**[2.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.1). Client/Server Messaging**

HTTP is a stateless request/response protocol that operates by

exchanging messages ([Section 3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3)) across a reliable transport- or

session-layer "connection" ([Section 6](https://www.rfc-editor.org/rfc/rfc7230" \l "section-6)). An HTTP "client" is a

program that establishes a connection to a server for the purpose of

sending one or more HTTP requests. An HTTP "server" is a program

that accepts connections in order to service HTTP requests by sending

HTTP responses.

The terms "client" and "server" refer only to the roles that these

programs perform for a particular connection. The same program might

act as a client on some connections and a server on others. The term

"user agent" refers to any of the various client programs that

initiate a request, including (but not limited to) browsers, spiders

(web-based robots), command-line tools, custom applications, and

mobile apps. The term "origin server" refers to the program that can

originate authoritative responses for a given target resource. The

terms "sender" and "recipient" refer to any implementation that sends

or receives a given message, respectively.

HTTP relies upon the Uniform Resource Identifier (URI) standard

[[RFC3986](https://www.rfc-editor.org/rfc/rfc3986" \o "\"Uniform Resource Identifier (URI): Generic Syntax\")] to indicate the target resource ([Section 5.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-5.1)) and

relationships between resources. Messages are passed in a format

similar to that used by Internet mail [[RFC5322](https://www.rfc-editor.org/rfc/rfc5322" \o "\"Internet Message Format\")] and the Multipurpose

Internet Mail Extensions (MIME) [[RFC2045](https://www.rfc-editor.org/rfc/rfc2045" \o "\"Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies\")] (see [Appendix A of](https://www.rfc-editor.org/rfc/rfc7231" \l "appendix-A)

[[RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "appendix-A) for the differences between HTTP and MIME messages).

Most HTTP communication consists of a retrieval request (GET) for a

representation of some resource identified by a URI. In the simplest

case, this might be accomplished via a single bidirectional

connection (===) between the user agent (UA) and the origin

server (O).

request >

UA ======================================= O

< response

A client sends an HTTP request to a server in the form of a request

message, beginning with a request-line that includes a method, URI,

and protocol version ([Section 3.1.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1.1)), followed by header fields

containing request modifiers, client information, and representation

metadata ([Section 3.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2)), an empty line to indicate the end of the

header section, and finally a message body containing the payload

body (if any, [Section 3.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3)).

A server responds to a client's request by sending one or more HTTP

response messages, each beginning with a status line that includes

the protocol version, a success or error code, and textual reason

phrase ([Section 3.1.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1.2)), possibly followed by header fields containing

server information, resource metadata, and representation metadata

([Section 3.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2)), an empty line to indicate the end of the header

section, and finally a message body containing the payload body (if

any, [Section 3.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3)).

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

as defined in [Section 6.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-6.3).

The following example illustrates a typical message exchange for a

GET request ([Section 4.3.1 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.1)) on the URI

"http://www.example.com/hello.txt":

Client request:

GET /hello.txt HTTP/1.1

User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3

Host: www.example.com

Accept-Language: en, mi

Server response:

HTTP/1.1 200 OK

Date: Mon, 27 Jul 2009 12:28:53 GMT

Server: Apache

Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT

ETag: "34aa387-d-1568eb00"

Accept-Ranges: bytes

Content-Length: 51

Vary: Accept-Encoding

Content-Type: text/plain

Hello World! My payload includes a trailing CRLF.

**[2.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.2). Implementation Diversity**

When considering the design of HTTP, it is easy to fall into a trap

of thinking that all user agents are general-purpose browsers and all

origin servers are large public websites. That is not the case in

practice. Common HTTP user agents include household appliances,

stereos, scales, firmware update scripts, command-line programs,

mobile apps, and communication devices in a multitude of shapes and

sizes. Likewise, common HTTP origin servers include home automation

units, configurable networking components, office machines,

autonomous robots, news feeds, traffic cameras, ad selectors, and

video-delivery platforms.

The term "user agent" does not imply that there is a human user

directly interacting with the software agent at the time of a

request. In many cases, a user agent is installed or configured to

run in the background and save its results for later inspection (or

save only a subset of those results that might be interesting or

erroneous). Spiders, for example, are typically given a start URI

and configured to follow certain behavior while crawling the Web as a

hypertext graph.

The implementation diversity of HTTP means that not all user agents

can make interactive suggestions to their user or provide adequate

warning for security or privacy concerns. In the few cases where

this specification requires reporting of errors to the user, it is

acceptable for such reporting to only be observable in an error

console or log file. Likewise, requirements that an automated action

be confirmed by the user before proceeding might be met via advance

configuration choices, run-time options, or simple avoidance of the

unsafe action; confirmation does not imply any specific user

interface or interruption of normal processing if the user has

already made that choice.

**[2.7](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.7). Uniform Resource Identifiers**

Uniform Resource Identifiers (URIs) [[RFC3986](https://www.rfc-editor.org/rfc/rfc3986" \o "\"Uniform Resource Identifier (URI): Generic Syntax\")] are used throughout

HTTP as the means for identifying resources ([Section 2 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-2)).

URI references are used to target requests, indicate redirects, and

define relationships.

The definitions of "URI-reference", "absolute-URI", "relative-part",

"scheme", "authority", "port", "host", "path-abempty", "segment",

"query", and "fragment" are adopted from the URI generic syntax. An

"absolute-path" rule is defined for protocol elements that can

contain a non-empty path component. (This rule differs slightly from

the path-abempty rule of [RFC 3986](https://www.rfc-editor.org/rfc/rfc3986), which allows for an empty path to

be used in references, and path-absolute rule, which does not allow

paths that begin with "//".) A "partial-URI" rule is defined for

protocol elements that can contain a relative URI but not a fragment

component.

URI-reference = <URI-reference, see [[RFC3986], Section 4.1](https://www.rfc-editor.org/rfc/rfc3986" \l "section-4.1)>

absolute-URI = <absolute-URI, see [[RFC3986], Section 4.3](https://www.rfc-editor.org/rfc/rfc3986" \l "section-4.3)>

relative-part = <relative-part, see [[RFC3986], Section 4.2](https://www.rfc-editor.org/rfc/rfc3986" \l "section-4.2)>

scheme = <scheme, see [[RFC3986], Section 3.1](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.1)>

authority = <authority, see [[RFC3986], Section 3.2](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.2)>

uri-host = <host, see [[RFC3986], Section 3.2.2](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.2.2)>

port = <port, see [[RFC3986], Section 3.2.3](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.2.3)>

path-abempty = <path-abempty, see [[RFC3986], Section 3.3](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.3)>

segment = <segment, see [[RFC3986], Section 3.3](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.3)>

query = <query, see [[RFC3986], Section 3.4](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.4)>

fragment = <fragment, see [[RFC3986], Section 3.5](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.5)>

absolute-path = 1\*( "/" segment )

partial-URI = relative-part [ "?" query ]

Each protocol element in HTTP that allows a URI reference will

indicate in its ABNF production whether the element allows any form

of reference (URI-reference), only a URI in absolute form

(absolute-URI), only the path and optional query components, or some

combination of the above. Unless otherwise indicated, URI references

are parsed relative to the effective request URI ([Section 5.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-5.5)).

**[2.7.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.7.1). http URI Scheme**

The "http" URI scheme is hereby defined for the purpose of minting

identifiers according to their association with the hierarchical

namespace governed by a potential HTTP origin server listening for

TCP ([[RFC0793](https://www.rfc-editor.org/rfc/rfc0793" \o "\"Transmission Control Protocol\")]) connections on a given port.

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

[ "#" fragment ]

The origin server for an "http" URI is identified by the authority

component, which includes a host identifier and optional TCP port

([[RFC3986], Section 3.2.2](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.2.2)). The hierarchical path component and

optional query component serve as an identifier for a potential

target resource within that origin server's name space. The optional

fragment component allows for indirect identification of a secondary

resource, independent of the URI scheme, as defined in [Section 3.5 of](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.5)

[[RFC3986]](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.5).

A sender MUST NOT generate an "http" URI with an empty host

identifier. A recipient that processes such a URI reference MUST

reject it as invalid.

If the host identifier is provided as an IP address, the origin

server is the listener (if any) on the indicated TCP port at that IP

address. If host is a registered name, the registered name is an

indirect identifier for use with a name resolution service, such as

DNS, to find an address for that origin server. If the port

subcomponent is empty or not given, TCP port 80 (the reserved port

for WWW services) is the default.

Note that the presence of a URI with a given authority component does

not imply that there is always an HTTP server listening for

connections on that host and port. Anyone can mint a URI. What the

authority component determines is who has the right to respond

authoritatively to requests that target the identified resource. The

delegated nature of registered names and IP addresses creates a

federated namespace, based on control over the indicated host and

port, whether or not an HTTP server is present. See [Section 9.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.1) for

security considerations related to establishing authority.

When an "http" URI is used within a context that calls for access to

the indicated resource, a client MAY attempt access by resolving the

host to an IP address, establishing a TCP connection to that address

on the indicated port, and sending an HTTP request message

([Section 3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3)) containing the URI's identifying data ([Section 5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-5)) to the

server. If the server responds to that request with a non-interim

HTTP response message, as described in [Section 6 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-6), then

that response is considered an authoritative answer to the client's

request.

Although HTTP is independent of the transport protocol, the "http"

scheme is specific to TCP-based services because the name delegation

process depends on TCP for establishing authority. An HTTP service

based on some other underlying connection protocol would presumably

be identified using a different URI scheme, just as the "https"

scheme (below) is used for resources that require an end-to-end

secured connection. Other protocols might also be used to provide

access to "http" identified resources -- it is only the authoritative

interface that is specific to TCP.

The URI generic syntax for authority also includes a deprecated

userinfo subcomponent ([[RFC3986], Section 3.2.1](https://www.rfc-editor.org/rfc/rfc3986" \l "section-3.2.1)) for including user

authentication information in the URI. Some implementations make use

of the userinfo component for internal configuration of

authentication information, such as within command invocation

options, configuration files, or bookmark lists, even though such

usage might expose a user identifier or password. A sender MUST NOT

generate the userinfo subcomponent (and its "@" delimiter) when an

"http" URI reference is generated within a message as a request

target or header field value. Before making use of an "http" URI

reference received from an untrusted source, a recipient SHOULD parse

for userinfo and treat its presence as an error; it is likely being

used to obscure the authority for the sake of phishing attacks.

**[2.7.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.7.2). https URI Scheme**

The "https" URI scheme is hereby defined for the purpose of minting

identifiers according to their association with the hierarchical

namespace governed by a potential HTTP origin server listening to a

given TCP port for TLS-secured connections ([[RFC5246](https://www.rfc-editor.org/rfc/rfc5246" \o "\"The Transport Layer Security (TLS) Protocol Version 1.2\")]).

All of the requirements listed above for the "http" scheme are also

requirements for the "https" scheme, except that TCP port 443 is the

default if the port subcomponent is empty or not given, and the user

agent MUST ensure that its connection to the origin server is secured

through the use of strong encryption, end-to-end, prior to sending

the first HTTP request.

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

[ "#" fragment ]

Note that the "https" URI scheme depends on both TLS and TCP for

establishing authority. Resources made available via the "https"

scheme have no shared identity with the "http" scheme even if their

resource identifiers indicate the same authority (the same host

listening to the same TCP port). They are distinct namespaces and

are considered to be distinct origin servers. However, an extension

to HTTP that is defined to apply to entire host domains, such as the

Cookie protocol [[RFC6265](https://www.rfc-editor.org/rfc/rfc6265" \o "\"HTTP State Management Mechanism\")], can allow information set by one service

to impact communication with other services within a matching group

of host domains.

The process for authoritative access to an "https" identified

resource is defined in [[RFC2818](https://www.rfc-editor.org/rfc/rfc2818" \o "\"HTTP Over TLS\")].

**[2.7.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.7.3). http and https URI Normalization and Comparison**

Since the "http" and "https" schemes conform to the URI generic

syntax, such URIs are normalized and compared according to the

algorithm defined in [Section 6 of [RFC3986]](https://www.rfc-editor.org/rfc/rfc3986" \l "section-6), using the defaults

described above for each scheme.

If the port is equal to the default port for a scheme, the normal

form is to omit the port subcomponent. When not being used in

absolute form as the request target of an OPTIONS request, an empty

path component is equivalent to an absolute path of "/", so the

normal form is to provide a path of "/" instead. The scheme and host

are case-insensitive and normally provided in lowercase; all other

components are compared in a case-sensitive manner. Characters other

than those in the "reserved" set are equivalent to their

percent-encoded octets: the normal form is to not encode them (see

Sections [2.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.1) and [2.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.2) of [[RFC3986](https://www.rfc-editor.org/rfc/rfc3986" \o "\"Uniform Resource Identifier (URI): Generic Syntax\")]).

For example, the following three URIs are equivalent:

[http://example.com:80/~smith/home.html](http://example.com/~smith/home.html)

http://EXAMPLE.com/%7Esmith/home.html

[http://EXAMPLE.com:/%7esmith/home.html](http://example.com/~smith/home.html)

**[3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3). Message Format**

All HTTP/1.1 messages consist of a start-line followed by a sequence

of octets in a format similar to the Internet Message Format

[[RFC5322](https://www.rfc-editor.org/rfc/rfc5322" \o "\"Internet Message Format\")]: zero or more header fields (collectively referred to as

the "headers" or the "header section"), an empty line indicating the

end of the header section, and an optional message body.

HTTP-message = start-line

\*( header-field CRLF )

CRLF

[ message-body ]

The normal procedure for parsing an HTTP message is to read the

start-line into a structure, read each header field into a hash table

by field name until the empty line, and then use the parsed data to

determine if a message body is expected. If a message body has been

indicated, then it is read as a stream until an amount of octets

equal to the message body length is read or the connection is closed.

A recipient MUST parse an HTTP message as a sequence of octets in an

encoding that is a superset of US-ASCII [[USASCII](https://www.rfc-editor.org/rfc/rfc7230" \l "ref-USASCII" \o "\"Coded Character Set -- 7-bit American Standard Code for Information Interchange\")]. Parsing an HTTP

message as a stream of Unicode characters, without regard for the

specific encoding, creates security vulnerabilities due to the

varying ways that string processing libraries handle invalid

multibyte character sequences that contain the octet LF (%x0A).

String-based parsers can only be safely used within protocol elements

after the element has been extracted from the message, such as within

a header field-value after message parsing has delineated the

individual fields.

An HTTP message can be parsed as a stream for incremental processing

or forwarding downstream. However, recipients cannot rely on

incremental delivery of partial messages, since some implementations

will buffer or delay message forwarding for the sake of network

efficiency, security checks, or payload transformations.

A sender MUST NOT send whitespace between the start-line and the

first header field. A recipient that receives whitespace between the

start-line and the first header field MUST either reject the message

as invalid or consume each whitespace-preceded line without further

processing of it (i.e., ignore the entire line, along with any

subsequent lines preceded by whitespace, until a properly formed

header field is received or the header section is terminated).

The presence of such whitespace in a request might be an attempt to

trick a server into ignoring that field or processing the line after

it as a new request, either of which might result in a security

vulnerability if other implementations within the request chain

interpret the same message differently. Likewise, the presence of

such whitespace in a response might be ignored by some clients or

cause others to cease parsing.

**[3.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1). Start Line**

An HTTP message can be either a request from client to server or a

response from server to client. Syntactically, the two types of

message differ only in the start-line, which is either a request-line

(for requests) or a status-line (for responses), and in the algorithm

for determining the length of the message body ([Section 3.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3)).

In theory, a client could receive requests and a server could receive

responses, distinguishing them by their different start-line formats,

but, in practice, servers are implemented to only expect a request (a

response is interpreted as an unknown or invalid request method) and

clients are implemented to only expect a response.

start-line = request-line / status-line

**[3.1.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1.1). Request Line**

A request-line begins with a method token, followed by a single space

(SP), the request-target, another single space (SP), the protocol

version, and ends with CRLF.

request-line = method SP request-target SP HTTP-version CRLF

The method token indicates the request method to be performed on the

target resource. The request method is case-sensitive.

method = token

The request methods defined by this specification can be found in

[Section 4 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4), along with information regarding the HTTP

method registry and considerations for defining new methods.

The request-target identifies the target resource upon which to apply

the request, as defined in [Section 5.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-5.3).

Recipients typically parse the request-line into its component parts

by splitting on whitespace (see [Section 3.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.5)), since no whitespace is

allowed in the three components. Unfortunately, some user agents

fail to properly encode or exclude whitespace found in hypertext

references, resulting in those disallowed characters being sent in a

request-target.

Recipients of an invalid request-line SHOULD respond with either a

400 (Bad Request) error or a 301 (Moved Permanently) redirect with

the request-target properly encoded. A recipient SHOULD NOT attempt

to autocorrect and then process the request without a redirect, since

the invalid request-line might be deliberately crafted to bypass

security filters along the request chain.

HTTP does not place a predefined limit on the length of a

request-line, as described in [Section 2.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.5). A server that receives a

method longer than any that it implements SHOULD respond with a 501

(Not Implemented) status code. A server that receives a

request-target longer than any URI it wishes to parse MUST respond

with a 414 (URI Too Long) status code (see [Section 6.5.12 of](https://www.rfc-editor.org/rfc/rfc7231" \l "section-6.5.12)

[[RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-6.5.12)).

Various ad hoc limitations on request-line length are found in

practice. It is RECOMMENDED that all HTTP senders and recipients

support, at a minimum, request-line lengths of 8000 octets.

**[3.1.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1.2). Status Line**

The first line of a response message is the status-line, consisting

of the protocol version, a space (SP), the status code, another

space, a possibly empty textual phrase describing the status code,

and ending with CRLF.

status-line = HTTP-version SP status-code SP reason-phrase CRLF

The status-code element is a 3-digit integer code describing the

result of the server's attempt to understand and satisfy the client's

corresponding request. The rest of the response message is to be

interpreted in light of the semantics defined for that status code.

See [Section 6 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-6) for information about the semantics of

status codes, including the classes of status code (indicated by the

first digit), the status codes defined by this specification,

considerations for the definition of new status codes, and the IANA

registry.

status-code = 3DIGIT

The reason-phrase element exists for the sole purpose of providing a

textual description associated with the numeric status code, mostly

out of deference to earlier Internet application protocols that were

more frequently used with interactive text clients. A client SHOULD

ignore the reason-phrase content.

reason-phrase = \*( HTAB / SP / VCHAR / obs-text )

**[3.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2). Header Fields**

Each header field consists of a case-insensitive field name followed

by a colon (":"), optional leading whitespace, the field value, and

optional trailing whitespace.

header-field = field-name ":" OWS field-value OWS

field-name = token

field-value = \*( field-content / obs-fold )

field-content = field-vchar [ 1\*( SP / HTAB ) field-vchar ]

field-vchar = VCHAR / obs-text

obs-fold = CRLF 1\*( SP / HTAB )

; obsolete line folding

; see [Section 3.2.4](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.4)

The field-name token labels the corresponding field-value as having

the semantics defined by that header field. For example, the Date

header field is defined in [Section 7.1.1.2 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-7.1.1.2) as containing

the origination timestamp for the message in which it appears.

**[3.2.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.1). Field Extensibility**

Header fields are fully extensible: there is no limit on the

introduction of new field names, each presumably defining new

semantics, nor on the number of header fields used in a given

message. Existing fields are defined in each part of this

specification and in many other specifications outside this document

set.

New header fields can be defined such that, when they are understood

by a recipient, they might override or enhance the interpretation of

previously defined header fields, define preconditions on request

evaluation, or refine the meaning of responses.

A proxy MUST forward unrecognized header fields unless the field-name

is listed in the Connection header field ([Section 6.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-6.1)) or the proxy

is specifically configured to block, or otherwise transform, such

fields. Other recipients SHOULD ignore unrecognized header fields.

These requirements allow HTTP's functionality to be enhanced without

requiring prior update of deployed intermediaries.

All defined header fields ought to be registered with IANA in the

"Message Headers" registry, as described in [Section 8.3 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-8.3).

**[3.2.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.2). Field Order**

The order in which header fields with differing field names are

received is not significant. However, it is good practice to send

header fields that contain control data first, such as Host on

requests and Date on responses, so that implementations can decide

when not to handle a message as early as possible. A server MUST NOT

apply a request to the target resource until the entire request

header section is received, since later header fields might include

conditionals, authentication credentials, or deliberately misleading

duplicate header fields that would impact request processing.

A sender MUST NOT generate multiple header fields with the same field

name in a message unless either the entire field value for that

header field is defined as a comma-separated list [i.e., #(values)]

or the header field is a well-known exception (as noted below).

A recipient MAY combine multiple header fields with the same field

name into one "field-name: field-value" pair, without changing the

semantics of the message, by appending each subsequent field value to

the combined field value in order, separated by a comma. The order

in which header fields with the same field name are received is

therefore significant to the interpretation of the combined field

value; a proxy MUST NOT change the order of these field values when

forwarding a message.

Note: In practice, the "Set-Cookie" header field ([[RFC6265](https://www.rfc-editor.org/rfc/rfc6265" \o "\"HTTP State Management Mechanism\")]) often

appears multiple times in a response message and does not use the

list syntax, violating the above requirements on multiple header

fields with the same name. Since it cannot be combined into a

single field-value, recipients ought to handle "Set-Cookie" as a

special case while processing header fields. (See [Appendix A.2.3](https://www.rfc-editor.org/rfc/rfc7230" \l "appendix-A.2.3)

of [[Kri2001](https://www.rfc-editor.org/rfc/rfc7230" \l "ref-Kri2001" \o "\"HTTP Cookies: Standards, Privacy, and Politics\")] for details.)

**[3.2.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.3). Whitespace**

This specification uses three rules to denote the use of linear

whitespace: OWS (optional whitespace), RWS (required whitespace), and

BWS ("bad" whitespace).

The OWS rule is used where zero or more linear whitespace octets

might appear. For protocol elements where optional whitespace is

preferred to improve readability, a sender SHOULD generate the

optional whitespace as a single SP; otherwise, a sender SHOULD NOT

generate optional whitespace except as needed to white out invalid or

unwanted protocol elements during in-place message filtering.

The RWS rule is used when at least one linear whitespace octet is

required to separate field tokens. A sender SHOULD generate RWS as a

single SP.

The BWS rule is used where the grammar allows optional whitespace

only for historical reasons. A sender MUST NOT generate BWS in

messages. A recipient MUST parse for such bad whitespace and remove

it before interpreting the protocol element.

OWS = \*( SP / HTAB )

; optional whitespace

RWS = 1\*( SP / HTAB )

; required whitespace

BWS = OWS

; "bad" whitespace

**[3.2.4](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.4). Field Parsing**

Messages are parsed using a generic algorithm, independent of the

individual header field names. The contents within a given field

value are not parsed until a later stage of message interpretation

(usually after the message's entire header section has been

processed). Consequently, this specification does not use ABNF rules

to define each "Field-Name: Field Value" pair, as was done in

previous editions. Instead, this specification uses ABNF rules that

are named according to each registered field name, wherein the rule

defines the valid grammar for that field's corresponding field values

(i.e., after the field-value has been extracted from the header

section by a generic field parser).

No whitespace is allowed between the header field-name and colon. In

the past, differences in the handling of such whitespace have led to

security vulnerabilities in request routing and response handling. A

server MUST reject any received request message that contains

whitespace between a header field-name and colon with a response code

of 400 (Bad Request). A proxy MUST remove any such whitespace from a

response message before forwarding the message downstream.

A field value might be preceded and/or followed by optional

whitespace (OWS); a single SP preceding the field-value is preferred

for consistent readability by humans. The field value does not

include any leading or trailing whitespace: OWS occurring before the

first non-whitespace octet of the field value or after the last

non-whitespace octet of the field value ought to be excluded by

parsers when extracting the field value from a header field.

Historically, HTTP header field values could be extended over

multiple lines by preceding each extra line with at least one space

or horizontal tab (obs-fold). This specification deprecates such

line folding except within the message/http media type

([Section 8.3.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-8.3.1)). A sender MUST NOT generate a message that includes

line folding (i.e., that has any field-value that contains a match to

the obs-fold rule) unless the message is intended for packaging

within the message/http media type.

A server that receives an obs-fold in a request message that is not

within a message/http container MUST either reject the message by

sending a 400 (Bad Request), preferably with a representation

explaining that obsolete line folding is unacceptable, or replace

each received obs-fold with one or more SP octets prior to

interpreting the field value or forwarding the message downstream.

A proxy or gateway that receives an obs-fold in a response message

that is not within a message/http container MUST either discard the

message and replace it with a 502 (Bad Gateway) response, preferably

with a representation explaining that unacceptable line folding was

received, or replace each received obs-fold with one or more SP

octets prior to interpreting the field value or forwarding the

message downstream.

A user agent that receives an obs-fold in a response message that is

not within a message/http container MUST replace each received

obs-fold with one or more SP octets prior to interpreting the field

value.

Historically, HTTP has allowed field content with text in the

ISO-8859-1 charset [[ISO-8859-1](https://www.rfc-editor.org/rfc/rfc7230" \l "ref-ISO-8859-1" \o "\"Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1\")], supporting other charsets only

through use of [[RFC2047](https://www.rfc-editor.org/rfc/rfc2047" \o "\"MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text\")] encoding. In practice, most HTTP header

field values use only a subset of the US-ASCII charset [[USASCII](https://www.rfc-editor.org/rfc/rfc7230" \l "ref-USASCII" \o "\"Coded Character Set -- 7-bit American Standard Code for Information Interchange\")].

Newly defined header fields SHOULD limit their field values to

US-ASCII octets. A recipient SHOULD treat other octets in field

content (obs-text) as opaque data.

**[3.2.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.5). Field Limits**

HTTP does not place a predefined limit on the length of each header

field or on the length of the header section as a whole, as described

in [Section 2.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-2.5). Various ad hoc limitations on individual header

field length are found in practice, often depending on the specific

field semantics.

A server that receives a request header field, or set of fields,

larger than it wishes to process MUST respond with an appropriate 4xx

(Client Error) status code. Ignoring such header fields would

increase the server's vulnerability to request smuggling attacks

([Section 9.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.5)).

A client MAY discard or truncate received header fields that are

larger than the client wishes to process if the field semantics are

such that the dropped value(s) can be safely ignored without changing

the message framing or response semantics.

**[3.2.6](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.2.6). Field Value Components**

Most HTTP header field values are defined using common syntax

components (token, quoted-string, and comment) separated by

whitespace or specific delimiting characters. Delimiters are chosen

from the set of US-ASCII visual characters not allowed in a token

(DQUOTE and "(),/:;<=>?@[\]{}").

token = 1\*tchar

tchar = "!" / "#" / "$" / "%" / "&" / "'" / "\*"

/ "+" / "-" / "." / "^" / "\_" / "`" / "|" / "~"

/ DIGIT / ALPHA

; any VCHAR, except delimiters

A string of text is parsed as a single value if it is quoted using

double-quote marks.

quoted-string = DQUOTE \*( qdtext / quoted-pair ) DQUOTE

qdtext = HTAB / SP /%x21 / %x23-5B / %x5D-7E / obs-text

obs-text = %x80-FF

Comments can be included in some HTTP header fields by surrounding

the comment text with parentheses. Comments are only allowed in

fields containing "comment" as part of their field value definition.

comment = "(" \*( ctext / quoted-pair / comment ) ")"

ctext = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-text

The backslash octet ("\") can be used as a single-octet quoting

mechanism within quoted-string and comment constructs. Recipients

that process the value of a quoted-string MUST handle a quoted-pair

as if it were replaced by the octet following the backslash.

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

A sender SHOULD NOT generate a quoted-pair in a quoted-string except

where necessary to quote DQUOTE and backslash octets occurring within

that string. A sender SHOULD NOT generate a quoted-pair in a comment

except where necessary to quote parentheses ["(" and ")"] and

backslash octets occurring within that comment.

**[3.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3). Message Body**

The message body (if any) of an HTTP message is used to carry the

payload body of that request or response. The message body is

identical to the payload body unless a transfer coding has been

applied, as described in [Section 3.3.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3.1).

message-body = \*OCTET

The rules for when a message body is allowed in a message differ for

requests and responses.

The presence of a message body in a request is signaled by a

Content-Length or Transfer-Encoding header field. Request message

framing is independent of method semantics, even if the method does

not define any use for a message body.

The presence of a message body in a response depends on both the

request method to which it is responding and the response status code

([Section 3.1.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.1.2)). Responses to the HEAD request method ([Section 4.3.2](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.2)

[of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.2)) never include a message body because the associated

response header fields (e.g., Transfer-Encoding, Content-Length,

etc.), if present, indicate only what their values would have been if

the request method had been GET ([Section 4.3.1 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.1)). 2xx

(Successful) responses to a CONNECT request method ([Section 4.3.6 of](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)

[[RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)) switch to tunnel mode instead of having a message body.

All 1xx (Informational), 204 (No Content), and 304 (Not Modified)

responses do not include a message body. All other responses do

include a message body, although the body might be of zero length.

**[3.3.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3.1). Transfer-Encoding**

The Transfer-Encoding header field lists the transfer coding names

corresponding to the sequence of transfer codings that have been (or

will be) applied to the payload body in order to form the message

body. Transfer codings are defined in [Section 4](https://www.rfc-editor.org/rfc/rfc7230" \l "section-4).

Transfer-Encoding = 1#transfer-coding

Transfer-Encoding is analogous to the Content-Transfer-Encoding field

of MIME, which was designed to enable safe transport of binary data

over a 7-bit transport service ([[RFC2045], Section 6](https://www.rfc-editor.org/rfc/rfc2045" \l "section-6)). However, safe

transport has a different focus for an 8bit-clean transfer protocol.

In HTTP's case, Transfer-Encoding is primarily intended to accurately

delimit a dynamically generated payload and to distinguish payload

encodings that are only applied for transport efficiency or security

from those that are characteristics of the selected resource.

A recipient MUST be able to parse the chunked transfer coding

([Section 4.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-4.1)) because it plays a crucial role in framing messages

when the payload body size is not known in advance. A sender MUST

NOT apply chunked more than once to a message body (i.e., chunking an

already chunked message is not allowed). If any transfer coding

other than chunked is applied to a request payload body, the sender

MUST apply chunked as the final transfer coding to ensure that the

message is properly framed. If any transfer coding other than

chunked is applied to a response payload body, the sender MUST either

apply chunked as the final transfer coding or terminate the message

by closing the connection.

For example,

Transfer-Encoding: gzip, chunked

indicates that the payload body has been compressed using the gzip

coding and then chunked using the chunked coding while forming the

message body.

Unlike Content-Encoding ([Section 3.1.2.1 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-3.1.2.1)),

Transfer-Encoding is a property of the message, not of the

representation, and any recipient along the request/response chain

MAY decode the received transfer coding(s) or apply additional

transfer coding(s) to the message body, assuming that corresponding

changes are made to the Transfer-Encoding field-value. Additional

information about the encoding parameters can be provided by other

header fields not defined by this specification.

Transfer-Encoding MAY be sent in a response to a HEAD request or in a

304 (Not Modified) response ([Section 4.1 of [RFC7232]](https://www.rfc-editor.org/rfc/rfc7232" \l "section-4.1)) to a GET

request, neither of which includes a message body, to indicate that

the origin server would have applied a transfer coding to the message

body if the request had been an unconditional GET. This indication

is not required, however, because any recipient on the response chain

(including the origin server) can remove transfer codings when they

are not needed.

A server MUST NOT send a Transfer-Encoding header field in any

response with a status code of 1xx (Informational) or 204 (No

Content). A server MUST NOT send a Transfer-Encoding header field in

any 2xx (Successful) response to a CONNECT request ([Section 4.3.6 of](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)

[[RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)).

Transfer-Encoding was added in HTTP/1.1. It is generally assumed

that implementations advertising only HTTP/1.0 support will not

understand how to process a transfer-encoded payload. A client MUST

NOT send a request containing Transfer-Encoding unless it knows the

server will handle HTTP/1.1 (or later) requests; such knowledge might

be in the form of specific user configuration or by remembering the

version of a prior received response. A server MUST NOT send a

response containing Transfer-Encoding unless the corresponding

request indicates HTTP/1.1 (or later).

A server that receives a request message with a transfer coding it

does not understand SHOULD respond with 501 (Not Implemented).

**[3.3.2](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3.2). Content-Length**

When a message does not have a Transfer-Encoding header field, a

Content-Length header field can provide the anticipated size, as a

decimal number of octets, for a potential payload body. For messages

that do include a payload body, the Content-Length field-value

provides the framing information necessary for determining where the

body (and message) ends. For messages that do not include a payload

body, the Content-Length indicates the size of the selected

representation ([Section 3 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-3)).

Content-Length = 1\*DIGIT

An example is

Content-Length: 3495

A sender MUST NOT send a Content-Length header field in any message

that contains a Transfer-Encoding header field.

A user agent SHOULD send a Content-Length in a request message when

no Transfer-Encoding is sent and the request method defines a meaning

for an enclosed payload body. For example, a Content-Length header

field is normally sent in a POST request even when the value is 0

(indicating an empty payload body). A user agent SHOULD NOT send a

Content-Length header field when the request message does not contain

a payload body and the method semantics do not anticipate such a

body.

A server MAY send a Content-Length header field in a response to a

HEAD request ([Section 4.3.2 of [RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.2)); a server MUST NOT send

Content-Length in such a response unless its field-value equals the

decimal number of octets that would have been sent in the payload

body of a response if the same request had used the GET method.

A server MAY send a Content-Length header field in a 304 (Not

Modified) response to a conditional GET request ([Section 4.1 of](https://www.rfc-editor.org/rfc/rfc7232" \l "section-4.1)

[[RFC7232]](https://www.rfc-editor.org/rfc/rfc7232" \l "section-4.1)); a server MUST NOT send Content-Length in such a response

unless its field-value equals the decimal number of octets that would

have been sent in the payload body of a 200 (OK) response to the same

request.

A server MUST NOT send a Content-Length header field in any response

with a status code of 1xx (Informational) or 204 (No Content). A

server MUST NOT send a Content-Length header field in any 2xx

(Successful) response to a CONNECT request ([Section 4.3.6 of](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)

[[RFC7231]](https://www.rfc-editor.org/rfc/rfc7231" \l "section-4.3.6)).

Aside from the cases defined above, in the absence of

Transfer-Encoding, an origin server SHOULD send a Content-Length

header field when the payload body size is known prior to sending the

complete header section. This will allow downstream recipients to

measure transfer progress, know when a received message is complete,

and potentially reuse the connection for additional requests.

Any Content-Length field value greater than or equal to zero is

valid. Since there is no predefined limit to the length of a

payload, a recipient MUST anticipate potentially large decimal

numerals and prevent parsing errors due to integer conversion

overflows ([Section 9.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.3)).

If a message is received that has multiple Content-Length header

fields with field-values consisting of the same decimal value, or a

single Content-Length header field with a field value containing a

list of identical decimal values (e.g., "Content-Length: 42, 42"),

indicating that duplicate Content-Length header fields have been

generated or combined by an upstream message processor, then the

recipient MUST either reject the message as invalid or replace the

duplicated field-values with a single valid Content-Length field

containing that decimal value prior to determining the message body

length or forwarding the message.

Note: HTTP's use of Content-Length for message framing differs

significantly from the same field's use in MIME, where it is an

optional field used only within the "message/external-body"

media-type.

**[3.3.3](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.3.3). Message Body Length**

The length of a message body is determined by one of the following

(in order of precedence):

1. Any response to a HEAD request and any response with a 1xx

(Informational), 204 (No Content), or 304 (Not Modified) status

code is always terminated by the first empty line after the

header fields, regardless of the header fields present in the

message, and thus cannot contain a message body.

2. Any 2xx (Successful) response to a CONNECT request implies that

the connection will become a tunnel immediately after the empty

line that concludes the header fields. A client MUST ignore any

Content-Length or Transfer-Encoding header fields received in

such a message.

3. If a Transfer-Encoding header field is present and the chunked

transfer coding ([Section 4.1](https://www.rfc-editor.org/rfc/rfc7230" \l "section-4.1)) is the final encoding, the message

body length is determined by reading and decoding the chunked

data until the transfer coding indicates the data is complete.

If a Transfer-Encoding header field is present in a response and

the chunked transfer coding is not the final encoding, the

message body length is determined by reading the connection until

it is closed by the server. If a Transfer-Encoding header field

is present in a request and the chunked transfer coding is not

the final encoding, the message body length cannot be determined

reliably; the server MUST respond with the 400 (Bad Request)

status code and then close the connection.

If a message is received with both a Transfer-Encoding and a

Content-Length header field, the Transfer-Encoding overrides the

Content-Length. Such a message might indicate an attempt to

perform request smuggling ([Section 9.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.5)) or response splitting

([Section 9.4](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.4)) and ought to be handled as an error. A sender MUST

remove the received Content-Length field prior to forwarding such

a message downstream.

4. If a message is received without Transfer-Encoding and with

either multiple Content-Length header fields having differing

field-values or a single Content-Length header field having an

invalid value, then the message framing is invalid and the

recipient MUST treat it as an unrecoverable error. If this is a

request message, the server MUST respond with a 400 (Bad Request)

status code and then close the connection. If this is a response

message received by a proxy, the proxy MUST close the connection

to the server, discard the received response, and send a 502 (Bad

Gateway) response to the client. If this is a response message

received by a user agent, the user agent MUST close the

connection to the server and discard the received response.

5. If a valid Content-Length header field is present without

Transfer-Encoding, its decimal value defines the expected message

body length in octets. If the sender closes the connection or

the recipient times out before the indicated number of octets are

received, the recipient MUST consider the message to be

incomplete and close the connection.

6. If this is a request message and none of the above are true, then

the message body length is zero (no message body is present).

7. Otherwise, this is a response message without a declared message

body length, so the message body length is determined by the

number of octets received prior to the server closing the

connection.

Since there is no way to distinguish a successfully completed,

close-delimited message from a partially received message interrupted

by network failure, a server SHOULD generate encoding or

length-delimited messages whenever possible. The close-delimiting

feature exists primarily for backwards compatibility with HTTP/1.0.

A server MAY reject a request that contains a message body but not a

Content-Length by responding with 411 (Length Required).

Unless a transfer coding other than chunked has been applied, a

client that sends a request containing a message body SHOULD use a

valid Content-Length header field if the message body length is known

in advance, rather than the chunked transfer coding, since some

existing services respond to chunked with a 411 (Length Required)

status code even though they understand the chunked transfer coding.

This is typically because such services are implemented via a gateway

that requires a content-length in advance of being called and the

server is unable or unwilling to buffer the entire request before

processing.

A user agent that sends a request containing a message body MUST send

a valid Content-Length header field if it does not know the server

will handle HTTP/1.1 (or later) requests; such knowledge can be in

the form of specific user configuration or by remembering the version

of a prior received response.

If the final response to the last request on a connection has been

completely received and there remains additional data to read, a user

agent MAY discard the remaining data or attempt to determine if that

data belongs as part of the prior response body, which might be the

case if the prior message's Content-Length value is incorrect. A

client MUST NOT process, cache, or forward such extra data as a

separate response, since such behavior would be vulnerable to cache

poisoning.

**[3.4](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.4). Handling Incomplete Messages**

A server that receives an incomplete request message, usually due to

a canceled request or a triggered timeout exception, MAY send an

error response prior to closing the connection.

A client that receives an incomplete response message, which can

occur when a connection is closed prematurely or when decoding a

supposedly chunked transfer coding fails, MUST record the message as

incomplete. Cache requirements for incomplete responses are defined

in [Section 3 of [RFC7234]](https://www.rfc-editor.org/rfc/rfc7234" \l "section-3).

If a response terminates in the middle of the header section (before

the empty line is received) and the status code might rely on header

fields to convey the full meaning of the response, then the client

cannot assume that meaning has been conveyed; the client might need

to repeat the request in order to determine what action to take next.

A message body that uses the chunked transfer coding is incomplete if

the zero-sized chunk that terminates the encoding has not been

received. A message that uses a valid Content-Length is incomplete

if the size of the message body received (in octets) is less than the

value given by Content-Length. A response that has neither chunked

transfer coding nor Content-Length is terminated by closure of the

connection and, thus, is considered complete regardless of the number

of message body octets received, provided that the header section was

received intact.

**[3.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-3.5). Message Parsing Robustness**

Older HTTP/1.0 user agent implementations might send an extra CRLF

after a POST request as a workaround for some early server

applications that failed to read message body content that was not

terminated by a line-ending. An HTTP/1.1 user agent MUST NOT preface

or follow a request with an extra CRLF. If terminating the request

message body with a line-ending is desired, then the user agent MUST

count the terminating CRLF octets as part of the message body length.

In the interest of robustness, a server that is expecting to receive

and parse a request-line SHOULD ignore at least one empty line (CRLF)

received prior to the request-line.

Although the line terminator for the start-line and header fields is

the sequence CRLF, a recipient MAY recognize a single LF as a line

terminator and ignore any preceding CR.

Although the request-line and status-line grammar rules require that

each of the component elements be separated by a single SP octet,

recipients MAY instead parse on whitespace-delimited word boundaries

and, aside from the CRLF terminator, treat any form of whitespace as

the SP separator while ignoring preceding or trailing whitespace;

such whitespace includes one or more of the following octets: SP,

HTAB, VT (%x0B), FF (%x0C), or bare CR. However, lenient parsing can

result in security vulnerabilities if there are multiple recipients

of the message and each has its own unique interpretation of

robustness (see [Section 9.5](https://www.rfc-editor.org/rfc/rfc7230" \l "section-9.5)).

When a server listening only for HTTP request messages, or processing

what appears from the start-line to be an HTTP request message,

receives a sequence of octets that does not match the HTTP-message

grammar aside from the robustness exceptions listed above, the server

SHOULD respond with a 400 (Bad Request) response.