 Found....40
6.4.4 303 See Other....40
6.4.5 305 Use Proxy....41
6.4.6 306 (Unused)....41
6.4.7 307 Temporary Redirect....41

6.5 Client Error 4xx....41
6.5.1 400 Bad Request....41
6.5.2 402 Payment Required....41
6.5.3 403 Forbidden....41
6.5.4 404 Not Found....41
6.5.5 405 Method Not Allowed....42
6.5.6 406 Not Acceptable....42
6.5.7 408 Request Timeout....42
6.5.8 409 Conflict....42
6.5.9 410 Gone....42
6.5.10 411 Length Required....43
6.5.11 413 Payload Too Large....43
6.5.12 414 URI Too Long....43
6.5.13 415 Unsupported Media Type....43
6.5.14 417 Expectation Failed....43
6.5.15 426 Upgrade Required....43

6.6 Server Error 5xx....44
6.6.1 500 Internal Server Error....44
6.6.2 501 Not Implemented....44
6.6.3 502 Bad Gateway....44
6.6.4 503 Service Unavailable....44
6.6.5 504 Gateway Timeout....44
6.6.6 505 HTTP Version Not Supported....44

7 Response Header Fields....46

7.1 Control Data....46

7.1.1 Origination Date ... 46
7.1.2 Location ... 49
7.1.3 Retry-After ... 49
7.1.4 Vary ... 50
7.2 Validator Header Fields ... 50
7.3 Authentication Challenges ... 51
7.4 Response Context ... 51
7.4.1 Allow ... 51
7.4.2 Server ... 51
8 IANA Considerations ... 53
8.1 Method Registry ... 53
8.1.1 Procedure ... 53
8.1.2 Considerations for New Methods ... 53
8.1.3 Registrations ... 53
8.2 Status Code Registry ... 53
8.2.1 Procedure ... 54
8.2.2 Considerations for New Status Codes ... 54
8.2.3 Registrations ... 54
8.3 Header Field Registry ... 55
8.3.1 Considerations for New Header Fields ... 55
8.3.2 Registrations ... 56
8.4 Content Coding Registry ... 57
8.4.1 Procedure ... 57
8.4.2 Registrations ... 57
9 Security Considerations ... 58
9.1 Attacks Based on File and Path Names ... 58
9.2 Attacks Based on Command, Code, or Query Injection ... 58
9.3 Disclosure of Personal Information ... 58
9.4 Disclosure of Sensitive Information in URIs ... 58
9.5 Disclosure of Fragment after Redirects ... 59
9.6 Disclosure of Product Information ... 59
9.7 Browser Fingerprinting ... 59
10 Acknowledgments ... 61
11 References ... 62
11.1 Normative References ... 62
11.2 Informative References ... 62
A Differences between HTTP and MIME ... 64
A.1 MIME-Version ... 64
A.2 Conversion to Canonical Form ... 64
A.3 Conversion of Date Formats ... 64
A.4 Conversion of Content-Encoding ... 64

A.5 Conversion of Content-Transfer-Encoding ... 64
A.6 MHTML and Line Length Limitations ... 65

B Changes from RFC 2616 ... 66

C Imported ABNF ... 68

D Collected ABNF ... 69

Index ... 71

Authors' Addresses ... 74

1. Introduction

Each Hypertext Transfer Protocol (HTTP) message is either a request or a response. A server listens on a connection for a request, parses each message received, interprets the message semantics in relation to the identified request target, and responds to that request with one or more response messages. A client constructs request messages to communicate specific intentions, examines received responses to see if the intentions were carried out, and determines how to interpret the results. This document defines HTTP/1.1 request and response semantics in terms of the architecture defined in [RFC7230].

HTTP provides a uniform interface for interacting with a resource (Section 2), regardless of its type, nature, or implementation, via the manipulation and transfer of representations (Section 3).

HTTP semantics include the intentions defined by each request method (Section 4), extensions to those semantics that might be described in request header fields (Section 5), the meaning of status codes to indicate a machine-readable response (Section 6), and the meaning of other control data and resource metadata that might be given in response header fields (Section 7).

This document also defines representation metadata that describe how a payload is intended to be interpreted by a recipient, the request header fields that might influence content selection, and the various selection algorithms that are collectively referred to as "*content negotiation*" (Section 3.4).

1.1. Conformance and Error Handling

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

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [RFC7230].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [RFC7230], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix C describes rules imported from other documents. Appendix D shows the collected grammar with all list operators expanded to standard ABNF notation.

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

2. Resources

The target of an HTTP request is called a "*resource*". HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI), as described in Section 2.7 of [RFC7230].

When a client constructs an HTTP/1.1 request message, it sends the target URI in one of various forms, as defined in (Section 5.3 of [RFC7230]). When a request is received, the server reconstructs an effective request URI for the target resource (Section 5.5 of [RFC7230]).

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 4) and a few request-modifying header fields (Section 5). If there is a conflict between the method semantics and any semantic implied by the URI itself, as described in Section 4.2.1, the method semantics take precedence.

3. Representations

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

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

An origin server might be provided with, or be capable of generating, multiple representations that are each intended to reflect the current state of a target resource. In such cases, some algorithm is used by the origin server to select one of those representations as most applicable to a given request, usually based on content negotiation. This "*selected representation*" is used to provide the data and metadata for evaluating conditional requests [RFC7232] and constructing the payload for 200 (OK) and 304 (Not Modified) responses to GET (Section 4.3.1).

3.1. Representation Metadata

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

The following header fields convey representation metadata:

Header Field Name	Defined in...
Content-Type	Section 3.1.1.5
Content-Encoding	Section 3.1.2.2
Content-Language	Section 3.1.3.2
Content-Location	Section 3.1.4.2

3.1.1. Processing Representation Data

3.1.1.1. Media Type

HTTP uses Internet media types [RFC2046] in the Content-Type (Section 3.1.1.5) and Accept (Section 5.3.2) header fields in order to provide open and extensible data typing and type negotiation. Media types define both a data format and various processing models: how to process that data in accordance with each context in which it is received.

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

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

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

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

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

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

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

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

3.1.1.2. Charset

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

```
charset = token
```

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

3.1.1.3. Canonicalization and Text Defaults

Internet media types are registered with a canonical form in order to be interoperable among systems with varying native encoding formats. Representations selected or transferred via HTTP ought to be in canonical form, for many of the same reasons described by the Multipurpose Internet Mail Extensions (MIME) [RFC2045]. However, the performance characteristics of email deployments (i.e., store and forward messages to peers) are significantly different from those common to HTTP and the Web (server-based information services). Furthermore, MIME's constraints for the sake of compatibility with older mail transfer protocols do not apply to HTTP (see Appendix A).

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

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

3.1.1.4. Multipart Types

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

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

3.1.1.5. Content-Type

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

```
Content-Type = media-type
```

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

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

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

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

3.1.2. Encoding for Compression or Integrity

3.1.2.1. Content Codings

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

```
content-coding   = token
```

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

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

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

deflate: See Section 4.2.2 of [RFC7230].

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

3.1.2.2. Content-Encoding

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

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

An example of its use is

```
Content-Encoding: gzip
```

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

Unlike Transfer-Encoding (Section 3.3.1 of [RFC7230]), 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.

3.1.3. Audience Language

3.1.3.1. 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 5.3.5, whereas Content-Language uses the language-tag production defined below.

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

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

```
fr, en-US, es-419, az-Arab, x-pig-latin, man-Nkoo-GN
```

See [RFC5646] for further information.

3.1.3.2. Content-Language

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

```
Content-Language = 1#language-tag
```

Language tags are defined in Section 3.1.3.1. 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.

3.1.4. Identification

3.1.4.1. Identifying a Representation

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

For a request message:

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

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

1. If the request method is GET or HEAD and the response status code is 200 (OK), 204 (No Content), 206 (Partial Content), or 304 (Not Modified), the payload is a representation of the resource identified by the effective request URI (Section 5.5 of [RFC7230]).
2. If the request method is GET or HEAD and the response status code is 203 (Non-Authoritative Information), the payload is a potentially modified or enhanced representation of the target resource as provided by an intermediary.
3. If the response has a Content-Location header field and its field-value is a reference to the same URI as the effective request URI, the payload is a representation of the resource identified by the effective request URI.
4. If the response has a Content-Location header field and its field-value is a reference to a URI different from the effective request URI, then the sender asserts that the payload is a representation of the resource identified by the Content-Location field-value. However, such an assertion cannot be trusted unless it can be verified by other means (not defined by this specification).
5. Otherwise, the payload is unidentified.

3.1.4.2. Content-Location

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

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

The Content-Location value is not a replacement for the effective Request URI (Section 5.5 of [RFC7230]). It is representation metadata. It has the same syntax and semantics as the header field of the same name defined for MIME body parts in Section 4 of [RFC2557]. However, its appearance in an HTTP message has some special implications for HTTP recipients.

If Content-Location is included in a 2xx (Successful) response message and its value refers (after conversion to absolute form) to a URI that is the same as the effective request URI, then the recipient MAY consider the payload to be a current representation of that resource at the time indicated by the message origination date. For a GET (Section 4.3.1) or HEAD (Section 4.3.2) 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 4.3.4) or POST (Section 4.3.3), it implies that the server's response contains the new representation of that resource, thereby distinguishing it from representations that might only report about the action (e.g., "It worked!"). This allows authoring applications to update their local copies without the need for a subsequent GET request.

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

- For a response to a GET or HEAD request, this is an indication that the effective request URI refers to a resource that is subject to content negotiation and the Content-Location field-value is a more specific identifier for the selected representation.
- For a 201 (Created) response to a state-changing method, a Content-Location field-value that is identical to the Location field-value indicates that this payload is a current representation of the newly created resource.
- Otherwise, such a Content-Location indicates that this payload is a representation reporting on the requested action's status and that the same report is available (for future access with GET) at the given URI. For example, a purchase transaction made via a POST request might include a receipt document as the payload of the 200 (OK) response; the Content-Location field-value provides an identifier for retrieving a copy of that same receipt in the future.

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

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

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

3.2. Representation Data

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

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

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

3.3. Payload Semantics

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

The purpose of a payload in a request is defined by the method semantics. For example, a representation in the payload of a PUT request (Section 4.3.4) represents the desired state of the target resource if the request is successfully applied, whereas a representation in the payload of a POST request (Section 4.3.3) represents information to be processed by the target resource.

In a response, the payload's purpose is defined by both the request method and the response status code. For example, the payload of a 200 (OK) response to GET (Section 4.3.1) represents the current state of the target resource, as observed at the time of the message origination date (Section 7.1.1.2), whereas the payload of the same status code in a response to POST might represent either the processing result or the new state of the target resource after applying the processing. Response messages with an error status code usually contain a payload that represents the error condition, such that it describes the error state and what next steps are suggested for resolving it.

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

Header Field Name	Defined in...
Content-Length	Section 3.3.2 of [RFC7230]
Content-Range	Section 4.2 of [RFC7233]
Trailer	Section 4.4 of [RFC7230]
Transfer-Encoding	Section 3.3.1 of [RFC7230]

3.4. Content Negotiation

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

This specification defines two patterns of content negotiation that can be made visible within the protocol: "proactive", where the server selects the representation based upon the user agent's stated preferences, and "reactive" negotiation, where the server provides a list of representations for the user agent to choose from. Other patterns of content negotiation include "conditional content", where the representation consists of multiple parts that are selectively rendered based on user agent parameters, "active content", where the representation contains a script that makes additional (more specific) requests based on the user agent characteristics, and "Transparent Content Negotiation" ([RFC2295]), where content selection is performed by an intermediary. These patterns are not mutually exclusive, and each has trade-offs in applicability and practicality.

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

3.4.1. Proactive Negotiation

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

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

Proactive negotiation has serious disadvantages:

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

3.4.2. Reactive Negotiation

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

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

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

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

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

4. Request Methods

4.1. Overview

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

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

```
method = token
```

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

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

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

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

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

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

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

Implemented) status code. When a request method is received that is known by an origin server but not allowed for the target resource, the origin server SHOULD respond with the 405 (Method Not Allowed) status code.

4.2. Common Method Properties

4.2.1. Safe Methods

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

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

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

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

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

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

4.2.2. Idempotent Methods

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

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

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

4.2.3. Cacheable Methods

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

4.3. Method Definitions

4.3.1. GET

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

It is tempting to think of resource identifiers as remote file system pathnames and of representations as being a copy of the contents of such files. In fact, that is how many resources are implemented (see Section 9.1 for related security considerations). However, there are no such limitations in practice. The HTTP interface for a resource is just as likely to be implemented as a tree of content objects, a programmatic view on various database records, or a gateway to other information systems. Even when the URI mapping mechanism is tied to a file system, an origin server might be configured to execute the files with the request as input and send the output as the representation rather than transfer the files directly. Regardless, only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a response to GET.

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

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

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

4.3.2. HEAD

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

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

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

4.3.3. POST

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

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

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

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

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

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

If the result of processing a POST would be equivalent to a representation of an existing resource, an origin server MAY redirect the user agent to that resource by sending a 303 (See Other) response with the existing resource's identifier in the Location field. This has the benefits of providing the user agent a resource identifier and transferring the representation via a method more amenable to shared caching, though at the cost of an extra request if the user agent does not already have the representation cached.

4.3.4. PUT

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

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

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

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

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

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

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

c. reject the request with a 415 (Unsupported Media Type) response indicating that the target resource is limited to "text/html", perhaps including a link to a different resource that would be a suitable target for the new representation.

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

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

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

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

A PUT request applied to the target resource can have side effects on other resources. For example, an article might have a URI for identifying "the current version" (a resource) that is separate from the URIs identifying each particular version (different resources that at one point shared the same state as the current version resource). A successful PUT request on "the current version" URI might therefore create a new version resource in addition to changing the state of the target resource, and might also cause links to be added between the related resources.

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

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

4.3.5. DELETE

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

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

Relatively few resources allow the DELETE method — its primary use is for remote authoring environments, where the user has some direction regarding its effect. For example, a resource that was previously created using a PUT request, or identified via the Location header field after a 201 (Created) response to a POST request, might allow a corresponding DELETE request to undo those actions. Similarly, custom user agent implementations that implement an authoring function, such as revision control clients using HTTP for remote operations, might use DELETE based on an assumption that the server's URI space has been crafted to correspond to a version repository.

If a DELETE method is successfully applied, the origin server SHOULD send a 202 (Accepted) status code if the action will likely succeed but has not yet been enacted, a 204 (No Content) status code if the action has been enacted and no further information is to be supplied, or a 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status.

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

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

4.3.6. CONNECT

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

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

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

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

```

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

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

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

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

```

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

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

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

Responses to the CONNECT method are not cacheable.

4.3.7. OPTIONS

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

An OPTIONS request with an asterisk ("*") as the request-target (Section 5.3 of [RFC7230]) applies to the server in general rather than to a specific resource. Since a server's communication options typically depend on the resource, the "*" request is only useful as a "ping" or "no-op" type of method; it does nothing beyond allowing the client to test the capabilities of the server. For example, this can be used to test a proxy for HTTP/1.1 conformance (or lack thereof).

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

A server generating a successful response to OPTIONS SHOULD send any header fields that might indicate optional features implemented by the server and applicable to the target resource (e.g., Allow), including potential extensions not defined by this specification. The response payload, if any, might also describe the communication options in a machine or human-readable representation. A standard format for such a representation is not defined by this specification, but might be defined by future extensions to HTTP. A server MUST generate a Content-Length field with a value of "0" if no payload body is to be sent in the response.

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

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

Responses to the OPTIONS method are not cacheable.

4.3.8. TRACE

The TRACE method requests a remote, application-level loop-back of the request message. The final recipient of the request SHOULD reflect the message received, excluding some fields described below, back to the client as the message body of a 200 (OK) response with a Content-Type of "message/http" (Section 8.3.1 of

[RFC7230]). The final recipient is either the origin server or the first server to receive a Max-Forwards value of zero (0) in the request (Section 5.1.2).

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

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

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

Responses to the TRACE method are not cacheable.

5. Request Header Fields

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

5.1. Controls

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

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

5.1.1. Expect

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

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

The Expect field-value is case-insensitive.

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

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

For example, a request that begins with

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

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

Requirements for clients:

- A client MUST NOT generate a 100-continue expectation in a request that does not include a message body.
- A client that will wait for a 100 (Continue) response before sending the request message body MUST send an Expect header field containing a 100-continue expectation.

- A client that sends a 100-continue expectation is not required to wait for any specific length of time; such a client MAY proceed to send the message body even if it has not yet received a response. Furthermore, since 100 (Continue) responses cannot be sent through an HTTP/1.0 intermediary, such a client SHOULD NOT wait for an indefinite period before sending the message body.
- A client that receives a 417 (Expectation Failed) status code in response to a request containing a 100-continue expectation SHOULD repeat that request without a 100-continue expectation, since the 417 response merely indicates that the response chain does not support expectations (e.g., it passes through an HTTP/1.0 server).

Requirements for servers:

- A server that receives a 100-continue expectation in an HTTP/1.0 request MUST ignore that expectation.
- A server MAY omit sending a 100 (Continue) response if it has already received some or all of the message body for the corresponding request, or if the framing indicates that there is no message body.
- A server that sends a 100 (Continue) response MUST ultimately send a final status code, once the message body is received and processed, unless the connection is closed prematurely.
- A server that responds with a final status code before reading the entire message body SHOULD indicate in that response whether it intends to close the connection or continue reading and discarding the request message (see Section 6.6 of [RFC7230]).

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

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

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

5.1.2. Max-Forwards

The "Max-Forwards" header field provides a mechanism with the TRACE (Section 4.3.8) and OPTIONS (Section 4.3.7) 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.

5.2. Conditionals

The HTTP conditional request header fields [RFC7232] allow a client to place a precondition on the state of the target resource, so that the action corresponding to the method semantics will not be applied if the precondition evaluates to false. Each precondition defined by this specification consists of a comparison between a set of validators obtained from prior representations of the target resource to the current state of validators for the selected representation (Section 7.2). Hence, these preconditions evaluate whether the state of the target resource has changed since a given state known by the client. The effect of such an evaluation depends on the method semantics and choice of conditional, as defined in Section 5 of [RFC7232].

Header Field Name	Defined in...
If-Match	Section 3.1 of [RFC7232]
If-None-Match	Section 3.2 of [RFC7232]
If-Modified-Since	Section 3.3 of [RFC7232]
If-Unmodified-Since	Section 3.4 of [RFC7232]
If-Range	Section 3.2 of [RFC7233]

5.3. Content Negotiation

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

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

5.3.1. Quality Values

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

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

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

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

5.3.2. Accept

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

```
Accept = #( media-range [ accept-params ] )

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

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

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

> **Note:** Use of the "q" parameter name to separate media type parameters from Accept extension parameters is due to historical practice. Although this prevents any media type parameter named "q" from being used with a media range, such an event is believed to be unlikely given the lack of any "q" parameters in the IANA media type registry and the rare usage of any media type parameters in Accept. Future media types are discouraged from registering any parameter named "q".

The example

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

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

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

A more elaborate example is

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

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

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

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

have the following precedence:

1. text/plain;format=flowed
2. text/plain

3. text/*
4. */*

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

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

would cause the following values to be associated:

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

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

5.3.3. Accept-Charset

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

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

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

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

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

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

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

5.3.4. Accept-Encoding

The "Accept-Encoding" header field can be used by user agents to indicate what response content-codings (Section 3.1.2.1) are acceptable in the response. An "identity" token is used as a synonym for "no encoding" in order to communicate when no encoding is preferred.

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

Each codings value MAY be given an associated quality value representing the preference for that encoding, as defined in Section 5.3.1. The asterisk "*" symbol in an Accept-Encoding field matches any available content-coding not explicitly listed in the header field.

For example,

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

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

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

1. If no Accept-Encoding field is in the request, any content-coding is considered acceptable by the user agent.
2. If the representation has no content-coding, then it is acceptable by default unless specifically excluded by the Accept-Encoding field stating either "identity;q=0" or "*;q=0" without a more specific entry for "identity".
3. If the representation's content-coding is one of the content-codings listed in the Accept-Encoding field, then it is acceptable unless it is accompanied by a qvalue of 0. (As defined in Section 5.3.1, a qvalue of 0 means "not acceptable".)
4. If multiple content-codings are acceptable, then the acceptable content-coding with the highest non-zero qvalue is preferred.

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

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

5.3.5. Accept-Language

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

```
Accept-Language = 1#( language-range [ weight ] )
language-range  =
          <language-range, see [RFC4647], Section 2.1>
```

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

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

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

A request without any Accept-Language header field implies that the user agent will accept any language in response. If the header field is present in a request and none of the available representations for the response

have a matching language tag, the origin server can either disregard the header field by treating the response as if it is not subject to content negotiation or honor the header field by sending a 406 (Not Acceptable) response. However, the latter is not encouraged, as doing so can prevent users from accessing content that they might be able to use (with translation software, for example).

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

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

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

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

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

5.4. Authentication Credentials

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

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

5.5. Request Context

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

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

5.5.1. From

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

```
From    = mailbox

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

An example is:

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

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

A robotic user agent SHOULD send a valid From header field so that the person responsible for running the robot can be contacted if problems occur on servers, such as if the robot is sending excessive, unwanted, or invalid requests.

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

5.5.2. Referer

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

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

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

Example:

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

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

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

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

5.5.3. User-Agent

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

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

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

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

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

Example:

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

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

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

6. Response Status Codes

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

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

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

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

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

6.1. Overview of Status Codes

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

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

Code	Reason-Phrase	Defined in...
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5
205	Reset Content	Section 6.3.6
206	Partial Content	Section 4.1 of [RFC7233]
300	Multiple Choices	Section 6.4.1
301	Moved Permanently	Section 6.4.2
302	Found	Section 6.4.3
303	See Other	Section 6.4.4
304	Not Modified	Section 4.1 of [RFC7232]
305	Use Proxy	Section 6.4.5
307	Temporary Redirect	Section 6.4.7
400	Bad Request	Section 6.5.1
401	Unauthorized	Section 3.1 of [RFC7235]
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4

Code	Reason-Phrase	Defined in...
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
407	Proxy Authentication Required	Section 3.2 of [RFC7235]
408	Request Timeout	Section 6.5.7
409	Conflict	Section 6.5.8
410	Gone	Section 6.5.9
411	Length Required	Section 6.5.10
412	Precondition Failed	Section 4.2 of [RFC7232]
413	Payload Too Large	Section 6.5.11
414	URI Too Long	Section 6.5.12
415	Unsupported Media Type	Section 6.5.13
416	Range Not Satisfiable	Section 4.4 of [RFC7233]
417	Expectation Failed	Section 6.5.14
426	Upgrade Required	Section 6.5.15
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

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

6.2. Informational 1xx

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

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

A proxy MUST forward 1xx responses unless the proxy itself requested the generation of the 1xx response. For example, if a proxy adds an "Expect: 100-continue" field when it forwards a request, then it need not forward the corresponding 100 (Continue) response(s).

6.2.1. 100 Continue

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

When the request contains an Expect header field that includes a 100-continue expectation, the 100 response indicates that the server wishes to receive the request payload body, as described in Section 5.1.1. The client ought to continue sending the request and discard the 100 response.

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

6.2.2. 101 Switching Protocols

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

protocol being used on this connection. The server MUST generate an Upgrade header field in the response that indicates which protocol(s) will be switched to immediately after the empty line that terminates the 101 response.

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

6.3. Successful 2xx

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

6.3.1. 200 OK

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

GET	a representation of the target resource;
HEAD	the same representation as GET, but without the representation data;
POST	a representation of the status of, or results obtained from, the action;
PUT, DELETE	a representation of the status of the action;
OPTIONS	a representation of the communications options;
TRACE	a representation of the request message as received by the end server.

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

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

6.3.2. 201 Created

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

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

6.3.3. 202 Accepted

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

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

6.3.4. 203 Non-Authoritative Information

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

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

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

6.3.5. 204 No Content

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

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

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

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

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

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

6.3.6. 205 Reset Content

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

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

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

6.4. Redirection 3xx

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

There are several types of redirects:

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

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

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

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

6.4.1. 300 Multiple Choices

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

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

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

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

> **Note:** The original proposal for the 300 status code defined the URI header field as providing a list of alternative representations, such that it would be usable for 200, 300, and 406 responses and be transferred in responses to the HEAD method. However, lack of deployment and disagreement over syntax led to

both URI and Alternates (a subsequent proposal) being dropped from this specification. It is possible to communicate the list using a set of Link header fields [RFC5988], each with a relationship of "alternate", though deployment is a chicken-and-egg problem.

6.4.2. 301 Moved Permanently

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

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

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

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

6.4.3. 302 Found

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

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

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

6.4.4. 303 See Other

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

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

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

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

6.4.5. 305 Use Proxy

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

6.4.6. 306 (Unused)

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

6.4.7. 307 Temporary Redirect

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

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

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

6.5. Client Error 4xx

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

6.5.1. 400 Bad Request

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

6.5.2. 402 Payment Required

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

6.5.3. 403 Forbidden

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

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

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

6.5.4. 404 Not Found

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

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

6.5.5. 405 Method Not Allowed

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

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

6.5.6. 406 Not Acceptable

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

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

6.5.7. 408 Request Timeout

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

6.5.8. 409 Conflict

The *409 (Conflict)* status code indicates that the request could not be completed due to a conflict with the current state of the target resource. This code is used in situations where the user might be able to resolve the conflict and resubmit the request. The server SHOULD generate a payload that includes enough information for a user to recognize the source of the conflict.

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

6.5.9. 410 Gone

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

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging

to individuals no longer associated with the origin server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time — that is left to the discretion of the server owner.

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

6.5.10. 411 Length Required

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

6.5.11. 413 Payload Too Large

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

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

6.5.12. 414 URI Too Long

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

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

6.5.13. 415 Unsupported Media Type

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

6.5.14. 417 Expectation Failed

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

6.5.15. 426 Upgrade Required

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

Example:

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

This service requires use of the HTTP/3.0 protocol.
```

6.6. Server Error 5xx

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

6.6.1. 500 Internal Server Error

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

6.6.2. 501 Not Implemented

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

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

6.6.3. 502 Bad Gateway

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

6.6.4. 503 Service Unavailable

The *503 (Service Unavailable)* status code indicates that the server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay. The server MAY send a Retry-After header field (Section 7.1.3) to suggest an appropriate amount of time for the client to wait before retrying the request.

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

6.6.5. 504 Gateway Timeout

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

6.6.6. 505 HTTP Version Not Supported

The *505 (HTTP Version Not Supported)* status code indicates that the server does not support, or refuses to support, the major version of HTTP that was used in the request message. The server is indicating that it is

unable or unwilling to complete the request using the same major version as the client, as described in Section 2.6 of [RFC7230], other than with this error message. The server SHOULD generate a representation for the 505 response that describes why that version is not supported and what other protocols are supported by that server.

7. Response Header Fields

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

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

7.1. Control Data

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

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

7.1.1. Origination Date

7.1.1.1. Date/Time Formats

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

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

An example of the preferred format is

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

Examples of the two obsolete formats are

```
Sunday, 06-Nov-94 08:49:37 GMT   ; obsolete RFC 850 format
Sun Nov  6 08:49:37 1994         ; ANSI C's asctime() format
```

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

An HTTP-date value represents time as an instance of Coordinated Universal Time (UTC). The first two formats indicate UTC by the three-letter abbreviation for Greenwich Mean Time, "GMT", a predecessor of the UTC name; values in the asctime format are assumed to be in UTC. A sender that generates HTTP-date values from a local clock ought to use NTP ([RFC5905]) or some similar protocol to synchronize its clock to UTC.

Preferred format:

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

day-name     = %x4D.6F.6E ; "Mon", case-sensitive
             / %x54.75.65 ; "Tue", case-sensitive
             / %x57.65.64 ; "Wed", case-sensitive
             / %x54.68.75 ; "Thu", case-sensitive
             / %x46.72.69 ; "Fri", case-sensitive
             / %x53.61.74 ; "Sat", case-sensitive
             / %x53.75.6E ; "Sun", case-sensitive

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

day          = 2DIGIT
month        = %x4A.61.6E ; "Jan", case-sensitive
             / %x46.65.62 ; "Feb", case-sensitive
             / %x4D.61.72 ; "Mar", case-sensitive
             / %x41.70.72 ; "Apr", case-sensitive
             / %x4D.61.79 ; "May", case-sensitive
             / %x4A.75.6E ; "Jun", case-sensitive
             / %x4A.75.6C ; "Jul", case-sensitive
             / %x41.75.67 ; "Aug", case-sensitive
             / %x53.65.70 ; "Sep", case-sensitive
             / %x4F.63.74 ; "Oct", case-sensitive
             / %x4E.6F.76 ; "Nov", case-sensitive
             / %x44.65.63 ; "Dec", case-sensitive
year         = 4DIGIT

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

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

hour         = 2DIGIT
minute       = 2DIGIT
second       = 2DIGIT
```

Obsolete formats:

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

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

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

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

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

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

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

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

7.1.1.2. Date

The "Date" header field represents the date and time at which the message was originated, having the same semantics as the Origination Date Field (orig-date) defined in Section 3.6.1 of [RFC5322]. The field value is an HTTP-date, as defined in Section 7.1.1.1.

```
Date = HTTP-date
```

An example is

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

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

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

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

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

7.1.2. Location

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

```
Location = URI-reference
```

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

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

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

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

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

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

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

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

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

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

> **Note:** Some recipients attempt to recover from Location fields that are not valid URI references. This specification does not mandate or define such processing, but does allow it for the sake of robustness.
>
> **Note:** The Content-Location header field (Section 3.1.4.2) differs from Location in that the Content-Location refers to the most specific resource corresponding to the enclosed representation. It is therefore possible for a response to contain both the Location and Content-Location header fields.

7.1.3. Retry-After

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

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

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

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

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

Two examples of its use are

```
Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120
```

In the latter example, the delay is 2 minutes.

7.1.4. Vary

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

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

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

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

For example, a response that contains

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

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

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

1. To inform cache recipients that they MUST NOT use this response to satisfy a later request unless the later request has the same values for the listed fields as the original request (Section 4.1 of [RFC7234]). In other words, Vary expands the cache key required to match a new request to the stored cache entry.
2. To inform user agent recipients that this response is subject to content negotiation (Section 5.3) and that a different representation might be sent in a subsequent request if additional parameters are provided in the listed header fields (proactive negotiation).

An origin server SHOULD send a Vary header field when its algorithm for selecting a representation varies based on aspects of the request message other than the method and request target, unless the variance cannot be crossed or the origin server has been deliberately configured to prevent cache transparency. For example, there is no need to send the Authorization field name in Vary because reuse across users is constrained by the field definition (Section 4.2 of [RFC7235]). Likewise, an origin server might use Cache-Control directives (Section 5.2 of [RFC7234]) to supplant Vary if it considers the variance less significant than the performance cost of Vary's impact on caching.

7.2. Validator Header Fields

Validator header fields convey metadata about the selected representation (Section 3). In responses to safe requests, validator fields describe the selected representation chosen by the origin server while handling the

response. Note that, depending on the status code semantics, the selected representation for a given response is not necessarily the same as the representation enclosed as response payload.

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

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

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

7.3. Authentication Challenges

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

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

7.4. Response Context

The remaining response header fields provide more information about the target resource for potential use in later requests.

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

7.4.1. 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 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.

7.4.2. Server

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

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

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

Example:

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

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

8. IANA Considerations

8.1. Method Registry

The "Hypertext Transfer Protocol (HTTP) Method Registry" defines the namespace for the request method token (Section 4). The method registry has been created and is now maintained at <http://www.iana.org/assignments/http-methods>.

8.1.1. Procedure

HTTP method registrations MUST include the following fields:

- Method Name (see Section 4)
- Safe ("yes" or "no", see Section 4.2.1)
- Idempotent ("yes" or "no", see Section 4.2.2)
- Pointer to specification text

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

8.1.2. 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 3.3 of [RFC7230]) needs to be independent of method semantics (aside from responses to HEAD), definitions of new methods cannot change the parsing algorithm or prohibit the presence of a message body on either the request or the response message. Definitions of new methods can specify that only a zero-length message body is allowed by requiring a Content-Length header field with a value of "0".

A new method definition needs to indicate whether it is safe (Section 4.2.1), idempotent (Section 4.2.2), cacheable (Section 4.2.3), what semantics are to be associated with the payload body if any is present in the request and what refinements the method makes to header field or status code semantics. If the new method is cacheable, its definition ought to describe how, and under what conditions, a cache can store a response and use it to satisfy a subsequent request. The new method ought to describe whether it can be made conditional (Section 5.2) and, if so, how a server responds when the condition is false. Likewise, if the new method might have some use for partial response semantics ([RFC7233]), it ought to document this, too.

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

8.1.3. Registrations

The "Hypertext Transfer Protocol (HTTP) Method Registry" has been populated with the registrations below:

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

8.2. Status Code Registry

The "Hypertext Transfer Protocol (HTTP) Status Code Registry" defines the namespace for the response status-code token (Section 6). The status code registry is maintained at <http://www.iana.org/assignments/http-status-codes>.

This section replaces the registration procedure for HTTP Status Codes previously defined in Section 7.1 of [RFC2817].

8.2.1. Procedure

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 [RFC5226], Section 4.1).

8.2.2. 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 6. To allow existing parsers to process the response message, new status codes cannot disallow a payload, although they can mandate a zero-length payload body.

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

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

The definition of a new status code ought to specify whether or not it is cacheable. Note that all status codes can be cached if the response they occur in has explicit freshness information; however, status codes that are defined as being cacheable are allowed to be cached without explicit freshness information. Likewise, the definition of a status code can place constraints upon cache behavior. See [RFC7234] for more information.

Finally, the definition of a new status code ought to indicate whether the payload has any implied association with an identified resource (Section 3.1.4.1).

8.2.3. Registrations

The status code registry has been updated with the registrations below:

Value	Description	Reference
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5

Value	Description	Reference
205	Reset Content	Section 6.3.6
300	Multiple Choices	Section 6.4.1
301	Moved Permanently	Section 6.4.2
302	Found	Section 6.4.3
303	See Other	Section 6.4.4
305	Use Proxy	Section 6.4.5
306	(Unused)	Section 6.4.6
307	Temporary Redirect	Section 6.4.7
400	Bad Request	Section 6.5.1
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
408	Request Timeout	Section 6.5.7
409	Conflict	Section 6.5.8
410	Gone	Section 6.5.9
411	Length Required	Section 6.5.10
413	Payload Too Large	Section 6.5.11
414	URI Too Long	Section 6.5.12
415	Unsupported Media Type	Section 6.5.13
417	Expectation Failed	Section 6.5.14
426	Upgrade Required	Section 6.5.15
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

8.3. Header Field Registry

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

8.3.1. Considerations for New Header Fields

Header fields are key:value pairs that can be used to communicate data about the message, its payload, the target resource, or the connection (i.e., control data). See Section 3.2 of [RFC7230] for a general definition of header field syntax in HTTP messages.

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

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

New header field values typically have their syntax defined using ABNF ([RFC5234]), using the extension defined in Section 7 of [RFC7230] as necessary, and are usually constrained to the range of US-ASCII characters. Header fields needing a greater range of characters can use an encoding such as the one defined in [RFC5987].

Leading and trailing whitespace in raw field values is removed upon field parsing (Section 3.2.4 of [RFC7230]). Field definitions where leading or trailing whitespace in values is significant will have to use a container syntax such as quoted-string (Section 3.2.6 of [RFC7230]).

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

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

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

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

Many header fields use a format including (case-insensitively) named parameters (for instance, Content-Type, defined in Section 3.1.1.5). Allowing both unquoted (token) and quoted (quoted-string) syntax for the parameter value enables recipients to use existing parser components. When allowing both forms, the meaning of a parameter value ought to be independent of the syntax used for it (for an example, see the notes on parameter handling for media types in Section 3.1.1.1).

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

- Whether the field is a single value or whether it can be a list (delimited by commas; see Section 3.2 of [RFC7230]).

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

  Note that intermediaries and software libraries might combine multiple header field instances into a single one, despite the field's definition not allowing the list syntax. A robust format enables recipients to discover these situations (good example: "Content-Type", as the comma can only appear inside quoted strings; bad example: "Location", as a comma can occur inside a URI).
- Under what conditions the header field can be used; e.g., only in responses or requests, in all messages, only on responses to a particular request method, etc.
- Whether the field should be stored by origin servers that understand it upon a PUT request.
- Whether the field semantics are further refined by the context, such as by existing request methods or status codes.
- Whether it is appropriate to list the field-name in the Connection header field (i.e., if the header field is to be hop-by-hop; see Section 6.1 of [RFC7230]).
- Under what conditions intermediaries are allowed to insert, delete, or modify the field's value.
- Whether it is appropriate to list the field-name in a Vary response header field (e.g., when the request header field is used by an origin server's content selection algorithm; see Section 7.1.4).
- Whether the header field is useful or allowable in trailers (see Section 4.1 of [RFC7230]).
- Whether the header field ought to be preserved across redirects.
- Whether it introduces any additional security considerations, such as disclosure of privacy-related data.

8.3.2. Registrations

The "Message Headers" registry has been updated with the following permanent registrations:

Header Field Name	Protocol	Status	Reference
Accept	http	standard	Section 5.3.2
Accept-Charset	http	standard	Section 5.3.3
Accept-Encoding	http	standard	Section 5.3.4

Header Field Name	Protocol	Status	Reference
Accept-Language	http	standard	Section 5.3.5
Allow	http	standard	Section 7.4.1
Content-Encoding	http	standard	Section 3.1.2.2
Content-Language	http	standard	Section 3.1.3.2
Content-Location	http	standard	Section 3.1.4.2
Content-Type	http	standard	Section 3.1.1.5
Date	http	standard	Section 7.1.1.2
Expect	http	standard	Section 5.1.1
From	http	standard	Section 5.5.1
Location	http	standard	Section 7.1.2
Max-Forwards	http	standard	Section 5.1.2
MIME-Version	http	standard	Appendix A.1
Referer	http	standard	Section 5.5.2
Retry-After	http	standard	Section 7.1.3
Server	http	standard	Section 7.4.2
User-Agent	http	standard	Section 5.5.3
Vary	http	standard	Section 7.1.4

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

8.4. Content Coding Registry

The "HTTP Content Coding Registry" defines the namespace for content coding names (Section 4.2 of [RFC7230]). The content coding registry is maintained at <http://www.iana.org/assignments/http-parameters>.

8.4.1. Procedure

Content coding registrations MUST include the following fields:

- Name
- Description
- Pointer to specification text

Names of content codings MUST NOT overlap with names of transfer codings (Section 4 of [RFC7230]), unless the encoding transformation is identical (as is the case for the compression codings defined in Section 4.2 of [RFC7230]).

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

8.4.2. Registrations

The "HTTP Content Coding Registry" has been updated with the registrations below:

Name	Description	Reference
identity	Reserved (synonym for "no encoding" in Accept-Encoding)	Section 5.3.4

9. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns relevant to HTTP semantics and its use for transferring information over the Internet. Considerations related to message syntax, parsing, and routing are discussed in Section 9 of [RFC7230].

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

9.1. Attacks Based on File and Path Names

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

For example, UNIX, Microsoft Windows, and other operating systems use ".." as a path component to indicate a directory level above the current one, and they use specially named paths or file names to send data to system devices. Similar naming conventions might exist within other types of storage systems. Likewise, local storage systems have an annoying tendency to prefer user-friendliness over security when handling invalid or unexpected characters, recomposition of decomposed characters, and case-normalization of case-insensitive names.

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

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

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

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

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

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

9.3. 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.

9.4. Disclosure of Sensitive Information in URIs

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

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

Since the Referer header field tells a target site about the context that resulted in a request, it has the potential to reveal information about the user's immediate browsing history and any personal information that might be found in the referring resource's URI. Limitations on the Referer header field are described in Section 5.5.2 to address some of its security considerations.

9.5. 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 7.1.2), this might have the effect of disclosing one site's fragment to another site. If the first site uses personal information in fragments, it ought to ensure that redirects to other sites include a (possibly empty) fragment component in order to block that inheritance.

9.6. Disclosure of Product Information

The User-Agent (Section 5.5.3), Via (Section 5.7.1 of [RFC7230]), and Server (Section 7.4.2) 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.

9.7. Browser Fingerprinting

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

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

The User-Agent header field might contain enough information to uniquely identify a specific device, usually when combined with other characteristics, particularly if the user agent sends excessive details about the user's system or extensions. However, the source of unique information that is least expected by users is proactive negotiation (Section 5.3), including the Accept, Accept-Charset, Accept-Encoding, and Accept-Language header fields.

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

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

10. Acknowledgments

See Section 10 of [RFC7230].

11. References

11.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4647] Phillips, A., Ed. and M. Davis, Ed., "Matching of Language Tags", BCP 47, RFC 4647, September 2006.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, September 2009.

[RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, September 2011.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, June 2014.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, June 2014.

11.2. Informative References

[BCP13] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.

[BCP178] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, June 2012.

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

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

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

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC2049] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples", RFC 2049, November 1996.

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.

[RFC2295] Holtman, K. and A. Mutz, "Transparent Content Negotiation in HTTP", RFC 2295, March 1998.

[RFC2388] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 2388, August 1998.

[RFC2557] Palme, F., Hopmann, A., Shelness, N., and E. Stefferud, "MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2557, March 1999.

[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, June 1999.

[RFC2774] Frystyk, H., Leach, P., and S. Lawrence, "An HTTP Extension Framework", RFC 2774, February 2000.

[RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

[RFC2978] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, October 2000.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5322] Resnick, P., "Internet Message Format", RFC 5322, October 2008.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, March 2010.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

[RFC5987] Reschke, J., "Character Set and Language Encoding for Hypertext Transfer Protocol (HTTP) Header Field Parameters", RFC 5987, August 2010.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

[RFC6266] Reschke, J., "Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP)", RFC 6266, June 2011.

[RFC7238] Reschke, J., "The Hypertext Transfer Protocol (HTTP) Status Code 308 (Permanent Redirect)", RFC 7238, June 2014.

A. Differences between HTTP and MIME

HTTP/1.1 uses many of the constructs defined for the Internet Message Format [RFC5322] and the Multipurpose Internet Mail Extensions (MIME) [RFC2045] to allow a message body to be transmitted in an open variety of representations and with extensible header fields. However, RFC 2045 is focused only on email; applications of HTTP have many characteristics that differ from email; hence, HTTP has features that differ from MIME. These differences were carefully chosen to optimize performance over binary connections, to allow greater freedom in the use of new media types, to make date comparisons easier, and to acknowledge the practice of some early HTTP servers and clients.

This appendix describes specific areas where HTTP differs from MIME. Proxies and gateways to and from strict MIME environments need to be aware of these differences and provide the appropriate conversions where necessary.

A.1. MIME-Version

HTTP is not a MIME-compliant protocol. However, messages can include a single MIME-Version header field to indicate what version of the MIME protocol was used to construct the message. Use of the MIME-Version header field indicates that the message is in full conformance with the MIME protocol (as defined in [RFC2045]). Senders are responsible for ensuring full conformance (where possible) when exporting HTTP messages to strict MIME environments.

A.2. Conversion to Canonical Form

MIME requires that an Internet mail body part be converted to canonical form prior to being transferred, as described in Section 4 of [RFC2049]. Section 3.1.1.3 of this document describes the forms allowed for subtypes of the "text" media type when transmitted over HTTP. [RFC2046] requires that content with a type of "text" represent line breaks as CRLF and forbids the use of CR or LF outside of line break sequences. HTTP allows CRLF, bare CR, and bare LF to indicate a line break within text content.

A proxy or gateway from HTTP to a strict MIME environment ought to translate all line breaks within the text media types described in Section 3.1.1.3 of this document to the RFC 2049 canonical form of CRLF. Note, however, this might be complicated by the presence of a Content-Encoding and by the fact that HTTP allows the use of some charsets that do not use octets 13 and 10 to represent CR and LF, respectively.

Conversion will break any cryptographic checksums applied to the original content unless the original content is already in canonical form. Therefore, the canonical form is recommended for any content that uses such checksums in HTTP.

A.3. Conversion of Date Formats

HTTP/1.1 uses a restricted set of date formats (Section 7.1.1.1) to simplify the process of date comparison. Proxies and gateways from other protocols ought to ensure that any Date header field present in a message conforms to one of the HTTP/1.1 formats and rewrite the date if necessary.

A.4. Conversion of Content-Encoding

MIME does not include any concept equivalent to HTTP/1.1's Content-Encoding header field. Since this acts as a modifier on the media type, proxies and gateways from HTTP to MIME-compliant protocols ought to either change the value of the Content-Type header field or decode the representation before forwarding the message. (Some experimental applications of Content-Type for Internet mail have used a media-type parameter of ";conversions=<content-coding>" to perform a function equivalent to Content-Encoding. However, this parameter is not part of the MIME standards).

A.5. Conversion of Content-Transfer-Encoding

HTTP does not use the Content-Transfer-Encoding field of MIME. Proxies and gateways from MIME-compliant protocols to HTTP need to remove any Content-Transfer-Encoding prior to delivering the response message to an HTTP client.

Proxies and gateways from HTTP to MIME-compliant protocols are responsible for ensuring that the message is in the correct format and encoding for safe transport on that protocol, where "safe transport" is defined by the limitations of the protocol being used. Such a proxy or gateway ought to transform and label the data with an appropriate Content-Transfer-Encoding if doing so will improve the likelihood of safe transport over the destination protocol.

A.6. MHTML and Line Length Limitations

HTTP implementations that share code with MHTML [RFC2557] implementations need to be aware of MIME line length limitations. Since HTTP does not have this limitation, HTTP does not fold long lines. MHTML messages being transported by HTTP follow all conventions of MHTML, including line length limitations and folding, canonicalization, etc., since HTTP transfers message-bodies as payload and, aside from the "multipart/byteranges" type (Section A of [RFC7233]), does not interpret the content or any MIME header lines that might be contained therein.

B. Changes from RFC 2616

The primary changes in this revision have been editorial in nature: extracting the messaging syntax and partitioning HTTP semantics into separate documents for the core features, conditional requests, partial requests, caching, and authentication. The conformance language has been revised to clearly target requirements and the terminology has been improved to distinguish payload from representations and representations from resources.

A new requirement has been added that semantics embedded in a URI be disabled when those semantics are inconsistent with the request method, since this is a common cause of interoperability failure. (Section 2)

An algorithm has been added for determining if a payload is associated with a specific identifier. (Section 3.1.4.1)

The default charset of ISO-8859-1 for text media types has been removed; the default is now whatever the media type definition says. Likewise, special treatment of ISO-8859-1 has been removed from the Accept-Charset header field. (Section 3.1.1.3 and Section 5.3.3)

The definition of Content-Location has been changed to no longer affect the base URI for resolving relative URI references, due to poor implementation support and the undesirable effect of potentially breaking relative links in content-negotiated resources. (Section 3.1.4.2)

To be consistent with the method-neutral parsing algorithm of [RFC7230], the definition of GET has been relaxed so that requests can have a body, even though a body has no meaning for GET. (Section 4.3.1)

Servers are no longer required to handle all Content-* header fields and use of Content-Range has been explicitly banned in PUT requests. (Section 4.3.4)

Definition of the CONNECT method has been moved from [RFC2817] to this specification. (Section 4.3.6)

The OPTIONS and TRACE request methods have been defined as being safe. (Section 4.3.7 and Section 4.3.8)

The Expect header field's extension mechanism has been removed due to widely-deployed broken implementations. (Section 5.1.1)

The Max-Forwards header field has been restricted to the OPTIONS and TRACE methods; previously, extension methods could have used it as well. (Section 5.1.2)

The "about:blank" URI has been suggested as a value for the Referer header field when no referring URI is applicable, which distinguishes that case from others where the Referer field is not sent or has been removed. (Section 5.5.2)

The following status codes are now cacheable (that is, they can be stored and reused by a cache without explicit freshness information present): 204, 404, 405, 414, 501. (Section 6)

The 201 (Created) status description has been changed to allow for the possibility that more than one resource has been created. (Section 6.3.2)

The definition of 203 (Non-Authoritative Information) has been broadened to include cases of payload transformations as well. (Section 6.3.4)

The set of request methods that are safe to automatically redirect is no longer closed; user agents are able to make that determination based upon the request method semantics. The redirect status codes 301, 302, and 307 no longer have normative requirements on response payloads and user interaction. (Section 6.4)

The status codes 301 and 302 have been changed to allow user agents to rewrite the method from POST to GET. (Sections 6.4.2 and 6.4.3)

The description of the 303 (See Other) status code has been changed to allow it to be cached if explicit freshness information is given, and a specific definition has been added for a 303 response to GET. (Section 6.4.4)

The 305 (Use Proxy) status code has been deprecated due to security concerns regarding in-band configuration of a proxy. (Section 6.4.5)

The 400 (Bad Request) status code has been relaxed so that it isn't limited to syntax errors. (Section 6.5.1)

The 426 (Upgrade Required) status code has been incorporated from [RFC2817]. (Section 6.5.15)

The target of requirements on HTTP-date and the Date header field have been reduced to those systems generating the date, rather than all systems sending a date. (Section 7.1.1)

The syntax of the Location header field has been changed to allow all URI references, including relative references and fragments, along with some clarifications as to when use of fragments would not be appropriate. (Section 7.1.2)

Allow has been reclassified as a response header field, removing the option to specify it in a PUT request. Requirements relating to the content of Allow have been relaxed; correspondingly, clients are not required to always trust its value. (Section 7.4.1)

A Method Registry has been defined. (Section 8.1)

The Status Code Registry has been redefined by this specification; previously, it was defined in Section 7.1 of [RFC2817]. (Section 8.2)

Registration of content codings has been changed to require IETF Review. (Section 8.4)

The Content-Disposition header field has been removed since it is now defined by [RFC6266].

The Content-MD5 header field has been removed because it was inconsistently implemented with respect to partial responses.

C. Imported ABNF

The following core rules are included by reference, as defined in Section 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).

The rules below are defined in [RFC7230]:

```
BWS           = <BWS, see [RFC7230], Section 3.2.3>
OWS           = <OWS, see [RFC7230], Section 3.2.3>
RWS           = <RWS, see [RFC7230], Section 3.2.3>
URI-reference = <URI-reference, see [RFC7230], Section 2.7>
absolute-URI  = <absolute-URI, see [RFC7230], Section 2.7>
comment       = <comment, see [RFC7230], Section 3.2.6>
field-name    = <comment, see [RFC7230], Section 3.2>
partial-URI   = <partial-URI, see [RFC7230], Section 2.7>
quoted-string = <quoted-string, see [RFC7230], Section 3.2.6>
token         = <token, see [RFC7230], Section 3.2.6>
```

D. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [RFC7230].

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

BWS = <BWS, see [RFC7230], Section 3.2.3>

Content-Encoding = *( "," OWS ) content-coding *( OWS "," [ OWS
 content-coding ] )
Content-Language = *( "," OWS ) language-tag *( OWS "," [ OWS
 language-tag ] )
Content-Location = absolute-URI / partial-URI
Content-Type = media-type

Date = HTTP-date

Expect = "100-continue"

From = mailbox

GMT = %x47.4D.54 ; GMT

HTTP-date = IMF-fixdate / obs-date

IMF-fixdate = day-name "," SP date1 SP time-of-day SP GMT

Location = URI-reference

Max-Forwards = 1*DIGIT

OWS = <OWS, see [RFC7230], Section 3.2.3>

RWS = <RWS, see [RFC7230], Section 3.2.3>
Referer = absolute-URI / partial-URI
Retry-After = HTTP-date / delay-seconds

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

URI-reference = <URI-reference, see [RFC7230], Section 2.7>
User-Agent = product *( RWS ( product / comment ) )

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

absolute-URI = <absolute-URI, see [RFC7230], Section 2.7>
accept-ext = OWS ";" OWS token [ "=" ( token / quoted-string ) ]
accept-params = weight *accept-ext
asctime-date = day-name SP date3 SP time-of-day SP year

charset = token
codings = content-coding / "identity" / "*"
comment = <comment, see [RFC7230], Section 3.2.6>
content-coding = token

date1 = day SP month SP year
```

Index

1

100 Continue (status code) 35, **36**, 54
100-continue (expect value) **26**
101 Switching Protocols (status code) 35, **36**, 54
1xx Informational (status code class) **36**

2

200 OK (status code) 35, **37**, 54
201 Created (status code) 35, **37**, 54, 66
202 Accepted (status code) 35, **37**, 54
203 Non-Authoritative Information (status code) 35, **38**, 54, 66
204 No Content (status code) 35, **38**, 54
205 Reset Content (status code) 35, **38**, 54
2xx Successful (status code class) **37**

3

300 Multiple Choices (status code) 35, **39**, 42, 54
301 Moved Permanently (status code) 35, **40**, 54, 66
302 Found (status code) 35, **40**, 54, 66
303 See Other (status code) 35, **40**, 54, 66
305 Use Proxy (status code) 35, **41**, 54, 66
306 (Unused) (status code) **41**, 54
307 Temporary Redirect (status code) 35, **41**, 54
3xx Redirection (status code class) **39**, 66

4

400 Bad Request (status code) 35, **41**, 54, 67
402 Payment Required (status code) 35, **41**, 54
403 Forbidden (status code) 35, **41**, 54
404 Not Found (status code) 35, **41**, 54
405 Method Not Allowed (status code) 35, **42**, 54
406 Not Acceptable (status code) 35, **42**, 54
408 Request Timeout (status code) 35, **42**, 54
409 Conflict (status code) 35, **42**, 54
410 Gone (status code) 35, **42**, 54
411 Length Required (status code) 35, **43**, 54
413 Payload Too Large (status code) 35, **43**, 54
414 URI Too Long (status code) 35, **43**, 54
415 Unsupported Media Type (status code) 35, **43**, 54
417 Expectation Failed (status code) 35, **43**, 54
426 Upgrade Required (status code) 35, **43**, 54, 67
4xx Client Error (status code class) **41**

5

500 Internal Server Error (status code) 35, **44**, 54
501 Not Implemented (status code) 35, **44**, 54
502 Bad Gateway (status code) 35, **44**, 54
503 Service Unavailable (status code) 35, **44**, 54
504 Gateway Timeout (status code) 35, **44**, 54
505 HTTP Version Not Supported (status code) 35, **44**, 54
5xx Server Error (status code class) **44**

A

Accept header field 9, 28, **28**, 56
Accept-Charset header field 28, **30**, 56, 66
Accept-Encoding header field 11, 28, **30**, 56, 57
Accept-Language header field 12, 28, **31**, 56
Allow header field 18, 51, **51**, 56, 67

B

*BCP13*10, 62
*BCP178*55, 62
*BCP90*55, 55, 62

C

cacheable **19**
compress (content coding) **11**
conditional request **28**
CONNECT method 18, **23**, 53, 66
content coding **11**
content negotiation 7
Content-Encoding header field 9, 11, **11**, 56
Content-Language header field 9, **12**, 56
Content-Location header field 9, **13**, 21, 49, 56, 66
Content-Transfer-Encoding header field 64
Content-Type header field 9, 9, **10**, 56, 56

D

Date header field 15, 46, **48**, 56
deflate (content coding) **11**
DELETE method 18, **22**, 53

E

Expect header field 26, **26**, 36, 43, 56, 66

F

From header field 32, **32**, 56

G

GET method 9, 14, 15, 18, **20**, 53, 66
Grammar
  Accept **29**
  Accept-Charset **30**
  Accept-Encoding **31**
  accept-ext **29**
  Accept-Language **31**
  accept-params **29**
  Allow **51**
  asctime-date **48**
  charset **10**
  codings **31**
  content-coding **11**
  Content-Encoding **11**
  Content-Language **12**
  Content-Location **14**
  Content-Type **11**
  Date **48**
  date1 **47**
  day **47**
  day-name **47**
  day-name-l **47**
  delay-seconds **50**
  Expect **26**
  From **32**
  GMT **47**
  hour **47**
  HTTP-date **46**
  IMF-fixdate **47**

language-range **31**
language-tag **12**
Location **49**
Max-Forwards **27**
media-range **29**
media-type **9**
method **18**
minute **47**
month **47**
obs-date **47**
parameter **9**
product **34**
product-version **34**
qvalue **28**
Referer **33**
Retry-After **50**
rfc850-date **48**
second **47**
Server **52**
subtype **9**
time-of-day **47**
type **9**
User-Agent **34**
Vary **50**
weight **28**
year **47**
gzip (content coding) **11**

H
HEAD method 14, 18, **20**, 53

I
idempotent **19**

L
Location header field 21, 39, 46, **49**, 56, 59, 67

M
Max-Forwards header field 24, 25, 26, **27**, 56, 66
MIME-Version header field 56, **64**

O
OPTIONS method 18, **24**, 27, 53, 66
OWASP 58, 62

P
payload 15
POST method 14, 15, 18, **20**, 53
PUT method 14, 15, 18, **21**, 53, 66

R
Referer header field 32, **33**, 56, 59, 66
representation **9**
REST 9, 18, 62
Retry-After header field 44, 46, **49**, 56
RFC1945 39, 62
Section 9.3 39
RFC2045 10, 62, 64, 64
RFC2046 9, 10, 11, 62, 64
Section 4.5.1 11
Section 5.1.1 10
RFC2049 63, 64
Section 4 64
RFC2068 27, 39, 63
Section 10.3 39
RFC2119 7, 62
RFC2295 15, 63
RFC2388 10, 63
RFC2557 14, 63, 65
Section 4 14
RFC2616 32, 63
Section 14.4 32
RFC2774 53, 63
RFC2817 54, 63, 66, 67, 67
Section 7.1 54, 67
RFC2978 10, 63
RFC3986 33, 49, 49, 62
Section 4.2 49
Section 5 49
RFC4647 31, 32, 32, 32, 62
Section 2.1 31
Section 2.3 32
Section 3 32
Section 3.3.1 32
RFC5226 53, 54, 57, 63
Section 4.1 53, 54, 57
RFC5234 7, 55, 62, 68
Appendix B.1 68
RFC5246 23, 63
RFC5322 32, 32, 46, 47, 48, 48, 63, 64
Section 3.3 47, 48
Section 3.4 32, 32
Section 3.6.1 48
RFC5646 12, 12, 12, 62
Section 2.1 12
RFC5789 22, 63
RFC5905 46, 63
RFC5987 55, 63
RFC5988 40, 63
RFC6265 25, 32, 63
RFC6266 63, 67
RFC6365 7, 10, 62
RFC7230 7, 7, 7, 8, 8, 8, 11, 11, 11, 12, 13, 14, 15, 15, 15, 23, 24, 25, 25, 26, 26, 27, 34, 36, 38, 42, 43, 43, 43, 45, 52, 53, 55, 55, 56, 56, 56, 56, 56, 57, 57, 57, 58, 59, 61, 62, 66, 68, 68, 68, 68, 68, 68, 68, 68, 68, 68, 68, 69
Section 1.2 69
Section 2.5 7
Section 2.6 45
Section 2.7 8, 68, 68, 68
Section 3.2 34, 52, 55, 56, 68
Section 3.2.3 68, 68, 68
Section 3.2.4 56
Section 3.2.6 56, 68, 68, 68
Section 3.3.1 12, 15
Section 3.3 53
Section 3.3.2 15, 43
Section 4 57
Section 4.1 56
Section 4.2.1 11
Section 4.2 57, 57

Section 4.2.2 11
Section 4.2.3 11
Section 4.3 26
Section 4.4 15
Section 5.3 8, 23, 24, 43
Section 5.4 26
Section 5.5 8, 13, 14
Section 5.7.1 25, 59
Section 5.7.2 38
Section 6.1 42, 56
Section 6.6 27
Section 6.7 36, 43
Section 7 7, 55
Section 8.3.1 25
Section 9 58
Section 10 61
RFC7232 9, 18, 28, 28, 28, 28, 28, 28, 35, 35, 35, 51, 51, 51, 62
Section 2.2 51
Section 2.3 51
Section 3.1 28
Section 3.2 28
Section 3.3 28
Section 3.4 28
Section 4 35
Section 4.1 35
Section 4.2 35
Section 5 28
RFC7233 10, 15, 20, 22, 26, 28, 35, 35, 35, 51, 53, 62, 65
Section 2.3 51
Section 3.1 26
Section 3.2 28
Section 4 35
Section 4.1 35
Section 4.2 15, 22
Section 4.4 35
Appendix A 65
RFC7234 20, 20, 20, 20, 21, 22, 23, 26, 26, 35, 37, 38, 38, 38, 39, 40, 42, 42, 43, 43, 44, 46, 46, 46, 46, 50, 50, 54, 62
Section 4.1 50
Section 4.2.1 21
Section 4.2.2 37, 38, 38, 39, 40, 42, 42, 43, 43, 44
Section 4.3.5 20
Section 4.4 22, 23
Section 5.1 46
Section 5.2 20, 26, 46, 50
Section 5.3 46
Section 5.4 26
Section 5.5 38, 46
RFC7235 25, 32, 32, 32, 35, 35, 35, 50, 51, 51, 62
Section 3 35
Section 3.1 35
Section 3.2 35
Section 4.1 51
Section 4.2 32, 50
Section 4.3 51
Section 4.4 32
RFC7238 41, 63

S

safe **19**
selected representation **9**, 50
Server header field 51, **51**, 56, 59
Status Codes Classes
1xx Informational **36**
2xx Successful **37**
3xx Redirection **39**, 66
4xx Client Error **41**
5xx Server Error **44**

T

TRACE method 18, **24**, 27, 53, 66

U

User-Agent header field 32, **33**, 52, 56, 59

V

Vary header field 16, 46, **50**, 56, 56

X

x-compress (content coding) **11**
x-gzip (content coding) **11**

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA
EMail: fielding@gbiv.com
URI: http://roy.gbiv.com/

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany
EMail: julian.reschke@greenbytes.de
URI: http://greenbytes.de/tech/webdav/