HTTP Course

Flavio Copes

Updated August 3, 2026

Contents

Learn how HTTP requests and responses work, including methods, status codes, headers, caching, cookies, security, and modern versions.

What you'll learn: Inspect an HTTP exchange and reason about its method, status, headers, body, and browser behavior.

From a URL to a response

DNS, connections, clients, servers, requests, and responses.

What HTTP does

Understand the simple request and response convention that lets browsers, servers, command-line tools, and APIs communicate.

HTTP is the language clients and servers use to exchange messages on the Web.

A client sends a **request**. A server handles it and sends a **response**.

```
client  -- request -->  server
client  <-- response -- server
```

Your browser is an HTTP client, but it is not the only one. `curl`, mobile apps, search-engine crawlers, and servers can all make HTTP requests.

HTTP does not decide how a server builds its response. The server might read a file, query a database, call another service, or calculate a value. HTTP only gives both sides a predictable message format.

This separation is important. You can replace the browser, server language, or database without changing the basic request and response cycle.

Clients and servers

See the distinct jobs of an HTTP client and server, and why either side can be implemented with many different tools.

The client starts the exchange. It chooses a URL, builds a request, and sends it.

The server listens for requests. For each one, it decides what should happen and builds a response.

Imagine requesting `/articles/hello`:

1. The browser sends a request for that path.
2. The server matches the path to a route.
3. The route loads the article.
4. The server responds with HTML and a status code.

The server does not have to return a file named `hello.html`. A route can generate the response dynamically.

One program can play both roles. Your application server may receive a browser request, then become a client when it requests data from another API.

Think in terms of each exchange: who starts it is the client; who answers it is the server.

Read a URL

Break a URL into its scheme, host, port, path, query string, and fragment so you know what each part controls.

Consider this URL:

`https://example.com:443/articles/http?format=short#headers`

It contains several parts:

- `https` is the scheme.
- `example.com` is the host.
- `443` is the port.
- `/articles/http` is the path.
- `format=short` is the query string.
- `headers` is the fragment.

The browser uses the scheme, host, and port to connect. It sends the path and query string to the server as the request target.

The fragment is different. It identifies a place inside the returned document and is normally handled by the browser. It is not sent in the HTTP request.

Default ports are usually omitted: 80 for HTTP and 443 for HTTPS.

Before the request

Follow DNS lookup and connection setup to see what must happen before a browser can send its first HTTP message.

Typing a URL does not send an HTTP request immediately.

First, the browser needs an IP address. DNS translates a hostname such as `example.com` into an address the network can reach.

Next, the browser opens a connection. Traditional HTTP/1.1 and HTTP/2 use TCP. HTTPS also performs a TLS handshake so the connection is encrypted and the server can prove its identity.

Only then can the browser send the HTTP request.

The exact sequence can be faster than it sounds. Browsers cache DNS results, reuse existing connections, and negotiate modern protocols. Still, keeping these layers separate helps when debugging:

- DNS answers **where** the host is.
- the transport connection carries bytes.
- TLS protects those bytes.
- HTTP defines the messages exchanged.

Inspect the request cycle

Use the browser Network panel to identify the URL, method, status, resource type, and timing of a real HTTP exchange.

Open a page, then open your browser developer tools and select **Network**. Reload the page so the panel records every request.

Select the main document request and find:

- the complete request URL
- the request method
- the response status
- the remote address
- the protocol version
- the time spent waiting and downloading

Now look at the other rows. One HTML response can cause the browser to request stylesheets, scripts, images, and fonts. Each resource has its own HTTP exchange.

Try filtering by `Doc`, `CSS`, and `Img`. This turns an abstract request cycle into something visible.

Take this interactive quiz online

Requests

Methods, paths, query strings, headers, and request bodies.

Anatomy of a request

Read the request line, headers, blank line, and optional body that make up a complete HTTP request message.

An HTTP/1.1 request can be shown as plain text:

```
POST /notes HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 24

{"title":"Learn HTTP"}
```

The first line contains the method, request target, and protocol version.

Headers follow as name-value pairs. They add context about the host, accepted formats, authentication, caching, and the body.

An empty line marks the end of the headers. A body may follow. `POST`, `PUT`, and `PATCH` often carry one. `GET` normally does not.

HTTP/2 and HTTP/3 encode messages differently on the wire, but the same concepts remain: method, target, headers, and an optional body.

HTTP methods

Choose GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS according to the action a request asks the server to perform.

The method describes the request's intent.

- `GET` retrieves a resource.
- `POST` submits data or starts an action.
- `PUT` replaces a resource.
- `PATCH` updates part of a resource.

- `DELETE` removes a resource.
- `HEAD` asks for the headers a `GET` would return, without the body.
- `OPTIONS` asks which communication options are available.

For a notes API, you might use:

```
GET /notes
POST /notes
GET /notes/42
PATCH /notes/42
DELETE /notes/42
```

HTTP does not automatically enforce what a method means. A server could delete data in response to `GET`, but that would violate expectations and cause problems for caches, crawlers, and browsers.

Safe and idempotent methods

Understand the guarantees behind safe and idempotent methods so retries, links, caches, and automated clients behave predictably.

A **safe** method asks only to read information. `GET`, `HEAD`, and `OPTIONS` are safe. They should not request a change to server state.

An **idempotent** method has the same intended effect whether it is sent once or several times.

`PUT` is idempotent: sending the same complete replacement twice leaves the resource in the same state. `DELETE` is also idempotent: after the first deletion, repeating it does not delete another resource.

`POST` is usually not idempotent. Repeating a purchase request might create two orders.

This matters when a connection fails. A client can often retry an idempotent request safely. Retrying a non-idempotent request may need an idempotency key or application-specific protection.

Query strings and request bodies

Decide whether request data belongs in the URL query string or in the message body, and understand how servers receive each form.

A query string is part of the URL:

```
/articles?tag=css&page=2
```

It works well for filters, searches, sorting, and pagination. Query parameters are visible in the address bar and easy to bookmark or share.

A request body carries data after the headers:

```
{
  "title": "Learn HTTP",
  "published": true
}
```

Bodies are common with `POST`, `PUT`, and `PATCH`. The `Content-Type` header tells the server how the body is encoded.

Do not put secrets in a query string. URLs often appear in browser history, logs, analytics, and referrer data. HTTPS encrypts a URL while it travels, but it does not stop these local and application-

level records.

Send a request with curl

Build GET and POST requests from the command line so you can control methods, headers, and bodies without a browser interface.

`curl` is an HTTP client you can run in a terminal.

Request a page and show response headers:

```
curl -i https://example.com/
```

Send JSON with `POST`:

```
curl https://example.com/notes \
  -X POST \
  -H 'Content-Type: application/json' \
  -d '{"title":"Learn HTTP"}'
```

`-X` chooses the method, `-H` adds a header, and `-d` supplies a body. When you use `-d`, curl chooses `POST` by default, so `-X POST` is optional here.

Add `-v` for connection and request details. It is noisy, but useful when a redirect, TLS handshake, or request header is not behaving as expected.

Take this interactive quiz online

Responses

Status codes, headers, bodies, redirects, and failures.

Anatomy of a response

Read the status line, response headers, blank line, and optional body that a server sends back to an HTTP client.

An HTTP response has a status, headers, and usually a body.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 18

<h1>Hello</h1>
```

The first line contains the protocol version, numeric status code, and a short reason phrase.

Headers describe the response and how clients should handle it. The empty line separates the headers from the body.

The body can contain HTML, JSON, an image, a video, or any other representation. Some responses intentionally have no body. A `204 No Content` response is one example; a response to `HEAD` is another.

Status code families

Use the first digit of an HTTP status code to quickly distinguish information, success, redirection, client errors, and server errors.

HTTP status codes are grouped into five families:

- `1xx`: the exchange is still in progress.
- `2xx`: the request succeeded.
- `3xx`: another location or a cached response is involved.
- `4xx`: the request cannot be fulfilled as sent.
- `5xx`: the server failed while handling a valid request.

The family gives you a useful first diagnosis. A `404` and a `401` mean different things, but both tell you to investigate the request or client context. A `500` points you toward server logs instead.

Do not treat every non-`200` response as failure. `201`, `204`, `301`, and `304` can all be correct outcomes.

Success and redirects

Choose common 2xx and 3xx status codes and understand how browsers follow Location headers or reuse cached responses.

Common success codes include:

- `200 OK`: a general successful response.
- `201 Created`: a new resource was created.
- `204 No Content`: the request succeeded without a response body.

Redirects normally include a `Location` header:

```
HTTP/1.1 301 Moved Permanently
Location: https://example.com/new-page
```

`301` and `308` are permanent redirects. `302` and `307` are temporary. `307` and `308` explicitly preserve the request method; historical client behavior around `301` and `302` may turn a `POST` into a `GET`.

`304 Not Modified` belongs to the 3xx family but is not a normal redirect. It tells a client that its cached response is still valid.

Client errors

Distinguish bad input, missing authentication, forbidden access, missing resources, conflicts, and rate limits using common 4xx responses.

Use a 4xx response when the server cannot fulfill the request as sent.

- `400 Bad Request`: invalid syntax or input.
- `401 Unauthorized`: authentication is missing or invalid.
- `403 Forbidden`: the user is known but not allowed.
- `404 Not Found`: the target resource does not exist.
- `409 Conflict`: the request conflicts with current state.
- `422 Unprocessable Content`: the body is understood but fails semantic validation.
- `429 Too Many Requests`: the client exceeded a rate limit.

A useful error response explains what the client can fix. An API might return JSON with an error code and field-level details, while an HTML site may render a helpful page.

Do not leak secrets, stack traces, or database details in a public error body.

Server errors

Recognize internal failures, bad gateways, unavailable services, and gateway timeouts while keeping useful diagnostics out of public responses.

A 5xx status means the server failed to complete a request it should have been able to handle.

- `500 Internal Server Error`: an unexpected application failure.
- `502 Bad Gateway`: a gateway received a bad upstream response.
- `503 Service Unavailable`: the service is overloaded or temporarily down.
- `504 Gateway Timeout`: an upstream service did not answer in time.

The response shown to a visitor should be calm and generic. Detailed diagnostics belong in server logs, connected by a request ID when possible.

When debugging, check the exact status, response body, server logs, and whether another service sits between client and application. A gateway error may not come from your application process at all.

Take this interactive quiz online

Headers and representations

Describe and negotiate the content exchanged over HTTP.

Request headers

Read common request headers that identify the target host, client preferences, body format, authorization, and previous page.

Request headers add context to a request.

```
GET /articles HTTP/1.1
Host: example.com
Accept: text/html
Accept-Language: en
User-Agent: Mozilla/5.0 ...
```

`Host` identifies the requested hostname. This lets one server address host several websites.

`Accept` and `Accept-Language` describe formats and languages the client can handle. `Content-Type` describes a body the client is sending.

`Authorization` carries credentials such as a bearer token. `Cookie` carries cookies that match the URL. `Referer` can identify the page that led to the request, subject to browser policy.

Headers are not automatically trustworthy. A custom client can send a fake `User-Agent`, `Referer`, or forwarding header. Use cryptographic verification and trusted infrastructure for security decisions.

Response headers

Read response headers that describe body content, redirects, caching, cookies, and the server's instructions to the client.

Response headers describe the returned representation and how to handle it.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Cache-Control: max-age=300
```

`Content-Type` describes the body. `Content-Encoding` says the body was compressed. `Content-Length` gives its byte length when known.

`Location` points to another URL, usually with a redirect or `201 Created` response. `Set-Cookie` asks a browser to store a cookie. `Cache-Control`, `ETag`, and `Last-Modified` guide caches.

Security headers can restrict what a browser may do with the response. We will cover those later.

Inspect headers in the Network panel instead of guessing. A server, reverse proxy, CDN, or hosting platform may add or replace them.

Content types and encodings

Use Content-Type, charset, and Content-Encoding to describe a message body accurately and avoid parsing or text-decoding errors.

`Content-Type` tells the recipient what the body contains:

```
Content-Type: application/json
Content-Type: text/html; charset=utf-8
Content-Type: image/webp
```

The media type determines which parser or renderer should handle the bytes. A JSON body sent as `text/plain` may still look readable, but clients cannot reliably treat it as JSON.

The `charset` parameter describes text encoding. UTF-8 is the normal choice for web text.

`Content-Encoding` is different. It describes a transformation applied for transport, such as `gzip` or `br`. The recipient decodes that layer and then interprets the result according to `Content-Type`.

In short: type says **what it is**; encoding says **what happened to it on the way**.

Content negotiation

See how Accept headers let a client express preferences and how a server selects an available representation of a resource.

A resource can have more than one representation. The client states preferences with `Accept` headers:

```
Accept: text/html, application/json;q=0.8
Accept-Language: en, it;q=0.7
Accept-Encoding: br, gzip
```

The server chooses a supported option. Quality values such as `q=0.8` make one option less preferred than another.

The server is not required to support every requested representation. An API may always return JSON. A site may use distinct URLs instead of negotiating language.

When a response changes according to a request header, caches need to know. A response such as

`Vary: Accept-Encoding` tells a cache to keep different variants for different encoding preferences.

Take this interactive quiz online

Caching

Freshness, validators, conditional requests, and cache scope.

Why HTTP caching matters

Understand how browser and shared caches reduce repeated downloads, latency, bandwidth use, and work performed by the origin server.

A cache stores a response so it can satisfy a later request without downloading the full body again.

Browsers have private caches for one user. CDNs and proxies can provide shared caches for many users. Your origin server may also cache application work, but HTTP caching specifically concerns reusable responses.

Caching can make a site faster and reduce bandwidth and server load. It can also serve stale or private data when configured badly.

Every cache decision asks two questions:

1. May this response be stored?
2. May the stored response be reused now?

`Cache-Control` answers much of this. Validators such as `ETag` let a stale cache check whether its stored response still matches the current resource.

Cache-Control

Set freshness and storage rules with max-age, public, private, no-cache, no-store, and immutable response directives.

`Cache-Control` contains directives for caches.

```
Cache-Control: public, max-age=3600
```

This response can be stored by shared caches and remains fresh for one hour.

Useful directives include:

- `private`: only a private cache should store it.
- `no-store`: do not store the response.
- `no-cache`: storage is allowed, but validate before reuse.
- `max-age=60`: fresh for 60 seconds.
- `immutable`: the URL's content will not change while fresh.

`no-cache` is often misunderstood. It does not prohibit caching. It requires a check before reuse.

Versioned assets such as `/app.a1b2.css` can use a long lifetime and `immutable`. When the file changes, its URL changes too. Personalized account pages usually need `private` or `no-store`.

Validators and conditional requests

Revalidate stale responses with ETag or Last-Modified and understand why a 304 response can save a full body download.

When a cached response becomes stale, the client can ask whether it changed.

An `ETag` identifies a version of a representation:

```
ETag: "article-v5"
```

The next request can include:

```
If-None-Match: "article-v5"
```

If the representation is unchanged, the server returns `304 Not Modified` without the full body. The client reuses its stored copy.

`Last-Modified` and `If-Modified-Since` perform a similar check using a timestamp. ETags can be more precise because the server controls how versions are identified.

Freshness avoids a network request. Validation still contacts the server, but can avoid transferring and parsing the body.

Debug a cached response

Inspect freshness, validators, age, and transferred size to explain whether a response came from a browser cache, CDN, or origin.

Open the Network panel and reload a page twice. Select the same static resource after each load.

Look for:

- `Cache-Control` and its freshness lifetime
- `ETag` or `Last-Modified`
- `Age`, which may reveal time spent in a shared cache
- a `304` status after conditional validation
- "memory cache" or "disk cache" in the transferred-size column

Developer tools often include a **Disable cache** option. It works while the tools are open and is useful for comparing cached and uncached behavior.

Be careful when testing. A hard reload, normal reload, disabled cache, and private window can produce different results. Record exactly which one you used.

Take this interactive quiz online

State and browser security

Cookies, authorization, origins, CORS, and security headers.

Cookies add state

Follow a cookie from Set-Cookie to later Cookie request headers and see how state is layered on top of stateless HTTP exchanges.

HTTP does not automatically connect one request to the next. Cookies add a small piece of browser-managed state.

The server sets a cookie in a response:

```
Set-Cookie: session=abc123; Path=/; HttpOnly; Secure; SameSite=Lax
```

The browser stores it and sends it on matching later requests:

```
Cookie: session=abc123
```

`Path` and `Domain` control where a cookie is sent. `Max-Age` or `Expires` can make it persistent; otherwise it is normally a session cookie.

`HttpOnly` prevents JavaScript from reading it. `Secure` limits it to HTTPS. `SameSite` restricts when it is sent with cross-site requests.

Cookies are attached automatically, so keep them small and scope them carefully.

Sessions and bearer tokens

Compare server-side sessions with bearer-token authentication and understand how each credential is carried and verified.

With a server-side session, the server stores session data and gives the browser an identifier, usually in a cookie. Each request sends that ID so the server can find the session.

A bearer token is often sent explicitly:

```
Authorization: Bearer eyJhbGciOi...
```

The server verifies the token and determines what the caller may do. A token can contain signed claims or be an opaque reference to server-side data.

"Bearer" means possession is enough to use it. Protect session IDs and tokens like passwords. Use HTTPS, avoid placing them in URLs, limit their lifetime and permissions, and provide a way to revoke or rotate them.

Authentication proves who is making the request. Authorization decides whether that identity may perform the requested action.

Origins and CORS

Understand the browser same-origin policy, what makes two URLs different origins, and how servers opt in to cross-origin reads with CORS.

An origin is the combination of scheme, host, and port. These are different origins:

```
https://example.com
https://api.example.com
http://example.com
https://example.com:8443
```

The same-origin policy stops browser scripts from freely reading data from another origin.

CORS lets a server relax that rule with response headers:

```
Access-Control-Allow-Origin: https://app.example.com
```

For some requests, the browser first sends an `OPTIONS` preflight. It asks whether the server permits the intended method and headers.

CORS is enforced by browsers. It does not prevent another server or `curl` from sending requests.

It also does not replace authentication or authorization.

Browser security headers

Add response headers that constrain scripts, framing, MIME sniffing, referrer data, and future connections made by the browser.

Security headers tell a browser which capabilities to allow.

- `Content-Security-Policy` restricts where scripts, styles, images, and other resources may load from.
- `X-Content-Type-Options: nosniff` tells the browser to respect declared content types.
- `Referrer-Policy` limits referrer information sent to other sites.
- `Permissions-Policy` controls selected browser features.
- `frame-ancestors` inside CSP controls which pages may embed yours.

A strict policy can break an existing site, so deploy carefully. Start by inventorying required sources. CSP can run in report-only mode while you observe violations.

Headers reduce risk but do not repair unsafe application code. You still need output escaping, input validation, secure authentication, and current dependencies.

Take this interactive quiz online

HTTPS and modern HTTP

TLS, HTTP/2, HTTP/3, and practical protocol debugging.

HTTPS and TLS

Understand how TLS adds encryption, integrity, and server authentication while leaving familiar HTTP methods and messages intact.

HTTPS is HTTP carried through a TLS-protected connection.

TLS provides three important properties:

- encryption keeps message contents private in transit
- integrity detects tampering
- server authentication lets the client verify the certificate for the hostname

The HTTP concepts stay familiar. You still use URLs, methods, headers, status codes, and bodies.

HTTPS does not prove a site is honest or bug-free. It protects the connection to the authenticated host. Application security remains your responsibility.

Use HTTPS everywhere, not only on login pages. An unprotected page can be modified to steal credentials before a visitor reaches the secure form.

From HTTP/1.1 to HTTP/2

Compare reusable HTTP/1.1 connections with HTTP/2 binary framing, multiplexed streams, and compressed request and response headers.

HTTP/1.1 can reuse a TCP connection with keep-alive, but responses on one connection arrive in request order. Browsers historically opened several connections per origin to fetch resources in

parallel.

HTTP/2 represents messages as binary frames. Several independent streams share one connection, so a slow application response does not have to block every response behind it.

HTTP/2 also compresses headers, which helps when many requests repeat the same cookies and metadata.

The application semantics do not change. A `GET` is still a `GET`; a `404` is still a `404`. Browsers and servers negotiate the protocol, usually without changes to your routes.

Because every stream still shares one ordered TCP connection, packet loss can temporarily pause them all. HTTP/3 addresses that transport-level limitation.

HTTP/3 and QUIC

See why HTTP/3 uses QUIC over UDP and how independent streams reduce transport-level blocking when packets are lost.

HTTP/3 maps HTTP onto QUIC instead of TCP.

QUIC runs over UDP but implements reliable delivery, encryption, and connection management itself. Its streams are independent. If a packet for one stream is lost, unrelated streams can continue.

QUIC includes TLS security and can reduce connection setup work. It can also keep a connection alive when a device changes networks, because the connection is not identified only by one IP address and port pair.

You normally do not rewrite application code for HTTP/3. Your hosting platform, CDN, server, and browser negotiate it. Methods, status codes, URLs, and headers remain the same.

Use the Network panel's protocol column to see which version a request used.

Final HTTP inspection

Bring the course together by explaining a real document request, its security and caching policy, and the additional resources it triggers.

Open the Network panel on a site you use and select its main document request. Write down:

1. URL, method, and protocol version
2. status code and any redirect chain
3. request and response content types
4. cache freshness and validators
5. cookies sent or set
6. CORS and security headers
7. transferred size and timing

Then choose one stylesheet or image requested by the document and repeat the inspection. Explain why its caching policy differs from the HTML document.

Finally, reproduce a safe public request with `curl -I` or `curl -i`. Compare its headers with the browser's. Differences may come from cookies, content negotiation, compression, or an intermediary cache.

You now have the vocabulary to explain an HTTP exchange instead of treating the Network panel

as a wall of data.

Take this interactive quiz online

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/http/.