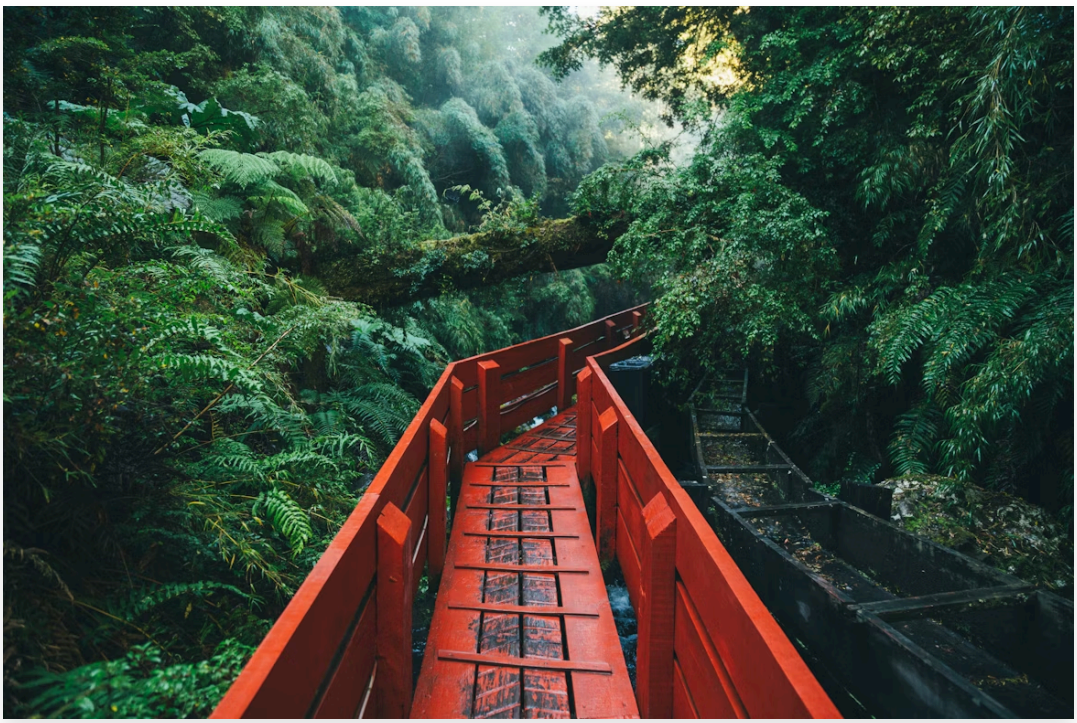
Fictionally Irrelevant.

Why you should use Server-Side Events over Web Sockets and Long-Polling.



Harshit Singhai

September 25, 2021

Server Sent Events are a standard allowing browser clients to receive a stream of updates from a server over a HTTP connection without resorting to polling. Unlike WebSockets, Server Sent Events are a one way communications channel - events flow from server to client only.

Benefits of Server-Sent Events (SSE)

There are many practical use cases for using SSE. Updating dynamic content, sending push notifications, and streaming data in Real-time are just a few of the applications that SSE can be utilized for. This post will explain SSE, when to use it, and implement a simple SSE application with FastAPI and React.

Server-Sent Events (SSE) is often overshadowed by its two big brothers — Web Sockets and Long-Polling. However, there are many practical use cases for using SSE. Updating dynamic content, sending push notifications, and streaming data in Real-time are just a few of the applications that SSE can be utilized for. We implement a simple SSE application with FastAPI and React.

> “ Traditionally, a web page has to send a request to the server to receive new data; that is, the page requests data from the server. With server-sent events, it’s possible for a server to send new data to a web page at any time, by pushing messages to the web page.

Source: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

What does this mean? Essentially, Server-Sent Events (SSE) enable users to subscribe to a stream of real-time data. Anytime this data stream updates, users can see new events in real-time. If you have worked with Long-Polling or Web Socket applications in the past, you may be wondering what's the big deal about SSE?

You can think of SSE as a unidirectional websocket. Only the server can send messages to subscribed clients. There are many web applications where web sockets maybe overkill. For example, updating the price of an item on a product page does not need bidirectional communication. The server simply needs one-way communication to update prices for all of its clients. This is a perfect use case for SSE.

In short, SSE is a great tool for streaming quick real-time data. They offer unidirectional communication from a server to its clients and are typically used for updating dynamic content on web pages.

Long Polling

Long Polling is a method of communication where the client periodically hits the server for new data. This form of communication is often used when the application being built involves human intervention or executing computationally expensive tasks.

How does it work?

When working with Server Sent Events, communications between client and server are initiated by the client (browser). The client creates a new JavaScript EventSource object, passing it the URL of an endpoint which is expected to return a stream of events over time.

The server receives a regular HTTP request from the client (these should pass through firewalls etc like any other HTTP request which can make this method work in situations where WebSockets may be blocked). The client expects a response with a series of event messages at arbitrary times. The server needs to leave the HTTP response open until it has no more events to send, decides that the connection has been open long enough and can be considered stale, or until the client explicitly closes the initial request.

Every time that the server writes an event to the HTTP response, the client will receive it and process it in a listener callback function.

Example code using FastAPI and React

You can find all the code here https://github.com/harshitsinghai77/server-sent-events-using-fastapi-and-reactjs

We will be using python and FastAPI. FastAPI is a great tool for SSE applications as it is really easy to use and is built upon starlette which has SSE capabilities built in.

Server side

We create a fastapi based backend endpoint. Here we have a `/stream` endpoint which sends server events. FastAPI is based on starlette. To help us in this flow, we will depend on a small library called `sse_starlette`

```
import logging

from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from sse_starlette.sse import EventSourceResponse
import asyncio
import uvicorn

logger = logging.getLogger()

MESSAGE_STREAM_DELAY = 1  # second
MESSAGE_STREAM_RETRY_TIMEOUT = 15000  # milisecond
app = FastAPI()

# add CORS so our web page can connect to our api
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

COUNTER = 0


def get_message():
    global COUNTER
    COUNTER += 1
    return COUNTER, COUNTER < 21


@app.get("/stream")
async def message_stream(request: Request):
    async def event_generator():
        while True:
            if await request.is_disconnected():
                logger.debug("Request disconnected")
                break

            # Checks for new messages and return them to client if any
            counter, exists = get_message()
            if exists:
                yield {
                    "event": "new_message",
                    "id": "message_id",
```

```
                    "retry": MESSAGE_STREAM_RETRY_TIMEOUT,
                    "data": f"Counter value {counter}",
                }
            else:
                yield {
                    "event": "end_event",
                    "id": "message_id",
                    "retry": MESSAGE_STREAM_RETRY_TIMEOUT,
                    "data": "End of the stream",
                }

            await asyncio.sleep(MESSAGE_STREAM_DELAY)

    return EventSourceResponse(event_generator())


if __name__ == "__main__":
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

Client

React client side logic which consume a server sent event endpoint and act based on events:

```
import { useEffect, useState } from "react";
import logo from "./logo.svg";
import "./App.css";

const evtSource = new EventSource("http://127.0.0.1:8000/stream");

function App() {
  const [messages, setMessage] = useState([]);
  useEffect(() => {
    evtSource.addEventListener("new_message", function (event) {
      // Logic to handle status updates
      setMessage((messages) => [...messages, event.data]);
    });

    evtSource.addEventListener("end_event", function (event) {
      setMessage((messages) => [...messages, event.data]);
      evtSource.close();
    });

    return () => {
      evtSource.close();
    };
  }, []);
  return (
```

```
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          {messages.map((el) => (
            <p>{el}</p>
          ))}
        </header>
      </div>
    );
  }

  export default App;
```

- Now we can handle how client should respond to different server events easily
- end condition is very important. Without this, client will reconnecting to server, even if server has successfully completed previous request.

HTTP Connection Behaviour

- Browser tries to keep this connection alive. Even if server side closes the connection, browser will try to reconnect, unless event source is closed.
- This is useful to keep the connection alive even in case of intermittent network issue or server went down etc.,
- If browser is closed, server can sense this and stop sending any more events. This optimizes server side resources

Conclusion

You can find all the code here https://github.com/harshitsinghai77/server-sent-events-using-fastapi-and-reactjs

That concludes our quick tour of Server Sent Events. Thanks for reading.

That's it for today, see you soon. :)

May the Source be with you 🚀

> You cannot overestimate the unimportance of practically everything.

View on GitHub