Engineer

WebSocket

↔ a protocol for a two-way connection that stays open

What it is

WebSocket is a protocol for a connection both sides can write to at any moment. An HTTP handshake establishes it, and it stays open until a peer closes it or the connection fails. The IETF published it as RFC 6455 in December 2011, and the part a browser exposes to scripts, the `WebSocket` object, is defined by the WHATWG WebSockets Standard. Plenty of code and documentation spells it “websocket” or “web socket”, and the plural WebSockets usually means the technology as a whole.

In the classic HTTP/1.1 handshake, the connection starts with an ordinary `GET`. The client adds `Upgrade: websocket` and `Connection: Upgrade`, a `Sec-WebSocket-Version` of 13 and a `Sec-WebSocket-Key`, which is a random 16-byte value encoded in base64. A server that agrees answers with status 101 Switching Protocols and a `Sec-WebSocket-Accept` header. That value proves the server read this very handshake rather than replaying an old answer: it is the client’s key with a fixed GUID appended, hashed with SHA-1 and encoded in base64. From that moment HTTP is finished on that connection, and both sides exchange frames instead. HTTP/2 and HTTP/3 instead carry WebSocket over one stream using extended CONNECT, as RFC 8441 and RFC 9220 specify.

A frame is small. Its header starts at two bytes, grows by two or eight more when the payload is longer than 125 bytes, and by another four on frames from the client, which carry a masking key. Each frame carries an opcode: text, binary, or continuation for a message split across several frames, plus three control frames for close, ping and pong. A complete text message must be valid UTF-8; an individual fragment can split a UTF-8 sequence. Every frame a client sends is masked with a fresh random 32-bit key, and a server never masks. Masking is not encryption. It exists so that a hostile script cannot choose the exact bytes that cross the network and use them to poison a caching proxy somewhere along the path.

Addresses use the `ws://` and `wss://` schemes, with port 80 and port 443 as the defaults, and `wss` runs the whole exchange inside TLS. Those are the ports HTTP already uses, on purpose: the handshake looks like any other web request to the firewalls and proxies it passes through.

Use the concept in a decision

Put “What is WebSocket?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

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. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

With short HTTP responses, the server speaks only when spoken to. Plenty of products need the opposite: a chat message, a price that just moved, a cursor in a shared document, a background job that has finished. Without a channel the server can write to, the client has to poll. It asks every few seconds, most answers only say that nothing has changed, each one carries a full set of headers, and news waits up to one interval before anyone sees it. An open WebSocket connection removes both costs. The server can send when something happens, without waiting for the next poll. Each message needs only a few bytes of framing instead of a full set of HTTP headers.

The price is state. An HTTP request can land on any server behind a load balancer, while a WebSocket connection belongs to one process for as long as it lives. That process holds memory for every open connection, a deploy or restart cuts all of them at once, and every proxy and load balancer on the path has to support the handshake and put up with a connection that may stay quiet for minutes. Running more than one server raises a question that plain HTTP never asks: when something happens on server A, how does it reach a client whose connection lives on server B? The usual answer is a shared publish-and-subscribe channel that every server listens to, and that is one more system to operate.

So it is a choice, not a default. When only the server has news to send, server-sent events, defined in the HTML Standard, push events over an ordinary HTTP response of type `text/event-stream`, and the browser reconnects by itself when

the response breaks off. WebSocket earns its extra moving parts when the client has as much to say as the server, as in chat, games and collaborative editing.

What it looks like in practice

The exchange below opens a chat connection. The key is the sample value from RFC 6455 itself, and the accept value is the one the standard prints, which every correct server must send back for that key.

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Origin: https://example.com

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

After the blank line that ends the server’s answer, nothing on the connection is HTTP any more. A browser does the whole handshake behind one constructor call and hands the rest to event listeners:

```
const socket = new WebSocket("wss://example.com/chat");
socket.addEventListener("open", () => socket.send("hello"));
socket.addEventListener("message", (event) => console.log(event.data));
socket.addEventListener("close", (event) => {
  console.log(event.code, event.reason, event.wasClean);
});
```

The close event says how the connection ended. Code 1000 is a normal closure and 1001 means a peer is going away, such as a server shutting down or a browser leaving the page. Code 1006 means the connection dropped without any close frame at all. No endpoint may send it; it exists only so the program on top can tell an abnormal end from a clean one.

The line between request-and-answer and a socket shows up clearly in one project in this site's portfolio. When an API moved onto Huma v2, every route went onto Huma's request and response model except exactly two: the WebSocket endpoints stayed on raw Fiber, because there the code genuinely wants the socket itself and Huma's model does not fit. A framework built around HTTP's own shape describes and validates requests and answers well, and a long-lived two-way connection is simply a different thing.

Where it goes wrong

A page on another site opens a connection with your visitors' cookies. The browser makes the handshake request to the server's host and can include its cookies, subject to the browser's cookie policies. Where those policies allow authenticated cross-site requests, a cookie-authenticated server that ignores the `Origin` header can accept a connection opened by another site's page as the visitor. RFC 6455 says a server meant only for certain sites should check `Origin` against the ones it expects and refuse the rest with 403 Forbidden. Do it in the handshake, before accepting the connection.

The access token ends up in the logs. The browser's `WebSocket` constructor takes a URL and an optional list of subprotocols, and nothing else, so a script cannot add an `Authorization` header. The quick way round is a token in the URL, where reverse proxies and access logs write it down. The RFC leaves authentication open and names cookies, HTTP authentication and TLS authentication as options. A cookie with the `Origin` check above works, and so does a short-lived, single-use ticket that the page fetches over an authenticated HTTP call and spends on the handshake.

A restart turns every client into a flood. An abrupt deploy or restart can close all connections at once. Clients that receive no close frame report code 1006; if they all reconnect immediately and keep trying, the server comes back up to what the RFC likens to a denial-of-service attack. RFC 6455 asks clients to wait a random time before the first attempt, suggesting anything between 0 and 5 seconds, and to wait longer after each failure, for example with truncated binary exponential backoff. A server that closes its connections with 1001 before it stops at least tells clients the break was planned.

```
function reconnectDelay(attempt) {
  if (attempt === 0) return Math.random() * 5000;
  const ceiling = Math.min(60_000, 1000 * 2 ** attempt);
```

```
  return Math.random() * ceiling;
}
```

Connections die and nobody notices. A connection that carries no traffic gives neither side any sign that the path still exists, and a peer that vanished without a close frame looks exactly like a quiet one until the next write fails. Ping and pong exist for this: an endpoint must answer a ping with a pong unless it has already received a close frame. When several pings are unanswered, it may answer only the latest. The RFC notes that a ping can serve as a keepalive or as a check that the other side still responds. The browser API offers no way to send a ping, so the timer belongs on the server: ping each connection at a fixed interval and close the ones that miss their pong.

One message eats the server's memory. The length field in a frame header leaves room for payloads far larger than any server has memory for, and the RFC warns that a hostile peer may send one enormous frame or an endless run of small fragments. Implementations must protect themselves, so set a maximum message size in the server library and close with code 1009, message too big, when a peer exceeds it. The same backlog builds in the other direction. In the browser, `bufferedAmount` reports the bytes `send()` has queued that are still waiting to go out on the network, and the WHATWG example sends its next update only when that number is zero. On the server, a client that reads slowly fills the outgoing buffer in the same way, so each connection needs a cap on it and a rule for what happens when the cap is reached.

Further reading

- RFC 6455: The WebSocket Protocol — the handshake, framing, masking and close codes, and the security reasoning behind each of them.
- WebSockets Standard — the browser API: the constructor, `send()`, `close()`, `bufferedAmount` and the close event.
- RFC 9110: HTTP Semantics — the `Upgrade` header field and the 101 Switching Protocols status the handshake relies on.
- Server-sent events — the HTML Standard's one-way alternative, with reconnection built in.
- RFC 8441: Bootstrapping WebSockets with HTTP/2 — extended CONNECT carries WebSocket on one multiplexed HTTP/2 stream.
- RFC 9220: Bootstrapping WebSockets with HTTP/3 — the corresponding extended-CONNECT mechanism for HTTP/3.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is WebSocket?” 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/websocket/