Engineer

HTTP

⇄ the request and answer the web is built on

What it is

HTTP, the Hypertext Transfer Protocol, is the set of rules a browser, an app or a script follows to ask a server for something and receive a reply, and when that conversation runs inside an encrypted TLS connection the same protocol goes by HTTPS. Every exchange has two halves. The client sends a request: a method such as `GET` or `POST`, the target it wants, a list of header fields and sometimes a body. The server answers with a three-digit status code, header fields of its own and, usually, a body. Loading a page, fetching an image and one program calling another are all the same exchange repeated.

```
GET /pricing/ HTTP/1.1
Host: example.com
Accept: text/html

HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Cache-Control: max-age=600
```

The specification calls it stateless. Each request has to carry everything the server needs in order to understand it, and the protocol keeps no memory from one exchange to the next. Logins, baskets and sessions are built on top, usually with cookies: a header field the server sets once and the client sends back on every later request.

The vocabulary is small and precise. `GET` and `HEAD` are safe, meaning they ask for a representation and are expected to change nothing. `PUT` and `DELETE` are idempotent: sending the same one twice leaves the server where sending it once would have. `POST` promises neither. Status codes are grouped by their first digit,

2xx for success, 3xx for redirection, 4xx for a problem with the request and 5xx for a problem on the server, so a client meeting an unfamiliar code still knows roughly what happened.

That meaning has stayed put while the wire format has been replaced twice. HTTP/1.1 writes messages as lines of text over a TCP connection, and each connection returns its responses one at a time, in order. HTTP/2 keeps the same methods and fields but frames them in binary and interleaves many requests on one connection. HTTP/3 carries the same semantics over QUIC, a transport built on UDP, which avoids TCP's transport-level head-of-line blocking across independent streams. Since 2022 the IETF has published the semantics in one document and each version's wire format in another, which is why "what does 404 mean" and "how are the bytes laid out" now have separate answers.

HTTP versus HTTPS is therefore no contest between two protocols: HTTPS is HTTP with TLS underneath, which encrypts the exchange and proves the server's name. WebSocket starts life as an HTTP request and then upgrades the connection into a two-way channel, and gRPC is a convention for calling remote functions that runs over HTTP/2.

Use the concept in a decision

Put "What is HTTP?" to work in a real decision. We can help relate the concept to the system you are building or buying. Wi-Fi, firewalls and remote access set up so staff can work from anywhere and only the right people get in.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Managed 30 Ubuntu Linux VPS instances — 30 Linux servers managed, with recovery planned for

Managed 30 Ubuntu Linux VPS instances Get in touch →

Why it matters

Nearly every system that talks to another over the internet already speaks HTTP, so the decision that matters is rarely whether to use it. It is whether to use what it defines or to treat it as a pipe and rebuild the rest on top.

Using it fully pays off because a long chain of software already reads the vocabulary. A client library may retry a `GET` that timed out; it must leave a `POST` alone, and so must any proxy in between. A cache can keep a response marked `max-age=600` for ten minutes and serve it to the next reader without asking the server. Monitoring counts 5xx answers as errors without being told to.

An API that sends every call as a `POST` to one address and reports failure inside a 200 response gives all of that up, and every client then has to learn that API's private rules.

The model has edges. Request and answer suit a client that knows what it wants; they suit a server with news to push far less, which is why live updates usually travel over WebSocket, server-sent events or polling.

Plain HTTP on the public internet is a different question with a short answer. Browsers label it as insecure, anyone on the path can read and alter it, and certificates cost nothing, so public traffic belongs on HTTPS. The choice between HTTP/1.1, HTTP/2 and HTTP/3 is mostly a setting on the server or the edge in front of it, and application code stays the same because the meaning of every request is the same.

What it looks like in practice

The clearest case on this site is an API that was moved onto a framework built around HTTP's own shape. The HTTP API moved from Fiber to Huma v2, which runs on top of Fiber, so each endpoint is declared with input and output structs and the validation comes from the types rather than from checks written by hand. An OpenAPI description falls out of the same declarations, and a CI check compares the routes the server registers against the ones the description advertises. At 649 paths and 760 operations parity is 100%, held there because a route nobody described fails the build. Exactly two endpoints stayed on raw Fiber: the WebSocket ones, where Huma's request-and-response model does not fit.

The site itself uses content negotiation. Every page is published as HTML and as Markdown, and one way to get the Markdown is to send a request with `Accept: text/markdown` and let the server choose. The onion address on that page is plain `http://` on purpose. Because that address encodes the service's public key, Tor has already authenticated and encrypted the circuit by the time the first HTTP request travels over it.

HTTP also turns up as a tool in infrastructure code. When a second server was provisioned, both providers' firewalls were written directly against their REST APIs through a generic HTTP task instead of a vendor collection. That removed a dependency and its version drift, at the cost of writing each request shape by

hand. It has a further cost the page names: the generic HTTP task declares no check-mode support, so a dry run of the play that creates the machine skips every task in it, and the rehearsal is run against the firewall play instead.

Where it goes wrong

Most HTTP bugs come from using a word in the vocabulary to mean something other than what the rest of the chain hears.

An error that arrives as a success

An API answers 200 and puts `"error": "card declined"` in the body. Caches may store it, dashboards count it as a success, and retry logic in the client never fires. Send the status code that fits, 4xx or 5xx, and describe the problem in the body; RFC 9457 defines a JSON shape for that, served as `application/problem+json`.

A link that deletes things

A `GET` request changes state, for example an unsubscribe or delete link in an email. Crawlers, link previews in chat apps, security scanners and browser prefetching all follow links on their own, and each of them can trigger it. Anything that changes data belongs behind `POST`, `PUT` or `DELETE`, with a confirmation page if the link must exist.

A payment taken twice

The client sends a `POST`, the connection drops before the answer arrives, and the client tries again. The server had already processed the first one. HTTP gives the client no way to tell those cases apart, so the API has to: accept an idempotency key in a header and return the stored result for a repeated key, or let the client choose the new resource's identifier and create it with `PUT`.

A form that turns into a GET

A server answers a `POST` with 301 or 302 to send it elsewhere. For historical reasons the specification lets user agents switch the method to `GET` on those two codes, and many do, so the body is dropped on the way. When the method has to survive the redirect, use 307 or 308.

The wrong variant from a cache

One URL serves HTML or Markdown, English or Danish, depending on request header fields. A cache that has been told nothing stores whichever variant it saw first and hands it to the next reader. The response has to name the fields it depends on, for example `Vary: Accept`, so that a cache keeps one copy per variant instead of one per URL.

Further reading

- RFC 9110: HTTP Semantics — methods, status codes and header fields, defined once for every version.
- RFC 9111: HTTP Caching — when a stored response may be reused and what `Cache-Control` tells a cache.
- RFC 9112: HTTP/1.1 — the text wire format, connection handling and message framing.
- RFC 9113: HTTP/2 — binary frames and many streams over one connection.
- RFC 9114: HTTP/3 — the same semantics mapped onto QUIC streams.
- RFC 9457: Problem Details for HTTP APIs — a standard JSON body for describing what went wrong.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is HTTP?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/http/