Engineer

Server-sent events

"A" en envägsström av händelser från en server över HTTP

Vad det är

Server-sent events är ett sätt för en webbserver att skicka meddelanden till en sida genom ett vanligt HTTP-svar, som servern håller öppet och skriver i varje gång det finns något nytt. De definieras i HTML-standarden, som WHATWG underhåller, och den beskriver både textformatet för strömmen och `EventSource`, gränssnittet i webbläsaren som läser strömmen. Namnet förkortas ofta SSE, och formatet kallas ofta efter sin medietyp, `text/event-stream`.

En sida öppnar en ström med en enda rad, `new EventSource(url)`. Webbläsaren skickar en `GET`-begäran, som kan ha med sig `Accept: text/event-stream`, och väntar sig ett svar med status 200 och den innehållstypen. Sedan skickar servern sina headrar men gör aldrig klart svaret. Varje gång något händer skriver den några rader text, och webbläsaren gör om varje block av rader till en händelse som sidan kan lyssna efter.

Formatet är medvetet enkelt. En ström är alltid UTF-8, och varje rad är ett fält. En `data`-rad bär innehållet, och flera `data`-rader i följd sätts ihop med radbrytningar. En `event`-rad ger händelsen en typ, och utan den är typen `message`. En `id`-rad sätter id:t för den senaste händelsen, och en `retry`-rad anger hur många millisekunder webbläsaren väntar innan den ansluter igen. En rad som börjar med kolon är en kommentar, som webbläsaren ignorerar. En tom rad avslutar blocket och skickar i väg händelsen, så ett block som avbryts före sin tomma rad kastas.

Återanslutning är en del av standarden. När anslutningen bryts väntar webbläsaren under återanslutningstiden, som börjar på ett värde som webbläsaren själv väljer, enligt standarden troligen några sekunder, och skickar sedan begäran på nytt. Har servern gett sina händelser id:n har den nya begäran med sig det senaste av dem i headern `Last-Event-ID`, och servern kan fortsätta

därifrån. Vissa svar avslutar i stället strömmen för gott: en annan status än 200 eller en annan innehållstyp får webbläsaren att ge upp och sluta försöka, och standarden pekar ut 204 No Content som sättet för en server att säga att strömmen är slut.

Riktningen går bara åt ett håll. Efter den första begäran går ingenting från sidan till servern över den anslutningen; när sidan har något att säga skickar den en egen HTTP-begäran.

Använd begreppet i ett beslut

Använd ”Vad är server-sent events?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Många funktioner som ser levande ut behöver bara en riktning: förloppet i ett långt jobb, ett nytt meddelande, en siffra på en dashboard, ett svar som kommer i delar. Utan en ström måste sidan fråga om och om igen, så kallad polling. Den frågar med några sekunders mellanrum, de flesta svaren säger att ingenting har ändrats, och nyheter väntar upp till ett helt intervall innan någon ser dem. Med en ström skriver servern i samma ögonblick som något händer.

Det vanliga alternativet för att skicka data till en sida är WebSocket, som byter över anslutningen från HTTP till ett eget protokoll. Server-sent events håller sig till HTTP från början till slut, så de passerar samma proxyer och lastbalanserare som vilken annan begäran som helst, de använder samma cookies och samma regler för begäranden mellan olika ursprung, och de kan dela en HTTP/2-anslutning med resten av sidan. Återanslutning och återupptagning från den senaste händelsen följer med webbläsaren i stället för att vara kod som varje team skriver själv. WebSocket är fortfarande rätt val när klienten har lika mycket att säga som servern, som i en chatt eller en delad editor.

Formatet är inte heller bundet till webbläsaren. Vilken HTTP-klient som helst kan läsa det, och specifikationer utanför webben använder det. I Model Context Protocol, som AI-agenter anropar verktyg genom, låter transporten Streamable

HTTP en server svara på en begäran antingen med ett enda JSON-objekt eller med ett `text/event-stream`-svar. På den strömmen kan servern skicka notifieringar och egna begäranden före det slutliga svaret, och den kan sätta id:n på händelserna, så att en klient som tappar anslutningen kan fortsätta med `Last-Event-ID`. Streamable HTTP ersatte HTTP+SSE, transporten i den tidigare protokollversionen 2024-11-05.

Begränsningarna är värda att känna till innan man väljer det. Strömmen bär bara text, så binära data måste först kodas som text. Webbläsarens `EventSource`-konstruktor tar en URL och en enda inställning, `withCredentials`, och ingenting annat, så en sida kan inte lägga till en `Authorization`-header i begäran.

Så ser det ut i praktiken

Utbytet nedan följer ett långt exportjobb. Webbläsaren har tappat anslutningen efter händelse 41 och ansluter igen, så begäran har med sig det id:t, och servern fortsätter från händelse 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"}
```

`retry` -raden ber webbläsaren vänta fem sekunder före varje senare återanslutning, och kommentarraden bär ingen händelse alls; den finns där för att hålla anslutningen aktiv medan jobbet är tyst. Sidan lyssnar efter de två händelsetyperna vid namn:

```
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();
});
```

Anropet till `close()` spelar roll. När servern avslutar ett svar på vanligt sätt behandlar webbläsaren det som en bruten anslutning och ansluter igen, så utan anropet skulle en ström för ett färdigt jobb öppnas igen efter fem sekunder. `error` -hanteraren skiljer mellan de två sorternas fel: så länge `readyState` är `CONNECTING` försöker webbläsaren fortfarande, och när den är `CLOSED` har den gett upp.

Arbete som fortsätter efter att agenten har fått sitt svar är ett av ämnena på sidan om din programvara som verktyg för AI-agenter här på webbplatsen. Den beskriver hur varje operation som en agent ska ha tillgång till erbjuds som ett verktyg, med en MCP-server som erbjuder samma operationer till de agenter som talar det protokollet, och hur tidskrävande arbete flyttas till en kö så att agenten får ett svar medan arbetet fortsätter. I MCP-specifikationen kan en server använda en händelseström som den ovan för att skicka egna förfrågningar eller notifieringar före det slutliga svaret på klientens förfrågan.

Var det går fel

Händelser som kommer i klumpar, eller alla på en gång i slutet. Något mellan servern och webbläsaren håller tillbaka svaret. nginx, till exempel, buffrar som standard svaren från servern bakom sig och skickar bara vidare ett svar allteftersom det kommer när buffring är avstängd. HTML-standarden varnar också för att HTTP-chunking som utförs av ett lager som inte känner till tidpunkterna i strömmen kan göra den opålitlig. Stäng av buffring för strömmens

location, i nginx med `proxy_buffering off` eller med headern `X-Accel-Buffering: no` i svaret, töm skrivbufferten efter varje händelse i serverkoden, och testa genom hela kedjan, proxyn inräknad, i stället för mot servern ensam.

Tysta strömmar som kapas av en timeout. En proxy som inte ser en enda byte på ett tag utgår från att anslutningen är död. I nginx är standardvärdet för `proxy_read_timeout` 60 sekunder mellan två läsningar, och därefter stängs anslutningen, och HTML-standarden varnar för att vissa äldre proxyer bryter anslutningar efter en kort timeout. Den föreslår en kommentarrad ungefär var 15:e sekund, som kostar några få byte och som webbläsaren ignorerar.

Händelser som försvinner vid en återanslutning. Webbläsaren skickar bara `Last-Event-ID` om servern har gett sina händelser id:n, och headern hjälper bara om servern fortfarande kan hitta det som kom efter det id:t. Ge varje händelse ett id som fungerar som en markör i strömmen, behåll ett fönster med de senaste händelserna som kan spelas upp igen, och skicka hela det aktuella tillståndet när ett id är för gammalt att fortsätta från. MCP-specifikationen lägger till en regel som är värd att kopiera: id:n hör till en enda ström, och en server får inte spela upp meddelanden på en ström när de var avsedda för en annan.

Återanslutning som aldrig slutar, eller slutar för gott. En ström som avslutas normalt kommer tillbaka efter återanslutningstiden, om och om igen, så ett färdigt jobb fortsätter att öppna sin ström på nytt om inte sidan anropar `close()` eller servern svarar på återanslutningen med 204. Det motsatta händer under en driftsättning. En proxy som svarar 502 eller 503 medan servrarna startar om ger varje öppen `EventSource` en annan status än 200, och varenda en av dem slutar för gott. Lyssna efter `error`, och när `readyState` är `CLOSED`, skapa en ny `EventSource` efter en slumpmässig fördröjning som växer för varje fel.

För många strömmar till en webbläsare. Webbläsare begränsar hur många anslutningar de håller öppna till en server, och en ström över HTTP/1.1 upptar en av dem så länge den pågår. HTML-standarden varnar för att en webbplats där varje öppen sida har sin egen `EventSource` kan slå i den gränsen. Leverera strömmen över HTTP/2, där strömmar delar en anslutning och RFC 9113 rekommenderar att minst 100 tillåts samtidigt, eller dela en enda `EventSource` mellan flikarna genom en shared worker.

En token i adressen. Eftersom `EventSource` inte kan skicka en `Authorization`-header blir den snabba lösningen en token i URL:en, där omvända proxyer och åtkomstloggar skriver ner den. Autentisera strömmen med sessionens cookie, som sidan redan har, och lägg till `withCredentials` när strömmen kommer från ett annat ursprung, eller med en kortlivad engångsbiljett som sidan hämtar med en autentiserad begäran precis innan den öppnar strömmen.

Läs mer

- HTML Standard: Server-sent events — formatet för strömmen, gränssnittet `EventSource`, återanslutning och standardens anmärkningar till författare om proxyer och gränser för antalet anslutningar.
- Transports - Model Context Protocol — Streamable HTTP, där en server kan svara med en händelseström, och hur en avbruten ström återupptas.
- RFC 9113: HTTP/2 — strömmar som delar en anslutning, och inställningen som begränsar hur många som kan vara öppna samtidigt.
- nginx: Module ngx_http_proxy_module — `proxy_buffering`, headern `X-Accel-Buffering` och `proxy_read_timeout`.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är server-sent events?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/server-sent-events/