Hypertext Transfer Protocol

The **Hypertext Transfer Protocol** (**HTTP**) is an application layer protocol for distributed, collaborative, hypermedia information systems.[1] HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access, for example by a mouse click or by tapping the screen in a web browser.

Development of HTTP was initiated by Tim Berners-Lee at CERN in 1989. Development of early HTTP Requests for Comments (RFCs) was a coordinated effort by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), with work later moving to the IETF.

HTTP/1.1 was first documented in RFC 2068 (https://tools.ietf.org/html/rfc2068) in 1997. That specification was obsoleted by RFC 2616 (https://tools.ietf.org/html/rfc2616) in 1999, which was likewise replaced by the RFC 7230 (https://tools.ietf.org/html/rfc7230) family of RFCs in 2014.

HTTP/2 is a more efficient expression of HTTP's semantics "on the wire", and was published in 2015; it is now supported by virtually all web browsers[2] and major web servers over Transport Layer Security (TLS) using an Application-Layer Protocol Negotiation (ALPN) extension[3] where TLS 1.2 or newer is required.[4][5]

HTTP/3 is the proposed successor to HTTP/2,[6][7] which is already in use on the web (enabled by default in latest macOS), using UDP instead of TCP for the underlying transport protocol. Like HTTP/2, it does not obsolete previous major versions of the protocol. Support for HTTP/3 was added to Cloudflare and Google Chrome in September 2019,[8][9] and can be enabled in the stable versions of Chrome and Firefox.[10]

Contents

Hypertext Transfer Protocol

International standard	RFC 1945 HTTP/1.0 (1996) RFC 2616 HTTP/1.1 (1999) RFC 7540 HTTP/2 (2015) RFC 7541 Header Compression (2, 2015) RFC 7230 Message Syntax and Routing (1.1, 2014) RFC 7231 Semantics and Content (1.1, 2014) RFC 7232 Conditional Requests (1.1, 2014) RFC 7233 Range Requests (1.1, 2014) RFC 7234 Caching (1.1, 2014) RFC 7235 Authentication (1.1, 2014)
Developed by	initially CERN; IETF, W3C
Introduced	1991

Technical overview

HTTP functions as a request–response protocol in the client–server computing model. A web browser, for example, may be the *client* and an application running on a computer hosting a website may be the *server*. The client submits an HTTP *request* message to the server. The server, which provides *resources* such as HTML files and other content, or performs other functions on behalf of the client, returns a *response* message to the client. The response contains completion status information about the request and may also contain requested content in its message body.

URL beginning with the HTTP scheme and the WWW domain name label

A web browser is an example of a *user agent* (UA). Other types of user agent include the indexing software used by search providers (web crawlers), voice browsers, mobile apps, and other software that accesses, consumes, or displays web content.

HTTP is designed to permit intermediate network elements to improve or enable communications between clients and servers. High-traffic websites often benefit from web cache servers that deliver content on behalf of upstream servers to improve response time. Web browsers cache previously accessed web resources and reuse them, when possible, to reduce network traffic. HTTP proxy servers at private network boundaries can facilitate communication for clients without a globally routable address, by relaying messages with external servers.

HTTP is an application layer protocol designed within the framework of the Internet protocol suite. Its definition presumes an underlying and reliable transport layer protocol,[11] and Transmission Control Protocol (TCP) is commonly used. However, HTTP can be adapted to use unreliable protocols such as the User Datagram Protocol (UDP), for example in HTTPU and Simple Service Discovery Protocol (SSDP).

HTTP resources are identified and located on the network by Uniform Resource Locators (URLs), using the Uniform Resource Identifiers (URI's) schemes *http* and *https*. As defined in RFC 3986 (https://tools.ietf.org/html/rfc3986), URIs are encoded as hyperlinks in HTML documents, so as to form interlinked hypertext documents.

HTTP/1.1 is a revision of the original HTTP (HTTP/1.0). In HTTP/1.0 a separate connection to the same server is made for every resource request. HTTP/1.1 can reuse a connection multiple times to download images, scripts, stylesheets, *etc* after the page has been delivered. HTTP/1.1 communications therefore experience less latency as the establishment of TCP connections presents considerable overhead.

History


Tim Berners-Lee

The term hypertext was coined by Ted Nelson in 1965 in the Xanadu Project, which was in turn inspired by Vannevar Bush's 1930s vision of the microfilm-based information retrieval and management "memex" system described in his 1945 essay "As We May Think". Tim Berners-Lee and his team at CERN are credited with inventing the original HTTP, along with HTML and the associated technology for a web server and a text-based web browser. Berners-Lee first proposed the "WorldWideWeb" project in 1989—now known as the World Wide Web. The first version of the protocol had only one method, namely GET, which would request a page from a server.[12] The response from the server was always an HTML page.[13]

The first documented version of HTTP was **HTTP V0.9 (https://www.w3.org/pub/WWW/Protocols/HTTP/AsImplemented.html)** (1991). Dave Raggett led the HTTP Working Group (HTTP WG) in 1995 and wanted to expand the protocol with extended operations, extended negotiation, richer meta-information, tied with a security protocol which became more efficient by adding additional methods and header fields.[14][15] RFC 1945 (https://tools.ietf.org/html/rfc1945) officially introduced and recognized HTTP V1.0 in 1996.

The HTTP WG planned to publish new standards in December 1995[16] and the support for pre-standard HTTP/1.1 based on the then developing RFC 2068 (https://tools.ietf.org/html/rfc2068) (called HTTP-NG) was rapidly adopted by the major browser developers in early 1996. End-user adoption of the new browsers was rapid. In March 1996, one web hosting company reported that over 40% of browsers in use on the Internet were HTTP 1.1 compliant. That same web hosting company reported that by June 1996, 65% of all browsers accessing their servers were HTTP/1.1 compliant.[17] The HTTP/1.1 standard as defined in RFC 2068 (https://tools.ietf.org/html/rfc2068) was officially released in January 1997. Improvements and updates to the HTTP/1.1 standard were released under RFC 2616 (https://tools.ietf.org/html/rfc2616) in June 1999.

In 2007, the **HTTP Working Group (https://httpwg.org/)** was formed, in part, to revise and clarify the HTTP/1.1 specification. In June 2014, the WG released an updated six-part specification obsoleting RFC 2616 (https://tools.ietf.org/html/rfc2616):

- RFC 7230 (https://tools.ietf.org/html/rfc7230), *HTTP/1.1: Message Syntax and Routing*
- RFC 7231 (https://tools.ietf.org/html/rfc7231), *HTTP/1.1: Semantics and Content*
- RFC 7232 (https://tools.ietf.org/html/rfc7232), *HTTP/1.1: Conditional Requests*
- RFC 7233 (https://tools.ietf.org/html/rfc7233), *HTTP/1.1: Range Requests*
- RFC 7234 (https://tools.ietf.org/html/rfc7234), *HTTP/1.1: Caching*
- RFC 7235 (https://tools.ietf.org/html/rfc7235), *HTTP/1.1: Authentication*

HTTP/2 was published as RFC 7540 (https://tools.ietf.org/html/rfc7540) in May 2015.

Year	HTTP Version
1991	0.9
1996	1.0
1997	1.1
2015	2.0
2018	3.0

HTTP session

An HTTP session is a sequence of network request-response transactions. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server (typically port 80, occasionally port 8080; see List of TCP and UDP port numbers). An HTTP server listening on that port waits for a client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message of its own. The body of this message is typically the requested resource, although an error message or other information may also be returned.[1]

Persistent connections

In HTTP/0.9 and 1.0, the connection is closed after a single request/response pair. In HTTP/1.1 a keep-alive-mechanism was introduced, where a connection could be reused for more than one request. Such *persistent connections* reduce request latency perceptibly because the client does not need to re-negotiate the TCP 3-Way-Handshake connection after the first request has been sent. Another positive side effect is that, in general, the connection becomes faster with time due to TCP's slow-start-mechanism.

Version 1.1 of the protocol also made bandwidth optimization improvements to HTTP/1.0. For example, HTTP/1.1 introduced chunked transfer encoding to allow content on persistent connections to be streamed rather than buffered. HTTP pipelining further reduces lag time, allowing clients to send multiple requests before waiting for each response. Another addition to the protocol was byte serving, where a server transmits just the portion of a resource explicitly requested by a client.

HTTP session state

HTTP is a stateless protocol. A stateless protocol does not require the HTTP server to retain information or status about each user for the duration of multiple requests. However, some web applications implement states or server side sessions using for instance HTTP cookies or hidden variables within web forms.

HTTP authentication

HTTP provides multiple authentication schemes such as basic access authentication and digest access authentication which operate via a challenge-response mechanism whereby the server identifies and issues a challenge before serving the requested content.

HTTP provides a general framework for access control and authentication, via an extensible set of challenge-response authentication schemes, which can be used by a server to challenge a client request and by a client to provide authentication information.[18]

Authentication realms

The HTTP Authentication specification also provides an arbitrary, implementation-specific construct for further dividing resources common to a given root URI. The realm value string, if present, is combined with the canonical root URI to form the protection space component of the challenge. This in effect allows the server to define separate authentication scopes under one root URI.[18]

Message format

The client sends **requests** to the server and the server sends **responses**.

Request message

The request message consists of the following:

- a request line (e.g., *GET /images/logo.png HTTP/1.1*, which requests a resource called `/images/logo.png` from the server)
- request header fields (e.g., *Accept-Language: en*)
- an empty line
- an optional message body

The request line and other header fields must each end with <CR><LF> (that is, a carriage return character followed by a line feed character). The empty line must consist of only <CR><LF> and no other whitespace.[19] In the HTTP/1.1 protocol, all header fields except *Host* are optional.

A request line containing only the path name is accepted by servers to maintain compatibility with HTTP clients before the HTTP/1.0 specification in RFC 1945 (https://tools.ietf.org/html/rfc1945).[20]

Request methods

An HTTP 1.1 request made using telnet. The request message, response header section, and response body are highlighted.

HTTP defines methods (sometimes referred to as *verbs*, but nowhere in the specification does it mention *verb*, nor is OPTIONS or HEAD a verb) to indicate the desired action to be performed on the identified resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. Often, the resource corresponds to a file or the output of an executable residing on the server. The HTTP/1.0 specification[21] defined the GET, HEAD and POST methods and the HTTP/1.1 specification[22] added five new methods: OPTIONS, PUT, DELETE, TRACE and CONNECT. By being specified in these documents, their semantics are well-known and can be depended on. Any client can use any method and the server can be configured to support any combination of methods. If a method is unknown to an intermediate, it will be treated as an unsafe and non-idempotent method. There is no limit to the number of methods that can be defined and this allows for future methods to be specified without breaking existing infrastructure. For example, WebDAV defined seven new methods and RFC 5789 (https://tools.ietf.org/html/rfc5789) specified the PATCH method.

Method names are case sensitive.[23][24] This is in contrast to HTTP header field names which are case-insensitive.[25]

GET
: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. (This is also true of some other HTTP methods.)[1] The W3C has published guidance principles on this distinction, saying, "Web application design should be informed by the above principles, but also by the relevant limitations."[26] See safe methods below.

HEAD
: The HEAD method asks for a response identical to that of a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.

POST
: The POST method requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, for example, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.[27]

PUT
: The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.[28]

DELETE
: The DELETE method deletes the specified resource.

TRACE
: The TRACE method echoes the received request so that a client can see what (if any) changes or additions have been made by intermediate servers.

OPTIONS
: The OPTIONS method returns the HTTP methods that the server supports for the specified URL. This can be used to check the functionality of a web server by requesting '*' instead of a specific resource.

CONNECT
: [29] The CONNECT method converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.[30][31] See HTTP CONNECT method.

PATCH
: The PATCH method applies partial modifications to a resource.[32]

All general-purpose HTTP servers are required to implement at least the GET and HEAD methods, and all other methods are considered optional by the specification.[33]

Safe methods

Some of the methods (for example, GET, HEAD, OPTIONS and TRACE) are, by convention, defined as *safe*, which means they are intended only for information retrieval and should not change the state of the server. In other words, they should not have side effects, beyond relatively harmless effects such as logging, web caching, the serving of banner advertisements or incrementing a web counter. Making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe. However, this is not mandated by the standard, and it is explicitly acknowledged that it cannot be guaranteed.

By contrast, methods such as POST, PUT, DELETE and PATCH are intended for actions that may cause side effects either on the server, or external side effects such as financial transactions or transmission of email. Such methods are therefore not usually used by conforming web robots or web crawlers; some that do not conform tend to make requests without regard to context or consequences.

Despite the prescribed safety of *GET* requests, in practice their handling by the server is not technically limited in any way. Therefore, careless or deliberate programming can cause non-trivial changes on the server. This is discouraged, because it can cause problems for web caching, search engines and other automated agents, which can make unintended changes on the server. For example, a website might allow deletion of a resource through a URL such as *http://example.com/article/1234/delete*, which, if arbitrarily fetched, even using *GET*, would simply delete the article.[34]

One example of this occurring in practice was during the short-lived Google Web Accelerator beta, which prefetched arbitrary URLs on the page a user was viewing, causing records to be automatically altered or deleted *en masse*. The beta was suspended only weeks after its first release, following widespread criticism.[35][34]

Idempotent methods and web applications

Methods PUT and DELETE are defined to be idempotent, meaning that multiple identical requests should have the same effect as a single request. Methods GET, HEAD, OPTIONS and TRACE, being prescribed as safe, should also be idempotent, as HTTP is a stateless protocol.[1]

In contrast, the POST method is not necessarily idempotent, and therefore sending an identical POST request multiple times may further affect state or cause further side effects (such as financial transactions). In some cases this may be desirable, but in other cases this could be due to an accident, such as when a user does not realize that their action will result in sending another request, or they did not receive adequate feedback that their first request was successful. While web browsers may show alert dialog boxes to warn users in some cases where reloading a page may re-submit a POST request, it is generally up to the web application to handle cases where a POST request should not be submitted more than once.

Note that whether a method is idempotent is not enforced by the protocol or web server. It is perfectly possible to write a web application in which (for example) a database insert or other non-idempotent action is triggered by a GET or other request. Ignoring this recommendation, however, may result in undesirable consequences, if a user agent assumes that repeating the same request is safe when it is not.

Security

The TRACE method can be used as part of a class of attacks known as cross-site tracing; for that reason, common security advice is for it to be disabled in the server configuration.[36] Microsoft IIS supports a proprietary "TRACK" method, which behaves similarly, and which is likewise recommended to be disabled.[36]

Security of HTTP methods

HTTP method	RFC	Request has Body	Response has Body	Safe	Idempotent	Cacheable
GET	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Optional	No	Yes	Yes	Yes
POST	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Yes	Yes	No	No	Yes
PUT	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Yes	Yes	No	Yes	No
DELETE	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Optional	Yes	No	Yes	No
CONNECT	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Optional	Yes	No	No	No
OPTIONS	RFC 7231 (https://tools.ietf.org/html/rfc7231)	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231 (https://tools.ietf.org/html/rfc7231)	No	Yes	Yes	Yes	No
PATCH	RFC 5789 (https://tools.ietf.org/html/rfc5789)	Yes	Yes	No	No	No

Response message

The response message consists of the following:

- a status line which includes the status code and reason message (e.g., *HTTP/1.1 200 OK*, which indicates that the client's request succeeded)
- response header fields (e.g., *Content-Type: text/html*)
- an empty line
- an optional message body

The status line and other header fields must all end with <CR><LF>. The empty line must consist of only <CR><LF> and no other whitespace.[19] This strict requirement for <CR><LF> is relaxed somewhat within message bodies for consistent use of other system linebreaks such as <CR> or <LF> alone.[37]

Status codes

In HTTP/1.0 and since, the first line of the HTTP response is called the *status line* and includes a numeric *status code* (such as "404") and a textual *reason phrase* (such as "Not Found"). The way the user agent handles the response depends primarily on the code, and secondarily on the other response header fields. Custom status codes can be used, for if the user agent encounters a code it does not recognize, it can use the first digit of the code to determine the general class of the response.[38]

The standard *reason phrases* are only recommendations, and can be replaced with "local equivalents" at the web developer's discretion. If the status code indicated a problem, the user agent might display the *reason phrase* to the user to provide further information about the nature of the problem. The standard also allows the user agent to attempt to interpret the *reason phrase*, though this might be unwise since the standard explicitly

specifies that status codes are machine-readable and *reason phrases* are human-readable. HTTP status code is primarily divided into five groups for better explanation of request and responses between client and server as named:

- Informational 1XX
- Successful 2XX
- Redirection 3XX
- Client Error 4XX
- Server Error 5XX

Encrypted connections

The most popular way of establishing an encrypted HTTP connection is HTTPS.[39] Two other methods for establishing an encrypted HTTP connection also exist: Secure Hypertext Transfer Protocol, and using the HTTP/1.1 Upgrade header to specify an upgrade to TLS. Browser support for these two is, however, nearly non-existent.[40][41][42]

Example session

Below is a sample conversation between an HTTP client and an HTTP server running on www.example.com, port 80.

Client request

```
GET / HTTP/1.1
Host: www.example.com
```

A client request (consisting in this case of the request line and only one header field) is followed by a blank line, so that the request ends with a double newline, each in the form of a carriage return followed by a line feed. The "Host" field distinguishes between various DNS names sharing a single IP address, allowing name-based virtual hosting. While optional in HTTP/1.0, it is mandatory in HTTP/1.1. (The "/" means /index.html if there is one.)

Server response

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 155
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Hello World, this is a very simple HTML document.</p>
  </body>
</html>
```

The ETag (entity tag) header field is used to determine if a cached version of the requested resource is identical to the current version of the resource on the server. *Content-Type* specifies the Internet media type of the data conveyed by the HTTP message, while *Content-Length* indicates its length in bytes. The HTTP/1.1 webserver publishes its ability to respond to requests for certain byte ranges of the document by setting the field *Accept-Ranges: bytes*. This is useful, if the client needs to have only certain portions[43] of a resource sent by the server, which is called byte serving. When *Connection: close* is sent, it means that the web server will close the TCP connection immediately after the transfer of this response.

Most of the header lines are optional. When *Content-Length* is missing the length is determined in other ways. Chunked transfer encoding uses a chunk size of 0 to mark the end of the content. *Identity* encoding without *Content-Length* reads content until the socket is closed.

A *Content-Encoding* like *gzip* can be used to compress the transmitted data.

Similar protocols

- The Gopher protocol is a content delivery protocol that was displaced by HTTP in the early 1990s.
- The SPDY protocol is an alternative to HTTP developed at Google, superseded by HTTP/2.

See also

- Comparison of file transfer protocols
- Constrained Application Protocol – a semantically similar protocol to HTTP but used UDP or UDP-like messages targeted for devices with limited processing capability; re-uses HTTP and other internet concepts like Internet media type and web linking (RFC 5988)[44]
- Content negotiation
- Curl-loader – HTTP/S loading and testing open-source software
- Digest access authentication
- Fiddler (software)
- HTTP compression
- HTTP/2 – developed by the IETF's Hypertext Transfer Protocol (httpbis) working group[45]
- HTTP-MPLEX – A backwards compatible enhancement to HTTP to improve page and web object retrieval time in congested networks proposed by Robert Mattson
- List of HTTP header fields
- List of HTTP status codes
- Representational state transfer (REST)
- Variant object
- Web cache
- WebSocket
- Wireshark

References

1. Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim (June 1999). *Hypertext Transfer Protocol – HTTP/1.1* (https://tools.ietf.org/html/rfc2616). IETF. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616 (https://tools.ietf.org/html/rfc2616).

2. "Can I use... Support tables for HTML5, CSS3, etc" (https://caniuse.com/#search=http2). *caniuse.com*. Retrieved 2020-06-02.
3. "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension" (https://tools.ietf.org/html/rfc7301). IETF. July 2014. RFC 7301 (https://tools.ietf.org/html/rfc7301).
4. Belshe, M.; Peon, R.; Thomson, M. "Hypertext Transfer Protocol Version 2, Use of TLS Features" (https://http2.github.io/http2-spec/#TLSUsage). Retrieved 2015-02-10.
5. Benjamin, David. "Using TLS 1.3 with HTTP/2" (https://tools.ietf.org/html/rfc8740.html). *tools.ietf.org*. Retrieved 2020-06-02. "This lowers the barrier for deploying TLS 1.3, a major security improvement over TLS 1.2."
6. Bishop, Mike (July 9, 2019). "Hypertext Transfer Protocol Version 3 (HTTP/3)" (https://tools.ietf.org/html/draft-ietf-quic-http-22). *tools.ietf.org*. draft-ietf-quic-http-22. Retrieved 2019-08-16.
7. Cimpanu, Catalin. "HTTP-over-QUIC to be renamed HTTP/3 | ZDNet" (https://www.zdnet.com/article/http-over-quic-to-be-renamed-http3/). *ZDNet*. Retrieved 2018-11-19.
8. Cimpanu, Catalin (26 September 2019). "Cloudflare, Google Chrome, and Firefox add HTTP/3 support" (https://www.zdnet.com/article/cloudflare-google-chrome-and-firefox-add-http3-support/). *ZDNet*. Retrieved 27 September 2019.
9. "HTTP/3: the past, the present, and the future" (https://blog.cloudflare.com/http3-the-past-present-and-future/). *The Cloudflare Blog*. 2019-09-26. Retrieved 2019-10-30.
10. "Firefox Nightly supports HTTP 3 - General - Cloudflare Community" (https://community.cloudflare.com/t/firefox-nightly-supports-http-3/127778). 2019-11-19. Retrieved 2020-01-23.
11. "Overall Operation" (https://tools.ietf.org/html/rfc2616#section-1.4). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 12. sec. 1.4. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
12. Berners-Lee, Tim. "HyperText Transfer Protocol" (https://www.w3.org/History/19921103-hypertext/hypertext/WWW/Protocols/HTTP.html). World Wide Web Consortium. Retrieved 31 August 2010.
13. Tim Berners-Lee. "The Original HTTP as defined in 1991" (https://www.w3.org/Protocols/HTTP/AsImplemented.html). World Wide Web Consortium. Retrieved 24 July 2010.
14. Raggett, Dave. "Dave Raggett's Bio" (https://www.w3.org/People/Raggett/profile.html). World Wide Web Consortium. Retrieved 11 June 2010.
15. Raggett, Dave; Berners-Lee, Tim. "Hypertext Transfer Protocol Working Group" (https://www.w3.org/Arena/webworld/httpwgcharter.html). World Wide Web Consortium. Retrieved 29 September 2010.
16. Raggett, Dave. "HTTP WG Plans" (https://www.w3.org/Arena/webworld/httpwgplans.html). World Wide Web Consortium. Retrieved 29 September 2010.
17. "HTTP/1.1" (http://webarchive.loc.gov/all/20011121001051/https://www.webcom.com/glossary/http1.1.shtml). *Webcom.com Glossary entry*. Archived from the original (https://www.webcom.com/glossary/http1.1.shtml) on 2001-11-21. Retrieved 2009-05-29.
18. Fielding, Roy T.; Reschke, Julian F. (June 2014). *Hypertext Transfer Protocol (HTTP/1.1): Authentication* (https://tools.ietf.org/html/rfc7235). IETF. doi:10.17487/RFC7235 (https://doi.org/10.17487%2FRFC7235). RFC 7235 (https://tools.ietf.org/html/rfc7235).
19. "HTTP Message" (https://tools.ietf.org/html/rfc2616#section-4). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 31. sec. 4. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
20. "Apache Week. HTTP/1.1" (http://www.apacheweek.com/features/http11). 090502 apacheweek.com
21. Berners-Lee, Tim; Fielding, Roy T.; Nielsen, Henrik Frystyk. "Method Definitions" (https://tools.ietf.org/html/rfc1945#section-8). *Hypertext Transfer Protocol – HTTP/1.0* (https://tools.ietf.org/html/rfc1945). IETF. pp. 30–32. sec. 8. doi:10.17487/RFC1945 (https://doi.org/10.17487%2FRFC1945). RFC 1945 (https://tools.ietf.org/html/rfc1945).

22. "Method Definitions" (https://tools.ietf.org/html/rfc2616#section-9). *RFC 2616* (https://tools.ietf.org/html/rfc2616). pp. 51–57. sec. 9. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
23. "RFC-7210 section 3.1.1" (https://tools.ietf.org/html/rfc7230#section-3.1.1). Tools.ietf.org. Retrieved 2019-06-26.
24. "RFC-7231 section 4.1" (https://tools.ietf.org/html/rfc7231#section-4.1). Tools.ietf.org. Retrieved 2019-06-26.
25. "RFC-7230 section 3.2" (https://tools.ietf.org/html/rfc7230#section-3.2). Tools.ietf.org. Retrieved 2019-06-26.
26. Jacobs, Ian (2004). "URIs, Addressability, and the use of HTTP GET and POST" (https://www.w3.org/2001/tag/doc/whenToUseGet.html#checklist). *Technical Architecture Group finding*. W3C. Retrieved 26 September 2010.
27. "POST" (https://tools.ietf.org/html/rfc2616#section-9.5). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 54. sec. 9.5. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
28. "PUT" (https://tools.ietf.org/html/rfc2616#section-9.6). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 55. sec. 9.6. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
29. "CONNECT" (https://tools.ietf.org/html/rfc2616#section-9.9). *Hypertext Transfer Protocol – HTTP/1.1* (https://tools.ietf.org/html/rfc2616). IETF. June 1999. p. 57. sec. 9.9. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616 (https://tools.ietf.org/html/rfc2616). Retrieved 23 February 2014.
30. Khare, Rohit; Lawrence, Scott (May 2000). *Upgrading to TLS Within HTTP/1.1* (https://tools.ietf.org/html/rfc2817). IETF. doi:10.17487/RFC2817 (https://doi.org/10.17487%2FRFC2817). RFC 2817 (https://tools.ietf.org/html/rfc2817).
31. "Vulnerability Note VU#150227: HTTP proxy default configurations allow arbitrary TCP connections" (https://www.kb.cert.org/vuls/id/150227). US-CERT. 2002-05-17. Retrieved 2007-05-10.
32. Dusseault, Lisa; Snell, James M. (March 2010). *PATCH Method for HTTP* (https://tools.ietf.org/html/rfc5789). IETF. doi:10.17487/RFC5789 (https://doi.org/10.17487%2FRFC5789). RFC 5789 (https://tools.ietf.org/html/rfc5789).
33. "Method" (https://tools.ietf.org/html/rfc2616#section-5.1.1). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 36. sec. 5.1.1. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
34. Ediger, Brad (2007-12-21). *Advanced Rails: Building Industrial-Strength Web Apps in Record Time* (https://shop.oreilly.com/product/9780596510329.do). O'Reilly Media, Inc. p. 188. ISBN 978-0596519728. "A common mistake is to use GET for an action that updates a resource. [...] This problem came into the Rails public eye in 2005, when the Google Web Accelerator was released."
35. Cantrell, Christian (2005-06-01). "What Have We Learned From the Google Web Accelerator?" (https://web.archive.org/web/20170819161233/http://blogs.adobe.com/cantrell/archives/2005/06/what_have_we_le.html). *Adobe Blogs*. Adobe. Archived from the original (https://blogs.adobe.com/cantrell/archives/2005/06/what_have_we_le.html) on 2017-08-19. Retrieved 2018-11-19.
36. "Cross Site Tracing" (https://www.owasp.org/index.php/Cross_Site_Tracing). OWASP. Retrieved 2016-06-22.
37. "Canonicalization and Text Defaults" (https://tools.ietf.org/html/rfc2616#section-3.7.1). *RFC 2616* (https://tools.ietf.org/html/rfc2616). sec. 3.7.1. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.

38. "Status-Line" (https://tools.ietf.org/html/rfc2616#section-6.1). *RFC 2616* (https://tools.ietf.org/html/rfc2616). p. 39. sec. 6.1. doi:10.17487/RFC2616 (https://doi.org/10.17487%2FRFC2616). RFC 2616.
39. Canavan, John (2001). *Fundamentals of Networking Security*. Norwood, MA: Artech House. pp. 82–83. ISBN 9781580531764.
40. Zalewski, Michal. "Browser Security Handbook" (https://code.google.com/p/browsersec/wiki/Part1#True_URL_schemes). Retrieved 30 April 2015.
41. "Chromium Issue 4527: implement RFC 2817: Upgrading to TLS Within HTTP/1.1" (https://code.google.com/p/chromium/issues/detail?id=4527). Retrieved 30 April 2015.
42. "Mozilla Bug 276813 – [RFE] Support RFC 2817 / TLS Upgrade for HTTP 1.1" (https://bugzilla.mozilla.org/show_bug.cgi?id=276813). Retrieved 30 April 2015.
43. Luotonen, Ari; Franks, John (February 22, 1996). *Byte Range Retrieval Extension to HTTP* (https://tools.ietf.org/html/draft-ietf-http-range-retrieval-00). IETF. I-D draft-ietf-http-range-retrieval-00.
44. Nottingham, Mark (October 2010). *Web Linking* (https://tools.ietf.org/html/rfc5988). IETF. doi:10.17487/RFC5988 (https://doi.org/10.17487%2FRFC5988). RFC 5988 (https://tools.ietf.org/html/rfc5988).
45. "Hypertext Transfer Protocol Bis (httpbis) – Charter" (https://datatracker.ietf.org/wg/httpbis/charter/). IETF. 2012.

External links

- "Change History for HTTP" (https://www.w3.org/Protocols/History.html). W3.org. Retrieved 2010-08-01. A detailed technical history of HTTP.
- "Design Issues for HTTP" (https://www.w3.org/Protocols/DesignIssues.html). W3.org. Retrieved 2010-08-01. Design Issues by Berners-Lee when he was designing the protocol.
- "Classic HTTP Documents" (https://www.w3.org/Protocols/Classic.html). W3.org. 1998-05-14. Retrieved 2010-08-01. list of other classic documents recounting the early protocol history
- HTTP 0.9 – As Implemented in 1991 (https://www.w3.org/Protocols/HTTP/AsImplemented.html)