HTTP and Web Caching

Version RFC 9110 · verified 2026-11-02

The HTTP protocol as browsers and servers see it — semantics, caching, negotiation, versions, TLS, CORS and security headers.

Contents

Mental model 3

Messages and headers 3

Methods 5

Status codes 6

Caching: freshness 7

Caching: validators and Vary 9

Negotiation and compression 11

Versions and TLS 12

CORS 13

Cookies and security headers 15

Further reading 17

Mental model

HTTP is a stateless request/response contract: every request carries everything the server needs, and headers on both sides say how the message may be cached, negotiated and trusted. HTTP/1.1, HTTP/2 and HTTP/3 change only the transport — framing, multiplexing, TCP or QUIC — while methods, status codes and caching rules stay identical. Learn the semantics once and they apply to all three.

Messages and headers

Part	Example
Request line	`GET /users/7 HTTP/1.1`
Status line	`HTTP/1.1 200 OK`
Header field	`Content-Type: application/json`
Empty line	Ends the header section
Content	Optional body

A message is a start line, header fields, an empty line and optional content. Field names are case-insensitive;

HTTP/2 and HTTP/3 send them lowercase. `Host` is mandatory in HTTP/1.1 because one IP serves many sites.

```
GET /api/users/7 HTTP/1.1
Host: api.example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 25

{"id":7,"name":"Ada"}
```

> **Tip:** `curl -v URL` prints the request (`>`) and response (`<`) headers; `curl -sI URL` sends `HEAD` and prints headers only.

Methods

Method	Safe	Idempotent
`GET`, `HEAD`	Yes	Yes
`OPTIONS`	Yes	Yes
`PUT`, `DELETE`	No	Yes
`POST`	No	No
`PATCH` (RFC 5789)	No	No

Safe means the client asks for no state change, so crawlers and prefetchers may call it freely. **Idempotent** means sending the request twice has the same intended effect as once, so clients and proxies may retry it automatically after a dropped connection.

```
curl -X PUT https://api.example.com/users/7 \
  -H 'Content-Type: application/json' \
  -d '{"name":"Ada"}'
```

> **Warning:** A `GET` that changes state (`/delete?id=7`) will be triggered by link prefetchers, crawlers and retries. Side effects belong on unsafe methods.

Status codes

Code	Use
`200`, `201`, `204`	OK; created; OK with no content
`301`, `308` / `302`, `307`	Permanent / temporary redirect
`304`	Not Modified — use your cached copy
`401` / `403`	Not authenticated / not allowed
`404`, `409`, `422`, `429`	Missing; conflict; invalid; slow down
`500`, `502`, `503`, `504`	Server; bad gateway; unavailable; timeout

The first digit is the class: 2xx success, 3xx redirect, 4xx client error, 5xx server error. Clients that do not know a code treat it as the `x00` of its class. `307` and `308` guarantee the method and body are reused on redirect; `301` and `302` let clients change `POST` to `GET`.

```
URL=https://example.com/
curl -s -o /dev/null -w '%{http_code}\n' "$URL"
curl -sL "$URL"   # follow redirects
```

> **Gotcha:** `401` means “who are you?” and must carry a `WWW-Authenticate` header; `403` means “I know who you are, and no”. Returning `401` for a permission failure makes clients retry login forever.

Caching: freshness

Directive	Meaning
`max-age=600`	Fresh for 600 s

Directive	Meaning
`s-maxage=600`	Same, shared caches only (CDN)
`no-cache`	Store, but revalidate before every use
`no-store`	Never store
`private` / `public`	Browser only / any cache
`stale-while-revalidate=60`	Serve stale while refreshing (RFC 5861)

A cached response is **fresh** while its age is below its freshness lifetime: `s-maxage` (shared caches), else `max-age`, else `Expires` minus `Date`. With none of these, caches may guess a **heuristic** lifetime; RFC 9111 suggests 10% of the time since `Last-Modified`. The `Age` header reports how long a response has sat in caches.

```
Cache-Control: public, max-age=60, s-maxage=600
Age: 45
```

```
# browser: fresh for 15 s more (60 - 45)
# CDN:     fresh for 555 s more (600 - 45)
```

> **Gotcha:** `no-cache` does not mean “don’t cache” — it means “always revalidate”. Use `no-store` for responses that must never be written to disk, such as account pages.

Caching: validators and Vary

Header	Role
`ETag: "v42"`	Opaque version of the representation
`If-None-Match: "v42"`	Send only if changed
`Last-Modified`	Timestamp validator
`If-Modified-Since`	Timestamp check
`Vary: Accept-Encoding`	Cache key includes this request header

When a stored response is stale, the cache **revalidates**: it sends the validator back and the server replies `304 Not Modified` with no body when nothing changed. `Vary` lists request headers that changed the response, so the cache keeps one entry per value.

```
URL=https://example.com/
etag=$(curl -sI "$URL" | tr -d '\r' \
  | awk -F': ' 'tolower($1)=="etag"{print $2}')
curl -s -o /dev/null -w '%{http_code}\n' \
  -H "If-None-Match: $etag" "$URL"   # 304
```

> **Gotcha:** A response personalised by cookie but cached with `public` and no `Vary: Cookie` serves one user's page to the next. Mark per-user responses `private` or `no-store`.

Negotiation and compression

Request	Response
`Accept: application/json`	`Content-Type`
`Accept-Encoding: gzip, br, zstd`	`Content-Encoding: br`
`Accept-Language: de, en;q=0.8`	`Content-Language`
`Range: bytes=0-1023`	`206 + Content-Range`

The client states preferences, with optional `q` weights from 0 to 1; the server picks one and should list the request header it used in `Vary`. Compression is negotiated per response: `gzip` everywhere, `br` and `zstd` where the client lists them.

```
URL=https://example.com/
curl -s --compressed -o /dev/null \
  -w '%{size_download}\n' "$URL"  # smaller
curl -s -o /dev/null \
  -w '%{size_download}\n' "$URL"  # full size
```

> **Note:** Do not compress images, video or fonts that are already compressed; you spend CPU for no saving.

Versions and TLS

Version	Transport	Key change
HTTP/1.1	TCP	One request at a time per connection
HTTP/2	TCP (TLS in browsers)	Multiplexed binary streams, HPACK
HTTP/3	QUIC (UDP)	Per-stream loss recovery, QPACK

HTTP/2 removes head-of-line blocking between requests, but one lost TCP packet still stalls every stream on the connection; QUIC fixes that by recovering loss per stream. Browsers negotiate HTTP/2 inside the TLS handshake via ALPN (`h2`) and discover HTTP/3 through the `Alt-Svc` header or the DNS `HTTPS` record

(RFC 9460). TLS 1.3 completes a full handshake in one round trip, and the server proves its name with a certificate chain to a trusted root.

```
URL=https://example.com/
curl -s -o /dev/null \
  -w '%{http_version}\n' "$URL"   # 2
curl -sv -o /dev/null "$URL" 2>&1 \
  | grep ALPN   # ALPN: server accepted h2
```

CORS

Header	Sent by	Meaning
`Origin`	Browser	Calling page's origin
`Access-Control-Allow-Origin`	Server	Who may read
`Access-Control-Allow-Credentials`	Server	Cookies allowed
`Access-Control-Allow-Methods`	Server	Preflight answer

Header	Sent by	Meaning
`Access-Control-Max-Age`	Server	Cache the preflight

CORS is the browser relaxing the same-origin policy on the server's say-so; it protects users, not your API, since `curl` ignores it. Requests other than simple `GET` / `POST` — custom headers, `application/json`, `PUT` — first send an `OPTIONS` **preflight**.

```
OPTIONS /api/orders HTTP/1.1
Origin: https://app.example.com
Access-Control-Request-Method: PUT

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.example.
com
Access-Control-Allow-Methods: GET, PUT
Access-Control-Allow-Credentials: true
Vary: Origin
```

> **Gotcha:** With `credentials: "include"`, the browser rejects `Access-Control-Allow-Origin: *`. Echo the exact allowed origin, and add `Vary: Origin` so caches don't serve it to another origin.

Cookies and security headers

Header	Purpose
`Set-Cookie: ...; Secure; HttpOnly`	HTTPS only; hidden from JS
`SameSite=Lax` / `Strict` / `None`	Cross-site sending rules
`__Host-` name prefix	Secure, `Path=/`, no `Domain`
`Strict-Transport-Security`	HTTPS only, for `max-age` seconds
`Content-Security-Policy`	Allowed script and resource sources
`X-Content-Type-Options: nosniff`	No MIME sniffing

Cookies are how state rides on a stateless protocol. `SameSite=None` requires `Secure`; some browsers default to `Lax` when the attribute is missing. HSTS tells the browser to use HTTPS for the host until `max-age` expires, and CSP limits where scripts may load from, which blunts XSS.

```
# wrapped for width: one line each on the wire
Set-Cookie: __Host-sid=abc; Path=/; Secure;
  HttpOnly; SameSite=Lax
Strict-Transport-Security: max-age=31536000;
  includeSubDomains
Content-Security-Policy: default-src 'self';
  script-src 'self'
```

> **Gotcha:** Browsers ignore `Strict-Transport-Security` received over plain HTTP. The first visit is unprotected unless the domain is on the HSTS preload list (`preload`, `max-age` of at least a year, `includeSubDomains`).

Further reading

- RFC 9110 — HTTP Semantics
- RFC 9111 — HTTP Caching
- RFC 9114 — HTTP/3
- MDN — HTTP caching
- MDN — Cross-Origin Resource Sharing
- MDN — Set-Cookie