Real-time: WebSockets and SSE

Version RFC 6455 · verified 2026-11-02

Server-sent events vs WebSockets — wire formats, heartbeats, reconnection, backpressure, auth on upgrade, scaling and SignalR.

Contents

Mental model 2

Choosing a channel 2

SSE: format and EventSource 3

SSE: the server side 4

WebSocket: handshake and frames 5

Heartbeats 7

Reconnection and backpressure 8

Auth on the upgrade 9

Scaling and serverless 10

SignalR on ASP.NET Core 12

Further reading 13

Mental model

Server-sent events (SSE) are one HTTP response that never ends: the server keeps writing, the browser keeps reading and reconnects on its own. A WebSocket upgrades an HTTP request into a raw two-way message socket that you must keep alive, reconnect and authenticate yourself. Pick by direction first, then by what sits between you and the client: proxies, load balancers and serverless platforms.

Choosing a channel

Channel	Direction	Pick when
SSE	Server to client	Feeds, LLM token streams, progress
WebSocket	Both ways	Chat, collaboration, games
Long polling	Server to client	Nothing else gets through
WebTransport	Both, over HTTP/3	Backpressure or datagrams needed

SSE is plain HTTP, so caches, proxies, HTTP/2 multiplexing and cookie auth all just work, and the browser handles reconnection. Choose a WebSocket only when the client also sends frequent messages; a normal `fetch POST` beside an SSE stream covers occasional upstream traffic.

```
client -> server rarely?  SSE + fetch POST
both ways, high rate?     WebSocket
lossy, latency-critical?  WebTransport datagrams
```

> **Note:** WebTransport runs only over HTTP/3. Check current browser support on MDN before you depend on it.

SSE: format and EventSource

Field	Meaning
`data: ...`	Payload line; repeat for multi-line
`event: price`	Dispatch as a named event
`id: 42`	Sent back as `Last-Event-ID` on reconnect
`retry: 5000`	Reconnection delay in ms
`: comment`	Ignored; use as a heartbeat

The response has `Content-Type: text/event-stream`; a blank line ends each event. When the connection drops, `EventSource` waits the reconnection time and reconnects with the last `id` in a `Last-Event-ID` header, so the server can resume. Reply `204 No Content` to stop it reconnecting.

```
const es = new EventSource("/events");

es.addEventListener("price", (e) => {
  const tick = JSON.parse(e.data);
  console.log(tick.symbol, e.lastEventId);
});
es.onerror = () => {
  // readyState CONNECTING: retrying itself
```

```
  if (es.readyState === EventSource.CLOSED) {
    console.log("gave up");
  }
};
```

> **Gotcha:** Over HTTP/1.1, browsers allow 6 connections per domain across all tabs, and each open `EventSource` holds one. The seventh tab hangs. Serve SSE over HTTP/2, where the limit is negotiated streams (default 100).

SSE: the server side

Must do	Why
`Content-Type: text/event-stream`	Otherwise `EventSource` fails
Flush every event	Buffered bytes never arrive
Comment line every ~15 s	Proxies drop idle connections
Honour `Last-Event-ID`	Resume without gaps
Clean up on `close`	Timers leak per client

Any server that can write and flush a chunked response can serve SSE. The WHATWG spec recommends a comment line roughly every 15 seconds because some proxies close idle HTTP connections.

```
import { createServer } from "node:http";

createServer((req, res) => {
```

```
  res.writeHead(200, {
    "Content-Type": "text/event-stream",
    "Cache-Control": "no-cache",
  });
  const last = req.headers["last-event-id"];
  let id = Number(last ?? 0);
  const t = setInterval(() => res.write(
    `id: ${++id}\ndata: ${id}\n\n`), 1000);
  req.on("close", () => clearInterval(t));
}).listen(8080);
```

> **Gotcha:** Any layer that buffers the response — compression middleware, a reverse proxy — holds events until its buffer fills, so the stream arrives in bursts or not at all. Turn buffering off for the SSE route.

WebSocket: handshake and frames

Piece	Value
Request	`GET` with `Upgrade: websocket`
Response	`101 Switching Protocols`
`Sec-WebSocket-Accept`	base64(SHA-1(key + GUID))
Data opcodes	`0x1` text (UTF-8), `0x2` binary
Control opcodes	`0x8` close, `0x9` ping, `0xA` pong

The handshake is an ordinary HTTP/1.1 request, so it carries cookies and an `Origin` header. After `101`, both sides exchange frames; frames from the client are always masked. The accept value proves the server understood the WebSocket handshake.

```
import { createHash } from "node:crypto";

const GUID =
  "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
const accept = (key: string) =>
  createHash("sha1")
    .update(key + GUID)
    .digest("base64");

const key = "dGhlIHNhbXBsZSBub25jZQ==";
console.log(accept(key));
// s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

> **Gotcha:** `ws.close(code)` in the browser accepts only `1000` or `3000`–`4999`; anything else throws. Put your application's reasons in the 4000 range.

Heartbeats

Concern	Technique
Dead peer detection	Server pings; terminate if no pong
Idle proxy timeout	Ping more often than the timeout
Dead server, seen by client	Client-side timeout on silence

TCP does not tell you when a phone loses signal; the socket looks open until a write fails. Ping on a timer and drop clients that miss a pong. Browsers answer pings automatically but expose no ping API, so the client side detects a dead server by its own timeout.

```
import { WebSocketServer } from "ws";
const port = 8081;
const wss = new WebSocketServer({ port });
const alive = new WeakSet<object>();
wss.on("connection", (ws) => {
  alive.add(ws);
  ws.on("pong", () => alive.add(ws));
  ws.on("message", (m) => ws.send(String(m)));
});
setInterval(() => {
  for (const ws of wss.clients) {
    if (!alive.has(ws)) ws.terminate();
    else { alive.delete(ws); ws.ping(); }
```

```
  }
}, 30_000);
```

Reconnection and backpressure

Concern	Technique
Reconnect storm	Exponential backoff, full jitter, cap
Missed messages	Resume from a sequence number
Browser `WebSocket`	No backpressure; watch `bufferedAmount`
`ws` on Node	`bufferedAmount`, `send` callback
WebTransport	Backpressure via streams

Unlike `EventSource`, a `WebSocket` never reconnects by itself. The browser `WebSocket` also queues every `send` in memory and delivers every incoming message however slowly you process them (`WebSocketStream` fixes this but is non-standard). Check `bufferedAmount` before sending, and drop or coalesce updates for slow clients.

```
let ws: WebSocket;
function connect(u: string, n = 0) {
  ws = new WebSocket(u);
  ws.onopen = () => { n = 0; };
  ws.onclose = (e) => {
    if (e.code === 1000) return; // normal
    const cap = Math.min(30_000, 500 * 2 ** n);
    setTimeout(() => connect(u, n + 1),
```

```
      Math.random() * cap); // full jitter
  };
}
connect("wss://example.com/ws");
function send(msg: string) {
  if (ws.bufferedAmount < 1e6) ws.send(msg);
}
```

> **Warning:** Reconnecting every client after a fixed delay brings all of them back in the same second after a deploy. Randomise the delay (full jitter) and cap it.

Auth on the upgrade

Option	Trade-off
Cookie session	Works; check `Origin` against CSWSH
One-time ticket in query	Short-lived, single use; URLs get logged
Token in first message	Socket open before auth; enforce a timeout
`Authorization` header	Non-browser clients only

Browsers cannot set custom headers on `WebSocket` or `EventSource`, so an `Authorization: Bearer` header is impossible from the page. Authenticate in the HTTP upgrade, before the socket exists: issue a short-lived ticket over your normal authenticated API, then redeem it once.

```
import { createServer } from "node:http";
import { WebSocketServer as WSS } from "ws";
const ORIGIN = "https://app.example.com";
const tickets = new Set(["t-1"]); // one-use
const wss = new WSS({ noServer: true });
const srv = createServer().listen(8082);
srv.on("upgrade", (req, sock, head) => {
  const t = new URL(req.url ?? "/", ORIGIN)
    .searchParams.get("t") ?? "";
  const ok = req.headers.origin === ORIGIN
    && tickets.delete(t);
  if (!ok) return sock.destroy();
  wss.handleUpgrade(req, sock, head,
    (ws) => wss.emit("connection", ws, req));
});
```

> **Gotcha:** CORS does not apply to WebSockets. Any site can open a socket to your server with the user's cookies attached (cross-site WebSocket hijacking). Check `Origin` on every upgrade.

Scaling and serverless

Platform	Constraint
Many Node servers	Pub/sub (e.g. Redis) to fan out
API Gateway WebSocket	2 h max, 10 min idle, 128 KB message

Platform	Constraint
Durable Objects	Hibernation API; no billed duration idle
SignalR on a farm	Sticky sessions, or Azure SignalR Service

A client is connected to exactly one process, so a message published on server A must reach sockets held by server B: publish to a shared bus and let every server deliver to its own clients. Serverless platforms hold the socket for you and invoke your code per message.

```
import { DurableObject } from "cloudflare:workers";

export class Room extends DurableObject {
  async fetch() {
    const [client, server] =
      Object.values(new WebSocketPair());
    this.ctx.acceptWebSocket(server);
    return new Response(null,
      { status: 101, webSocket: client });
  }
  async webSocketMessage(ws, msg) {
    for (const p of this.ctx.getWebSockets())
      p.send(msg);
  }
}
```

> **Gotcha:** API Gateway closes every WebSocket after 2 hours and after 10 idle minutes, and neither quota is adjustable. Clients must reconnect by design, not as an error path.

SignalR on ASP.NET Core

API	Does
`AddSignalR()`	Register hub services
`MapHub<T>("/chat")`	Expose a hub endpoint
`Clients.All` / `Caller` / `Group(g)`	Choose recipients
`AddStackExchangeRedis(conn)`	Redis backplane
`KeepAliveInterval`	Server ping, default 15 s

SignalR negotiates the best transport — WebSockets, then server-sent events, then long polling — and adds hubs, groups and reconnection on top. In the browser it sends the access token as a query-string parameter, because those APIs cannot set headers.

```
using Microsoft.AspNetCore.SignalR;


var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSignalR();
var app = builder.Build();
app.MapHub<ChatHub>("/chat");
app.Run();
```

```
public class ChatHub : Hub
{
  public Task Send(string user, string text) =>
    Clients.All.SendAsync("message", user, text);
}
```

> **Gotcha:** The Redis backplane still needs sticky sessions. Only a single server, Azure SignalR Service, or WebSockets-only clients with `SkipNegotiation` avoid them.

Further reading

- RFC 6455 — The WebSocket Protocol
- WHATWG HTML — Server-sent events
- MDN — WebSockets API
- MDN — Using server-sent events
- ws on GitHub
- ASP.NET Core SignalR scaling