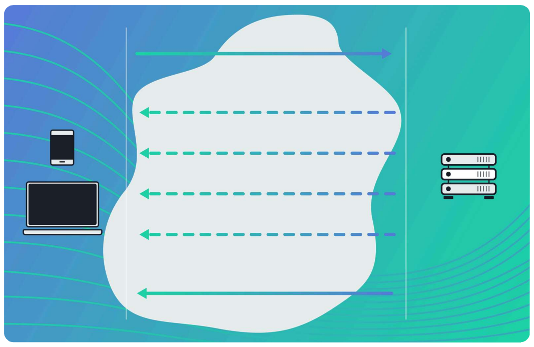
ECOSYSTEM

November 23, 2022



A Primer on Server-Sent Events (SSE)

The “what”, “how” and “why” of one of the best ways to push data across the web.

Leonid Lukyanov
Team Aklivity

Introduction

In our daily lives we’re not accustomed to measuring interactions in milliseconds, but when it comes to anything digital, an extra 100-200 milliseconds is noticeable “lag”. If the delay is bumped up to 300 milliseconds, an app will often be reported as “sluggish”, and if the hold-up is over 1000 milliseconds (*just 1 second!*), many users will lose focus, or even worse, close the app altogether. Considering that an average blink takes 400 milliseconds, user engagement

Table of Contents

(and subsequently business) can literally be lost in the blink of an eye.

To deliver modern, responsive experiences, data needs to be proactively pushed/streamed to end users. The internet, which was initially built around the HTTP request-response model, wasn't designed for this, and so over the years developers have had to come up with new technologies to unlock "realtime" interaction patterns over the web. One of these, is Server-Sent Events (SSE) — a lightweight, open protocol that allows servers to proactively push data to clients over a standard HTTP connection.

This article covers the basics of Server-Sent Events — what are they, how they work, and why they're needed. It also touches upon some of the security challenges that SSE may introduce and how to potentially overcome them.

What is SSE?

Server-Sent Events is a standardized push technology that enables sending notifications, messages and events from a server to a client via an HTTP request. First conceptualized in 2004, it was included as part of the HTML5 specification, and today it's supported by all browsers and web/mobile application environments.

With SSE, servers expose a URL through which clients subscribe to automatic message updates or continuous streams of data. It removes the need for (and complexity of) polling a server for the latest data and supports delivering "realtime" digital experiences.

How does SSE Work?

SSE provides a connection management layer and parsing logic that enable an HTTP response to be kept open while a server pushes new events to a client as they become available. The three main components of SSE are:

1. The EventSource API, which is used by clients to receive and process events.
2. An HTTP URL that a server exposes for clients to make an initial GET request to (via the EventSource API).
3. The Event Stream format (MIME type **text/event-stream**), which describes the structure of the messages sent by the server to the client. These messages are text-based, UTF-8 encoded and have the following optional fields: **event:**, **data:**, **id:** and **retry:**.

The EventSourceAPI

The EventSource API is a standard interface that defines how browsers and servers can communicate with each other using SSE.

When a web client wants to receive updates from a server via SSE, it first creates an EventSource object. This object represents the event stream between the client and server and carries different methods for processing events.

```
        //Instantiate the EventSource object 'events'
        var events = new EventSource("//api.example.com/events");

        //EventSource methods
        events.onopen(e => console.log("opened"));
        events.onmessage(e => console.log(`message: ${e.data}`));
        events.onerror(e => console.log("error: reconnecting"));
        events.addEventListener("bingo", e =>
  console.log(`bingo: ${e.data}`);
```

Next, the client sends an HTTP GET request to the server with the necessary headers to indicate that it wants to receive updates via SSE. If the server supports SSE and the request headers are valid, then the server will return an HTTP 200 OK response along with the results of the GET request in the body of the response. These results will be formatted as a **text/event-stream** consisting of an event name, some data, and a newline character (\n).

So, first the SSE client sends a request…

```
GET /events
accept: text/event-stream
cache-control: no-cache
```

…then the server sends response headers, triggering**.onopen()**.

```
200 OK
content-type: text/event-stream
```

```
> opened
```

Once the server opens an SSE stream, it sends a response body fragment, triggering **.onmessage()**.

```
data: Hello, world\n
\n
```

```
> message: Hello, world
```

By default, Server-Sent Events have an empty **event:** field and are of type "message", and a client can subscribe to all such "messages" with **.onmessage().** However, by modifying the **event:** field, a server can assign a custom type to an SSE stream, which the client can then handle differently with **.addEventListener()**.

Here the SSE server sends another response body fragment, this time triggering the **bingo** listener:

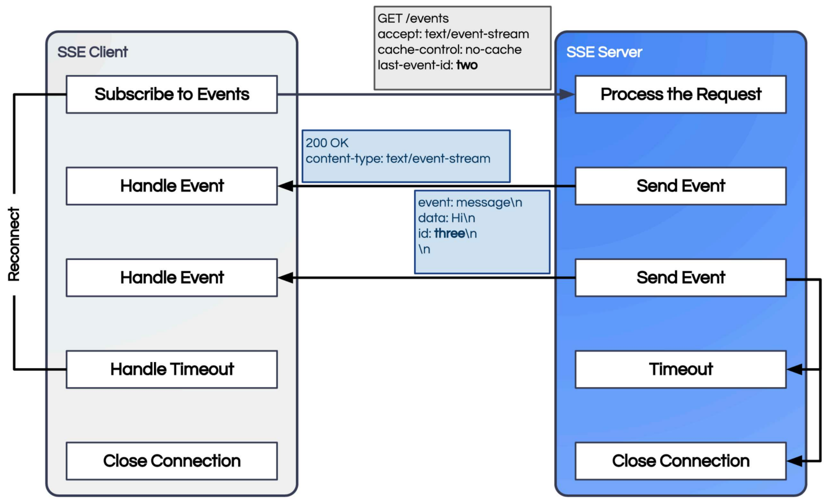
```
event: bingo\n
data: winner\n
\n
```

```
> bingo: winner
```

Auto-reconnect

SSE comes with built-in error handling. When the response between an SSE client and a server is interrupted, the event stream is automatically reestablished. Moreover, if the server sets the **id:** field in any message already received by the client, then upon reconnecting, the client will send back the last message **id:** it received inside a **Last-Event-ID** header.

An SSE server also has the opportunity to inform the client how long it should wait when the response is interrupted before initiating the reconnect request by setting the **retry:** field. If an SSE stream does need to be closed, then the client has to explicitly call **.close()**



The SSE workflow.

SSE's auto-reconnect capabilities are particularly useful when there's a need to reliably deliver data to clients that may have unstable connectivity, such as mobile phones in "spotty" reception areas.

Why SSE?

SSE is an elegant solution for pushing/streaming data between a server and a client. Compared to other streaming protocols and technologies, such as WebSocket, MQTT or proprietary pub/sub services, SSE is easier to get started with and manage. However, there are certain limitations to consider.

SSE Advantages

- It's a W3C standard supported by all major browsers and app frameworks (no proprietary SDKs or client libraries needed)
- Standard browser-based developer tools can be used for debugging clients
- Plays nicely with enterprise firewalls because it's HTTP-based (packet inspection is not an issue)
- Error handling via auto-reconnect comes built-in
- Can be "backported" to earlier browsers with JavaScript polyfilling

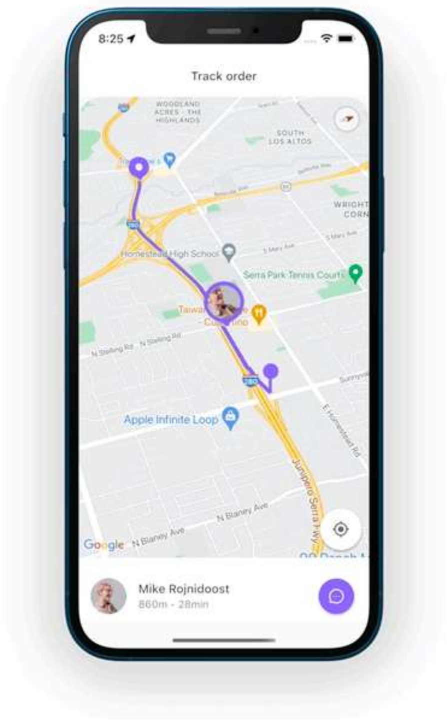
SSE Limitations

- SSE is unidirectional so it cannot be used to pass data from the client back to the server
- When used over HTTP 1.1 there's a limit of 6 open SSE connections per browser, but when SSE is used over HTTP/2 the maximum number of simultaneous HTTP streams is negotiated between the server and the client (defaults to 100)
- SSE is not supported by Microsoft Edge/IE v78 and earlier (but can be pollyfield in such cases)
- No support for binary data (only UTF-8 encoded text)

Use Cases

For web applications and websites that need to operate asynchronously or in "realtime", and there is no requirement for bi-directional data streaming between the client and server, SSE is a great fit. Some exciting use cases for SSE include:

- Receiving live sport scores
- Subscribing to a stream of stock/financial data
- Delivering in-app push notifications
- Displaying live geo tracking data on a map
- Social media news feeds
- Live analytics charts and graphs



SSE Security Challenges

Not only do different clients often have different data access privileges, but these privileges may change at any time. Ensuring that a client is consistently authorized to access a server is a critical security requirement, especially in an enterprise setting.

With HTTP request-response, client to server communication is batched, and as a result each request for data can be authenticated without any impact on user experience.

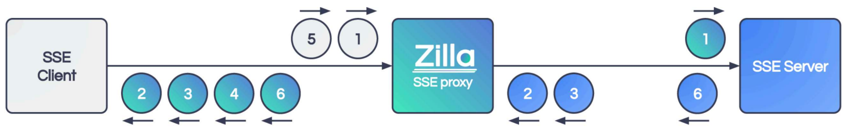
With SSE, a client only makes a single request, after which an open connection is established. While the initial request can be authenticated via the same techniques as in the request-response case, how should re-authentication be handled for a client that's in the midst of consuming a data stream?

The naive approach would be to simply terminate the SSE stream after a set period of time and have the client re-authenticate when it goes to establish a new request, but this disrupts the continuous nature of streaming, and leads to a frustrating user experience.

Securing SSE with aklivity Zilla

One way of handling the authentication challenges presented by SSE is with Zilla Continuous Stream Authorization.

Zilla is an open source event-driven API gateway that can act as an SSE reverse-proxy. When deployed between an SSE client and server, it gracefully re-authorizes the client on the server's behalf, without abruptly terminating message streams.



The six steps of Zilla Continuous Stream Authorization.

Continuous Authorization with Zilla works as follows:

1. An HTTP(S) request for an SSE stream with a JWT access token (authorized by Zilla) is made.
2. HTTP(S) response headers for an SSE stream (token authorized for N seconds) are issued.
3. A message event is sent by the SSE server, through Zilla, to the SSE client.
4. A challenge event is sent by the Zilla to the SSE client (less than N secs later). If no challenge-response is sent by the SSE client before the token expires, Zilla closes the event stream.

5. A new HTTP(S) request for challenge-response with a refreshed access token (reauthorized by Zilla) is made.
6. Another message event is sent by the SSE server, through Zilla, to the SSE client (more than N seconds later).

In summary, Zilla helps deliver a secure and smooth user experience with SSE.

PS: SSE, Zilla and Apache Kafka

Server-Sent Events are a great way to extend the reach of Apache Kafka-based event-driven architectures to the edge. Besides a vanilla SSE Proxy, Zilla can also be configured as a Kafka-SSE Proxy. In such a configuration, Zilla maps Kafka event streams to SSE streams and enables web clients to consume data directly from a Kafka topic.

With Zilla, Kafka can be turned into a fully fledged, asynchronous backend for mobile and browser applications, eliminating the need for a dedicated SSE server.

To learn more about Zilla and try out examples and demos, please visit its GitHub repository.

Conclusion

The world happens in "realtime", so data should reach end users likewise. Server-Sent Events (SSE) is standardized technology that supports pushing updates and continuously streaming data to web-based clients. Unlike other approaches, SSE comes with powerful features such as auto-reconnect built-in, and is HTTP-based, so it's easy and familiar to get started with. While it does have some limitations, such as being unidirectional, SSE ultimately can help accelerate the delivery of modern digital experiences.