Engineer

# Server-sent events

"A" a one-way stream of events from a server over HTTP

## What it is

Server-sent events are a way for a web server to push messages to a page over one ordinary HTTP response that it keeps open and writes into whenever there is news. They are defined in the HTML Standard, maintained by the WHATWG, which describes both the text format of the stream and `EventSource`, the browser interface that reads it. The name is often shortened to SSE, and the format is often called after its media type, `text/event-stream`.

A page opens a stream with one line, `new EventSource(url)`. The browser sends a `GET` request, which may carry `Accept: text/event-stream`, and expects a response with status 200 and that content type. The server sends the headers and then does not finish the body. Each time something happens, it writes a few lines of text, and the browser turns each block of lines into an event the page can listen for.

The format is deliberately plain. A stream is always UTF-8, and every line is one field. A `data` line carries the payload, and several `data` lines in a row are joined with line breaks. An `event` line gives the event a type, and without one the type is `message`. An `id` line sets the ID of the last event, and a `retry` line sets how many milliseconds the browser waits before it reconnects. A line that starts with a colon is a comment, which the browser ignores. A blank line ends the block and dispatches the event, so a block cut off before its blank line is thrown away.

Reconnection is part of the standard. When the connection drops, the browser waits for the reconnection time, which starts at a value of its own choosing, according to the standard probably a few seconds, and then sends the request again. If the server has given IDs to its events, the new request carries the last one in a `Last-Event-ID` header, and the server can carry on from there. Some

answers end the stream for good instead: a status other than 200 or a different content type makes the browser give up and stop trying, and the standard names 204 No Content as the way for a server to say that the stream is over.

The direction is one way only. After the opening request, nothing travels from the page to the server on that connection; when the page has something to say, it sends a separate HTTP request.

Use the concept in a decision

Put “What are server-sent events?” 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

Plenty of features that look live only need one direction: the progress of a long job, a notification, a figure on a dashboard, an answer that arrives in parts. Without a stream, the page has to poll. It asks every few seconds, most answers say nothing has changed, and news waits up to one interval before anyone sees it. With a stream, the server writes the moment something happens.

The usual alternative for pushing data is WebSocket, which switches the connection away from HTTP to a protocol of its own. Server-sent events stay inside HTTP from start to finish, so they pass through the same proxies and load balancers as any other request, they use the same cookies and the same rules for cross-origin requests, and they can share one HTTP/2 connection with the rest of the page. Reconnection and resuming from the last event come with the browser instead of being code that every team writes for itself. WebSocket is still the right choice when the client has as much to say as the server, as in chat or a shared editor.

The format is not tied to the browser either. Any HTTP client can read it, and specifications outside the web use it. In the Model Context Protocol, through which AI agents call tools, the Streamable HTTP transport lets a server answer a request either with one JSON object or with a `text/event-stream` response. On that stream it may send notifications and requests of its own before the final

response, and it can attach event IDs so that a client which loses the connection resumes with `Last-Event-ID`. Streamable HTTP replaced HTTP+SSE, the transport of the earlier protocol version 2024-11-05.

The limits are worth knowing before choosing it. The stream carries text only, so binary data has to be encoded as text first. The browser's `EventSource` constructor takes a URL and one option, `withCredentials`, and nothing else, so a page cannot add an `Authorization` header to the request.

## What it looks like in practice

The exchange below follows a long export job. The browser has lost its connection after event 41 and is reconnecting, so its request carries that ID, and the server carries on from event 42:

```
GET /jobs/4471/events HTTP/1.1
Host: example.com
Accept: text/event-stream
Last-Event-ID: 41

HTTP/1.1 200 OK
Content-Type: text/event-stream
Cache-Control: no-store

retry: 5000

: still working

id: 42
event: progress
data: {"done": 120, "total": 400}

id: 43
event: progress
data: {"done": 400, "total": 400}

id: 44
event: finished
data: {"report": "/jobs/4471/report.pdf"}
```

The `retry` line asks the browser to wait five seconds before any later reconnection, and the comment line carries no event at all; it is there to keep the connection busy while the job is quiet. The page listens for the two event types by name:

```
const source = new EventSource("/jobs/4471/events");
source.addEventListener("progress", (event) => {
  const { done, total } = JSON.parse(event.data);
  showProgress(done / total);
});
source.addEventListener("finished", (event) => {
  showReport(JSON.parse(event.data).report);
  source.close();
});
source.addEventListener("error", () => {
  if (source.readyState === EventSource.CLOSED) showRetryButton();
});
```

The call to `close()` matters. When the server ends a response normally, the browser treats it as a dropped connection and reconnects, so a stream for a finished job would otherwise open again after five seconds. The `error` handler tells the two kinds of failure apart: while `readyState` is `CONNECTING`, the browser is still trying, and once it is `CLOSED`, it has given up.

Work that goes on after the agent has its answer comes up on this site's page on turning existing software into tools for AI agents. It describes offering a company's operations as tools, with an MCP server offering the same operations to the agents that speak it, and slow work moved onto a queue so that the agent gets an answer while the work goes on. In the MCP specification, a server can use an event stream like the one above to send its own requests or notifications before the final response to the client's request.

## Where it goes wrong

**Events that arrive in bursts, or all at the end.** Something between the server and the browser holds the response back. nginx, for one, buffers responses from the server behind it by default, and passes a response on as it arrives only when buffering is off. The HTML Standard also warns that HTTP chunking done by a layer that knows nothing about the stream's timing can make it unreliable. Turn buffering off for the stream's location, in nginx with `proxy_buffering off` or a response header of `X-Accel-Buffering: no`, flush the output after every event in the server code, and test through the whole path, proxy included, rather than against the server alone.

**Quiet streams cut off by a timeout.** A proxy that sees no bytes for a while assumes the connection is dead. In nginx, `proxy_read_timeout` defaults to 60 seconds between two reads, after which the connection is closed, and the HTML

Standard warns that some older proxies drop connections after a short timeout. It suggests a comment line every 15 seconds or so, which costs a few bytes and which the browser ignores.

**Events lost in a reconnection.** The browser sends `Last-Event-ID` only if the server gave its events IDs, and the header helps only if the server can still find what came after that ID. Give every event an ID that works as a cursor in the stream, keep a window of recent events to replay, and send the full current state when an ID is too old to resume from. The MCP specification adds a rule worth copying: IDs belong to one stream, and a server must not replay on one stream messages meant for another.

**Reconnection that never stops, or stops for good.** A stream that ends normally comes back after the reconnection time, again and again, so a finished job keeps reopening its stream unless the page calls `close()` or the server answers the reconnection with 204. The opposite happens during a deploy. A proxy that answers 502 or 503 while the servers restart gives every open `EventSource` a status other than 200, and every one of them stops for good. Listen for `error`, and when `readyState` is `CLOSED`, create a new `EventSource` after a random delay that grows with each failure.

**Too many streams for one browser.** Browsers limit how many connections they keep open to one server, and a stream served over HTTP/1.1 holds one of them for as long as it lasts. The HTML Standard warns that a site where every open page has its own `EventSource` can run into that limit. Serve the stream over HTTP/2, where streams share one connection and RFC 9113 recommends allowing at least 100 at once, or share one `EventSource` between the tabs through a shared worker.

**The token in the address.** Because `EventSource` cannot send an `Authorization` header, the quick fix is a token in the URL, where reverse proxies and access logs write it down. Authenticate the stream with the session cookie the page already has, adding `withCredentials` when the stream comes from another origin, or with a short-lived, single-use ticket that the page fetches over an authenticated request just before it opens the stream.

# Further reading

- HTML Standard: Server-sent events — the stream format, the `EventSource` interface, reconnection and the authoring notes on proxies and connection limits.
- Transports - Model Context Protocol — Streamable HTTP, where a server may answer with an event stream, and how a broken stream resumes.
- RFC 9113: HTTP/2 — streams that share one connection, and the setting that limits how many are open at once.
- nginx: Module ngx_http_proxy_module — `proxy_buffering`, the `X-Accel-Buffering` header and `proxy_read_timeout`.

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 are server-sent events?” 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/server-sent-events/