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

R. Fielding, Ed.
Adobe
J. Reschke, Ed.
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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
1 of 101

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
2 of 101

Table of Contents

1. Introduction 6
1.1. Conformance and Error Handling 6
1.2. Syntax Notation 6
2. Resources 7
3. Representations 7
3.1. Representation Metadata 8
3.1.1. Processing Representation Data 8
3.1.2. Encoding for Compression or Integrity 11
3.1.3. Audience Language 13
3.1.4. Identification 14
3.2. Representation Data 17
3.3. Payload Semantics 17
3.4. Content Negotiation 18
3.4.1. Proactive Negotiation 19
3.4.2. Reactive Negotiation 20
4. Request Methods 21
4.1. Overview 21
4.2. Common Method Properties 22
4.2.1. Safe Methods 22
4.2.2. Idempotent Methods 23
4.2.3. Cacheable Methods 24
4.3. Method Definitions 24
4.3.1. GET 24
4.3.2. HEAD 25
4.3.3. POST 25
4.3.4. PUT 26
4.3.5. DELETE 29
4.3.6. CONNECT 30
4.3.7. OPTIONS 31
4.3.8. TRACE 32
5. Request Header Fields 33
5.1. Controls 33
5.1.1. Expect 34
5.1.2. Max-Forwards 36
5.2. Conditionals 36
5.3. Content Negotiation 37
5.3.1. Quality Values 37
5.3.2. Accept 38
5.3.3. Accept-Charset 40
5.3.4. Accept-Encoding 41
5.3.5. Accept-Language 42
5.4. Authentication Credentials 44
5.5. Request Context 44
5.5.1. From 44
5.5.2. Referer 45
5.5.3. User-Agent 46

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
3 of 101

6. Response Status Codes 47
6.1. Overview of Status Codes 48
6.2. Informational 1xx 50
6.2.1. 100 Continue 50
6.2.2. 101 Switching Protocols 50
6.3. Successful 2xx 51
6.3.1. 200 OK 51
6.3.2. 201 Created 52
6.3.3. 202 Accepted 52
6.3.4. 203 Non-Authoritative Information 52
6.3.5. 204 No Content 53
6.3.6. 205 Reset Content 53
6.4. Redirection 3xx 54
6.4.1. 300 Multiple Choices 55
6.4.2. 301 Moved Permanently 56
6.4.3. 302 Found 56
6.4.4. 303 See Other 57
6.4.5. 305 Use Proxy 58
6.4.6. 306 (Unused) 58
6.4.7. 307 Temporary Redirect 58
6.5. Client Error 4xx 58
6.5.1. 400 Bad Request 58
6.5.2. 402 Payment Required 59
6.5.3. 403 Forbidden 59
6.5.4. 404 Not Found 59
6.5.5. 405 Method Not Allowed 59
6.5.6. 406 Not Acceptable 60
6.5.7. 408 Request Timeout 60
6.5.8. 409 Conflict 60
6.5.9. 410 Gone 60
6.5.10. 411 Length Required 61
6.5.11. 413 Payload Too Large 61
6.5.12. 414 URI Too Long 61
6.5.13. 415 Unsupported Media Type 62
6.5.14. 417 Expectation Failed 62
6.5.15. 426 Upgrade Required 62
6.6. Server Error 5xx 62
6.6.1. 500 Internal Server Error 63
6.6.2. 501 Not Implemented 63
6.6.3. 502 Bad Gateway 63
6.6.4. 503 Service Unavailable 63
6.6.5. 504 Gateway Timeout 63
6.6.6. 505 HTTP Version Not Supported 64
7. Response Header Fields 64
7.1. Control Data 64
ed 7.1.1. Origination Date 65
7.1.2. Location 68
7.1.3. Retry-After 69

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
4 of 101

7.1.4. Vary 70
7.2. Validator Header Fields 71
7.3. Authentication Challenges 72
7.4. Response Context 72
7.4.1. Allow 72
7.4.2. Server 73
8. IANA Considerations 73
8.1. Method Registry 73
8.1.1. Procedure 74
8.1.2. Considerations for New Methods 74
8.1.3. Registrations 75
8.2. Status Code Registry 75
8.2.1. Procedure 75
8.2.2. Considerations for New Status Codes 76
8.2.3. Registrations 76
8.3. Header Field Registry 77
8.3.1. Considerations for New Header Fields 78
8.3.2. Registrations 80
8.4. Content Coding Registry 81
8.4.1. Procedure 81
8.4.2. Registrations 81
9. Security Considerations 81
9.1. Attacks Based on File and Path Names 82
9.2. Attacks Based on Command, Code, or Query Injection 82
9.3. Disclosure of Personal Information 83
9.4. Disclosure of Sensitive Information in URIs 83
9.5. Disclosure of Fragment after Redirects 84
9.6. Disclosure of Product Information 84
9.7. Browser Fingerprinting 84
10. Acknowledgments 85
11. References 85
11.1. Normative References 85
11.2. Informative References 86
Appendix A. Differences between HTTP and MIME 89
A.1. MIME-Version 89
A.2. Conversion to Canonical Form 89
A.3. Conversion of Date Formats 90
A.4. Conversion of Content-Encoding 90
A.5. Conversion of Content-Transfer-Encoding 90
A.6. MHTML and Line Length Limitations 90
Appendix B. Changes from RFC 2616 91
Appendix C. Imported ABNF 93
Appendix D. Collected ABNF 94
Index 97

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
5 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
6 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
7 of 101

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 )
```


Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
8 of 101

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).

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
9 of 101

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
```


Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
10 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
11 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
12 of 101

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
```


Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
13 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
14 of 101

- 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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
15 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
16 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
17 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
18 of 101

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,

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
19 of 101

- 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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
20 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
21 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
22 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
23 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
24 of 101

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;

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
25 of 101

- 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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
26 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
27 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
28 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
29 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
30 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
31 of 101

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).

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
32 of 101

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]

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
33 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
34 of 101

- 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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
35 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
36 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
37 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
38 of 101

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,

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
39 of 101

```
  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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
40 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
41 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
42 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
43 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
44 of 101

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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
45 of 101

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
```


Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
46 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
47 of 101

- 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.

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
48 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
49 of 101

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

Bright Data Ltd. v. Oxylabs, UAB
IPR2023-00840, EX. 1005
50 of 101