WHAT THE HECK ARE SERVER SENT EVENTS?

ABOUT ME

» Developer Experience Engineer at LaunchDarkly

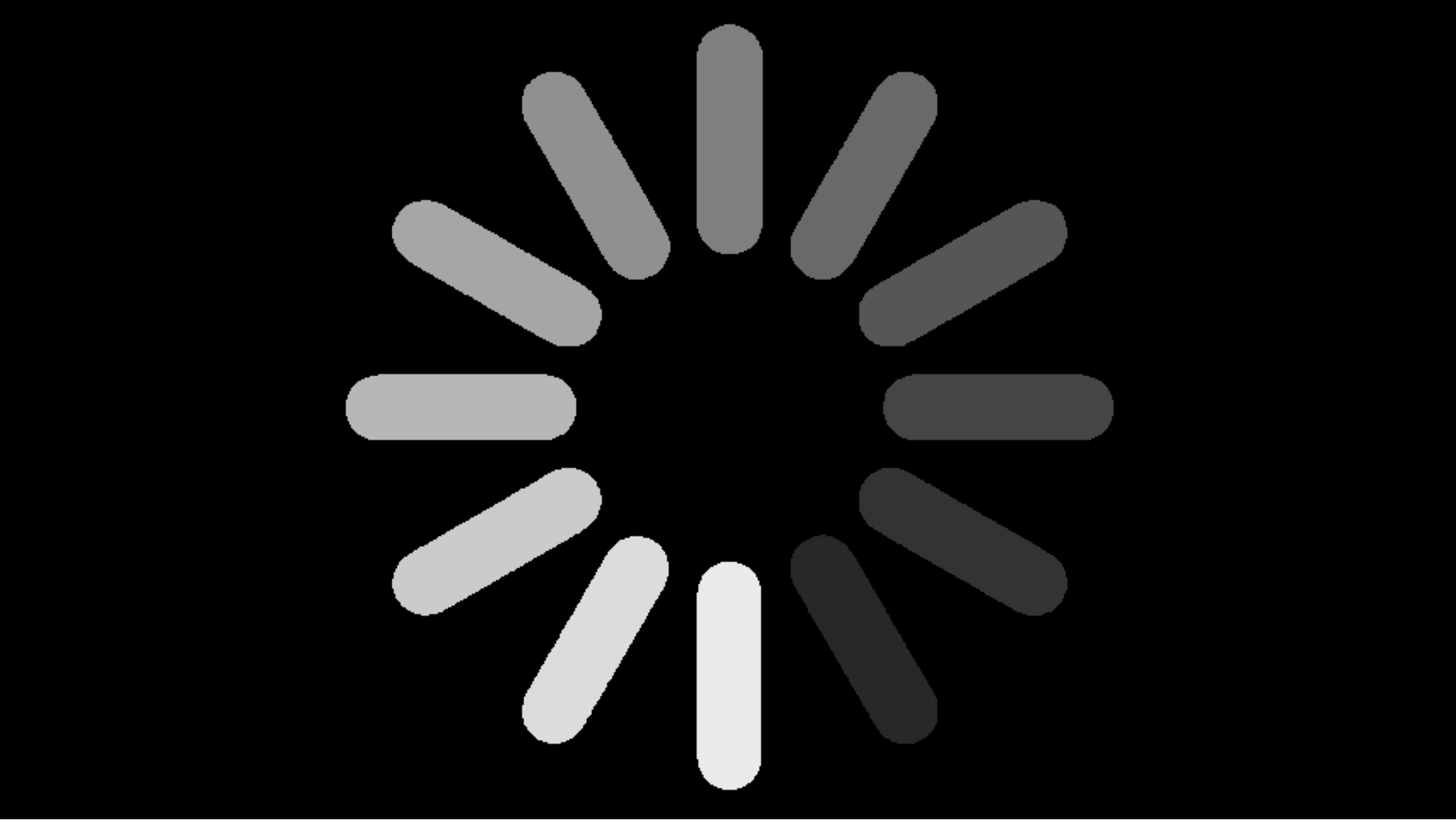
» Co-author of The Jamstack Book from Manning

» Editor of the Jamstacked newsletter

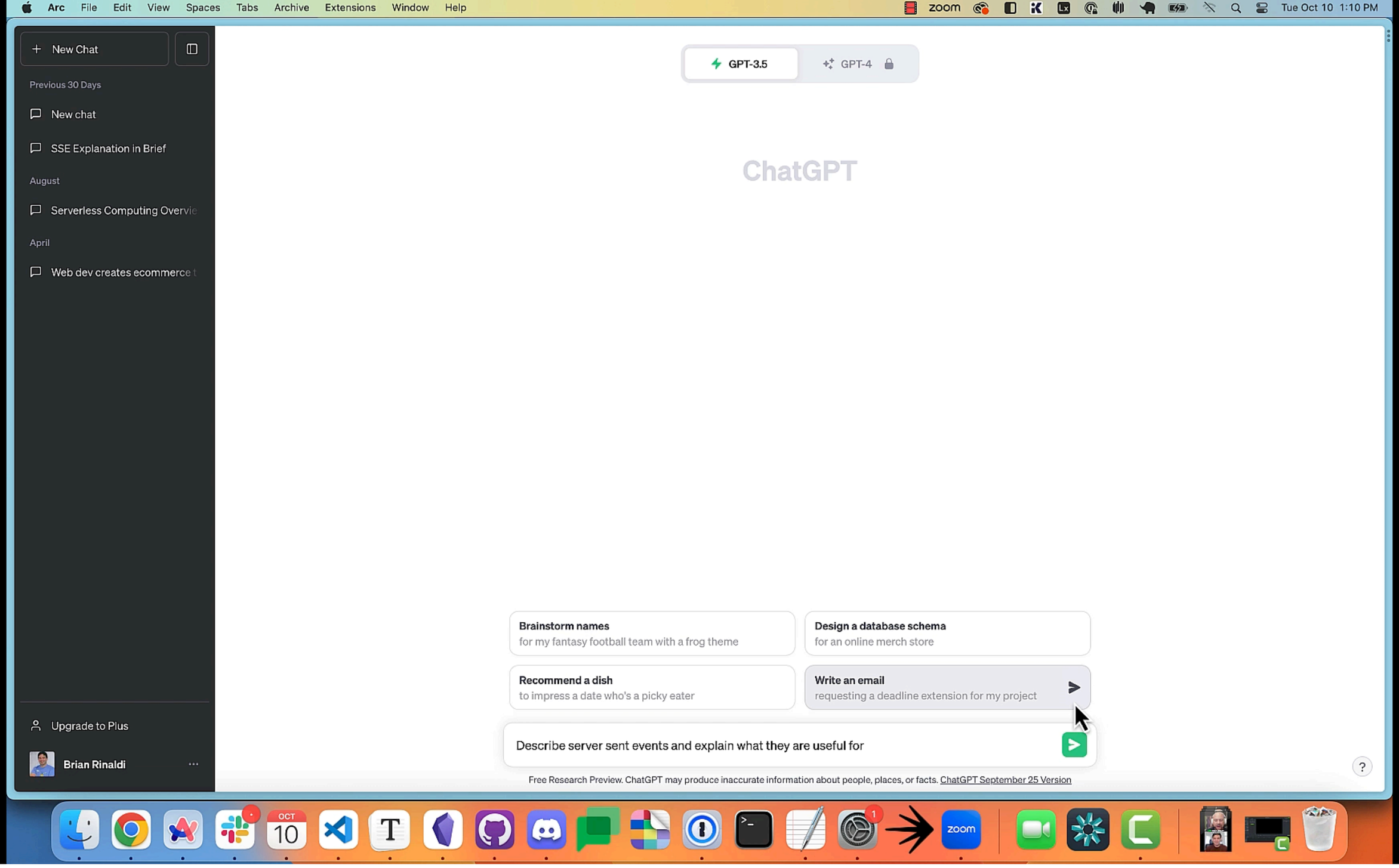
» Run the CFE.dev community and Orlando Devs


LaunchDarkly

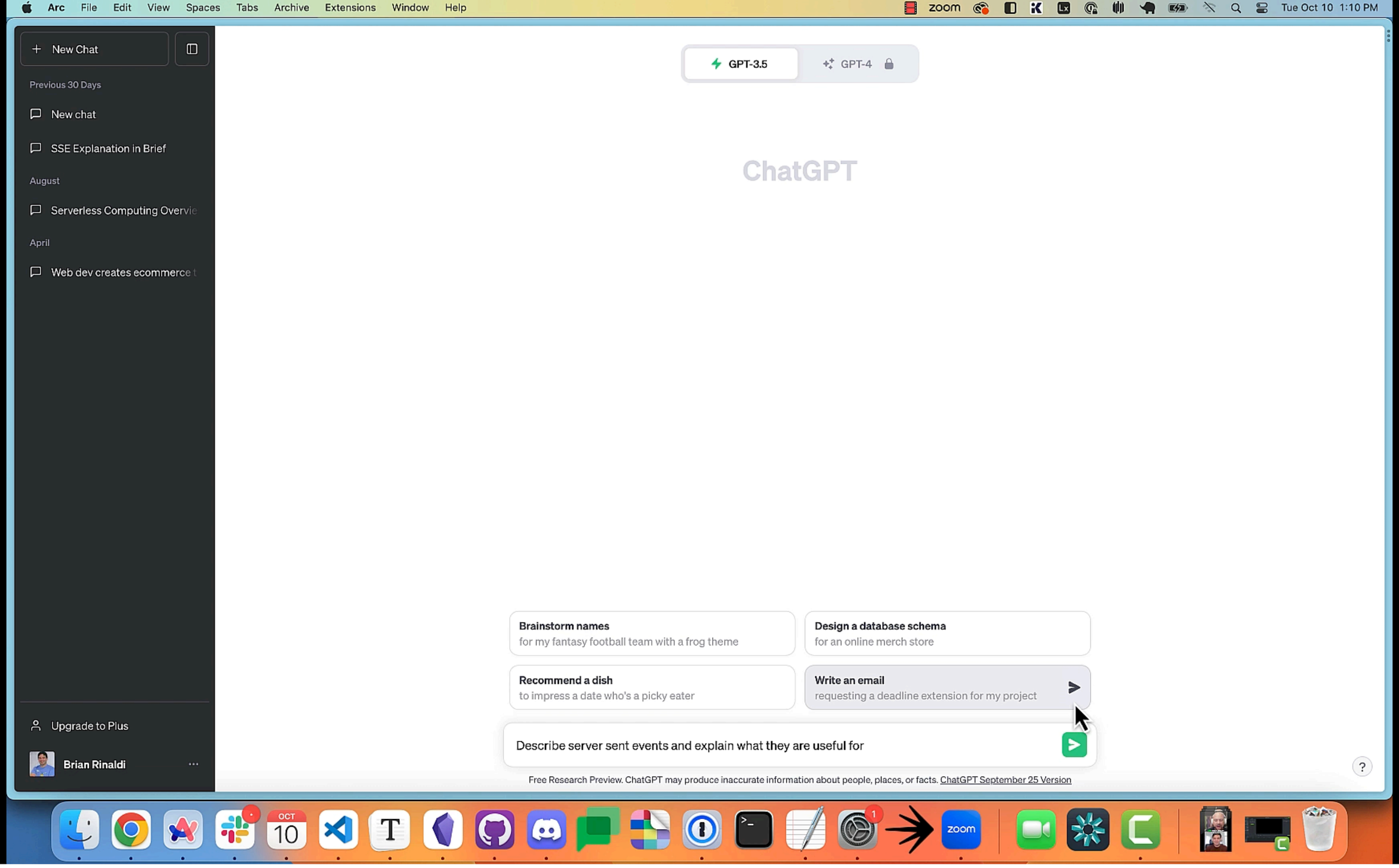
the traditional server side model is one where a request is made, the client waits and then eventually the server responds. For most stuff, this works just fine, but for processes that take a long time, it can be painful.



It's a problem that we frequently quote-unquote solve with spinners.



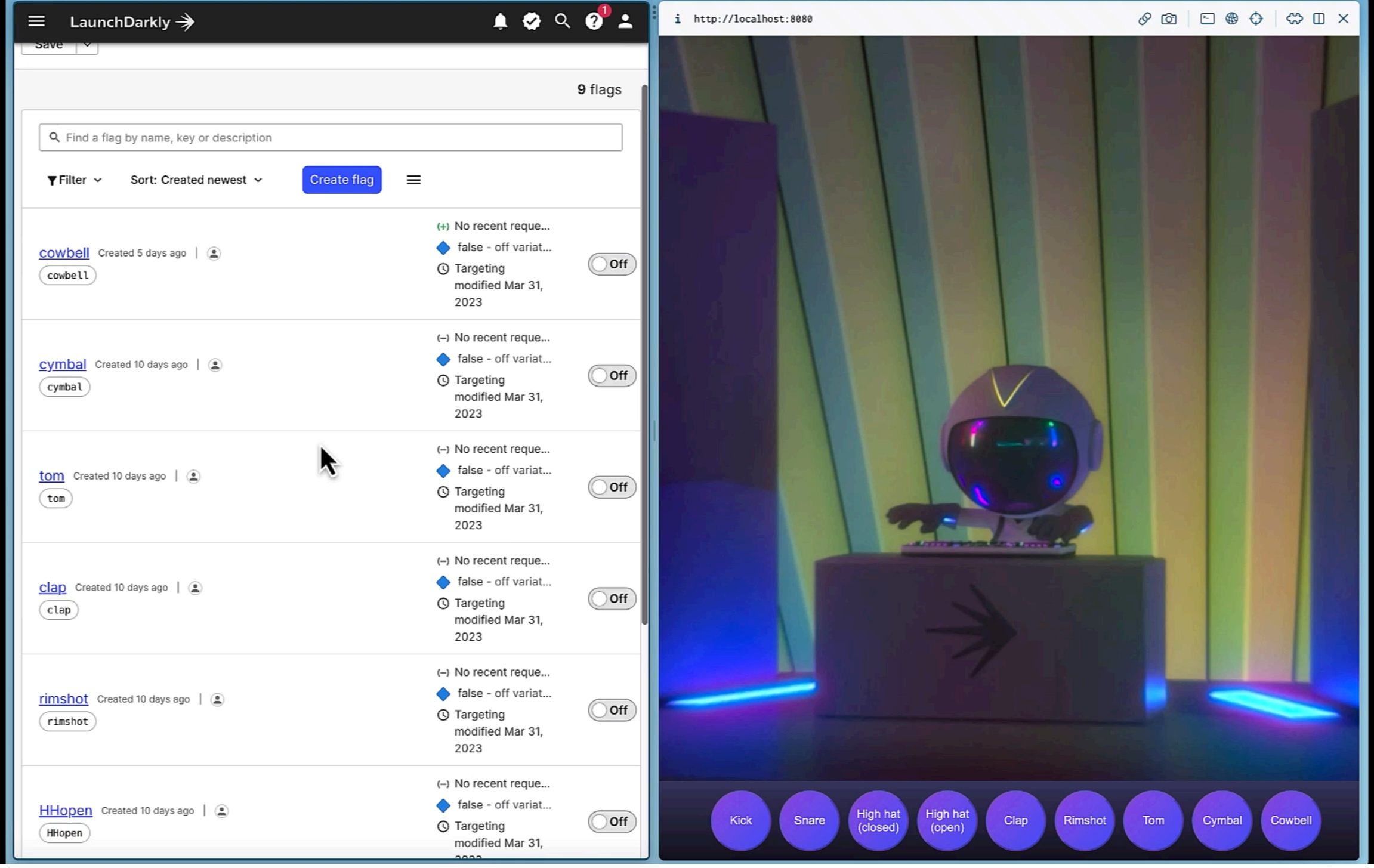
Take ChatGPT for example. It can take minutes for a response to fully generate from ChatGPT. This is a simulated experience of how that would feel for a simple prompt. Yeah, not great.



But that's not how ChatGPT or other similar systems work of course. Instead, they send out the response as it is being generated. While the process takes the same amount of time, the user experience is much improved.

But it's not just about server load times. In some cases, once the response is sent, something has changed on the server but the client remains unaware so the change isn't reflected.

And it's not just about server load times. Sometimes, we need to provide the client with information that has changed or updated. A prototypical example might be a news feed, social media feed or stock ticker.



Or, in the case of LaunchDarkly, we want you to get feature flag updates almost instantaneously. For example, if you want to use feature flags to DJ.

Joking aside, this is a demo application I wrote that let's users create a DJ mix to illustrate how flag updates are nearly instantaneous - typically within 20 milliseconds to all connected clients. The point here is that changes on the LaunchDarkly dashboard are reflected in the client almost instantaneously.

POTENTIAL SOLUTIONS

» Polling

» Long Polling

» WebSockets

POLLING

Pros

» Easy to implement - just a basic setInterval().

» Works across browsers/devices since it's old school.

Cons

» Inefficient since fetch requests are made even when no new data may exist.

» Delayed updates because data may have changed but the update is determined by the polling interval.

LONG POLLING

Pros

» More responsive than traditional polling because server holds the request open for new data.

» Easy to implement on the client.

Cons

» High server load due to holding connections open and need to reopen connections.

» Complex implementation on the server to ensure data isn't lost during reconnection.

Connections held open for extended periods

WEBSOCKETS

Pros

» Real-time bidirectional communication on a long lived connection.

» Low latency - once the connection is established, data is transmitted instantly.

» Very efficient since there's no need for things like HTTP headers.

Cons

» Complex implementation (most folks use a library).

» Connection failures, especially on mobile in the case of unstable or unreliable network connections.

The overhead and complexity of WebSockets may not be worthwhile. For example, all of the use cases we discussed above only require one way communication from the server to the client.

WHAT ABOUT SERVER SENT EVENTS (SSE)?

Server sent events are often overlooked but are a perfect choice for these kinds of use cases. As we'll see, they balance the simplicity of polling with the real time qualities of WebSockets

The basics of server sent events are simple. The client establishes a connection. Once the connection is established, the server can stream events back to the client until such time as the stream is closed or disconnected.

SEVER SENT EVENT AREN'T NEW

```
» Proposed in 2004

» First implemented in 2006 (in Opera)

» Official in 2011

» No Edge support until 2020

» Currently supported in ~97% of browsers globally
   (per CanIUse.com)
```

Server sent events have actually been around for a long time. They were often overlooked though because of a lack of Edge support and because they lived in the shadow of websockets, which can do achieve the same result via two way communication channel.

However, recently server sent events have had a surge in popularity due in part to interfaces like ChatGPT using them and other interfaces trying to replicate that user experience but also because other popular APIs, including ChatGPT's, have added support for streaming responses.

```
const evtSource = new EventSource("/my/api");

evtSource.onmessage = (event) => {
    const type = event.type;
    const data = event.data;

  // do something with the data
};
```

The key to consuming server sent events is the eventsource interface which connects to an event stream on the server side. This example listens to any event sent from the stream, which is at the URL provided to the EventSource, which is the only property EventSource accepts.

```
const evtSource = new EventSource("/my/api");

evtSource.addEventListener("message", (event) => {
    const { type, data } = event;

    // do something with the data
}
```

A better example might be to instead add event listeners to the stream rather than process all events. Events coming from the stream all have both a data and a type, so you can listen for the right types from a single stream which can send multiple types. This means that you can have different actions occur on each type. For example, one might bring new data, one might bring updated data and another might remove data. Message in this case will capture events without an event field and events with a specific type of message

```
evtSource.addEventListener("gpt", (event) => {
    const { type, data } = event;

    // do something with the data
}

evtSource.addEventListener("mastodon", (event) => {
    const { type, data } = event;

    // do something with the data
}
```

You can listen for multiple different event types from a single eventsource stream and respond differently to each one.

SENDING EVENTS FROM THE SERVER

```
event: message\n
data: {"name": "Brian, "msg": "Ping!"}\n
id: 9\n\n
```

Data needs to be formatted in a very particular way with the data prefix and the data in a single line string. You can still send JSON in the data response. The event and id are optional, but, as we saw, the sending the event type can allow you to customize how you respond to different data events. The ID is also optional but can help you determine how to respond, for example if you've seen this update already.

SENDING EVENTS FROM THE SERVER

```
const res = new Response(data, {
    status: 200
    headers: {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
    }
});
```

You also need to make sure that you set the proper headers for a long lived streaming response.

SENDING EVENTS FROM THE SERVER

```
for await (const part of stream) {
  const data = part.content || "";
  res.write(`data: ${JSON.stringify(data)}\n\n`);
}
```

Then you'll have some processs that continuously sends the data response as new data becomes available and appends that to the response.

This all seems pretty straightforward in theory and maybe I'm an idiot but I personally couldn't get it to work right.

BETTER-SSE

GitHub: https://github.com/MatthewWid/better-sse
Created by Matthew Widdicombe

```
npm install better-sse
```

The better sse project saved me a great deal as it just worked and the API made it dead simple to create and append the proper response on the server.

BETTER-SSE

```
const { createSession } = require("better-sse");

app.get("/gpt", async (req, res) => {
  const query = req.query.query;
  const session = await createSession(req, res);

  res.sse = session;
  const stream = await openai.chat.completions.create(...);

  for await (const part of stream) {
    res.sse.push(part.choices[0]?.delta?.content || "", "gpt");
  }
});
```

WHY SERVER SENT EVENTS?

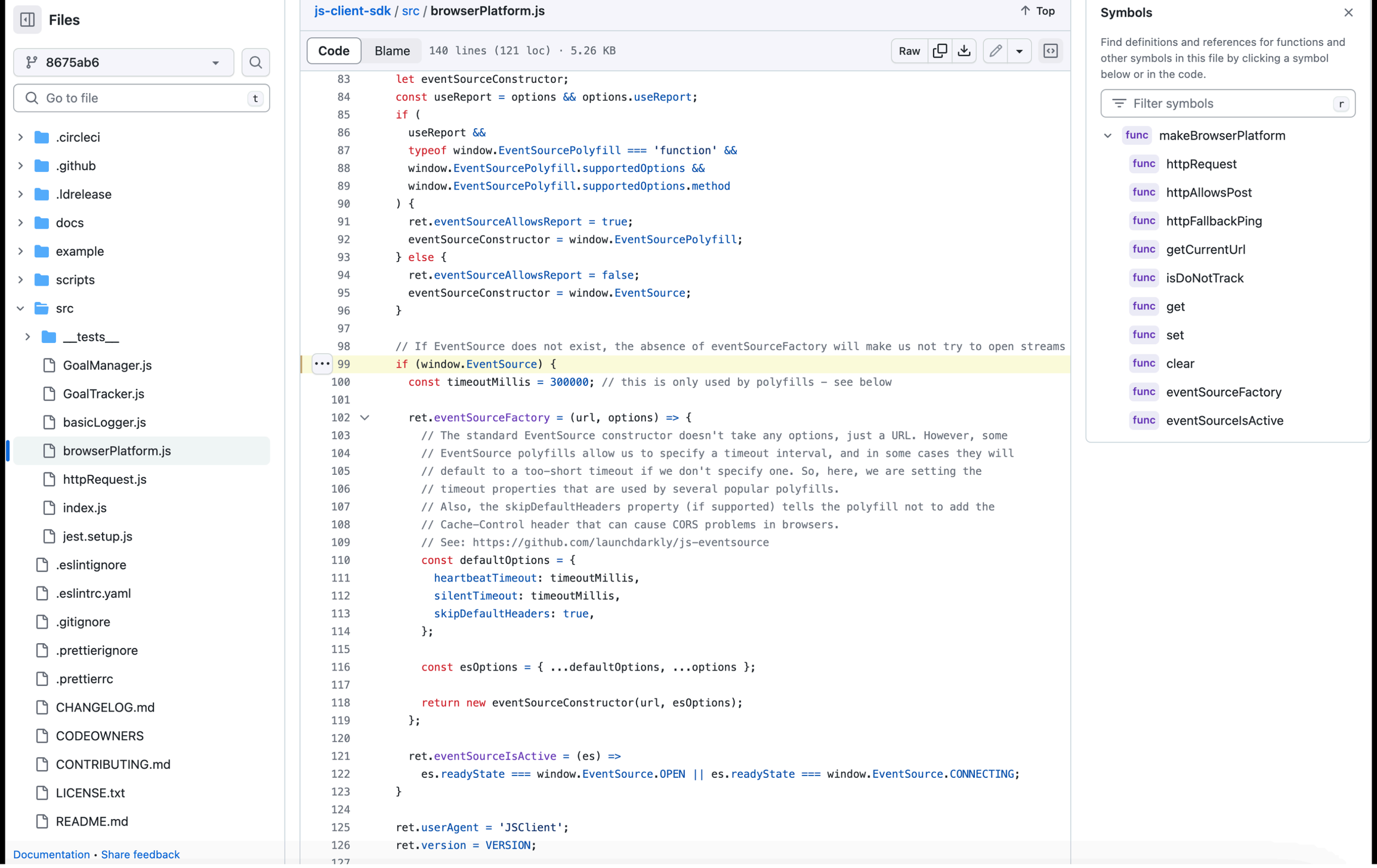
» Simple API

» Secure (read only)

» Easy to create on the server (HTTP rather than TCP)

» Automatic reconnection

So, if WebSockets and server sent events can accomplish the same reads but websockets add the ability to write, why would you choose server sent events? First, as you've seen, it offers a really simple API. If you primarily need to read and occassionally write, websockets can be more complex. Plus, the fact that they are read only makes it more secure. And there is no need to implement TCP because server sent events just communicate over HTTP and if the connection is interrupted, it will automatically attempt a reconnect.




```
83    let eventSourceConstructor;
84    const useReport = options && options.useReport;
85    if (
86      useReport &&
87      typeof window.EventSourcePolyfill === 'function' &&
88      window.EventSourcePolyfill.supportedOptions &&
89      window.EventSourcePolyfill.supportedOptions.method
90    ) {
91      ret.eventSourceAllowsReport = true;
92      eventSourceConstructor = window.EventSourcePolyfill;
93    } else {
94      ret.eventSourceAllowsReport = false;
95      eventSourceConstructor = window.EventSource;
96    }
97
98    // If EventSource does not exist, the absence of eventSourceFactory will make us not try to open streams
99    if (window.EventSource) {
100     const timeoutMillis = 300000; // this is only used by polyfills - see below
101
102     ret.eventSourceFactory = (url, options) => {
103       // The standard EventSource constructor doesn't take any options, just a URL. However, some
104       // EventSource polyfills allow us to specify a timeout interval, and in some cases they will
105       // default to a too-short timeout if we don't specify one. So, here, we are setting the
106       // timeout properties that are used by several popular polyfills.
107       // Also, the skipDefaultHeaders property (if supported) tells the polyfill not to add the
108       // Cache-Control header that can cause CORS problems in browsers.
109       // See: https://github.com/launchdarkly/js-eventsource
110       const defaultOptions = {
111         heartbeatTimeout: timeoutMillis,
112         silentTimeout: timeoutMillis,
113         skipDefaultHeaders: true,
114       };
115
116       const esOptions = { ...defaultOptions, ...options };
117
118       return new eventSourceConstructor(url, esOptions);
119     };
120
121     ret.eventSourceIsActive = (es) =>
122       es.readyState === window.EventSource.OPEN || es.readyState === window.EventSource.CONNECTING;
123   }
124
125   ret.userAgent = 'JSClient';
126   ret.version = VERSION;
```

As I indicated earlier, LaunchDarkly's SDKs rely heavily on server sent events. Once the SDK is initalized it caches all of the flag information locally and then creates a EventSource stream as the default method for sending updates. This is how you see the client change almost immediately after you make a change in the LaunchDarkly dashboard.

```
83        let eventSourceConstructor;
84        const useReport = options && options.useReport;
85        if (
86          useReport &&
87          typeof window.EventSourcePolyfill === 'function' &&
88          window.EventSourcePolyfill.supportedOptions &&
89          window.EventSourcePolyfill.supportedOptions.method
90        ) {
91          ret.eventSourceAllowsReport = true;
92          eventSourceConstructor = window.EventSourcePolyfill;
93        } else {
94          ret.eventSourceAllowsReport = false;
95          eventSourceConstructor = window.EventSource;
96        }
97
```

If you were paying super close attention to the prior slide, you may have noticed that the JavaScript client SDK uses a server sent events polyfill. While we discussed that server sent events currently has 96% support across browsers, something like the LaunchDarkly SDK needs to support legacy browsers and unsupported browsers.

launchdarkly-eventsource

2.0.1 • Public • Published a month ago

Readme | Code Beta | 0 Dependencies | 9 Dependents | 14 Versions

EventSource

npm v2.0.1 | NO BUILDS | downloads | package not found or too new

Dependencies

This library is a pure JavaScript implementation of the EventSource client. The API aims to be W3C compatible.

You can use it with Node.js, or as a browser polyfill for browsers that don't have native `EventSource` support. However, the current implementation is inefficient in a browser due to the use of Node API shims, and is not recommended for use as a polyfill; a future release will improve this.

This is a fork of the original EventSource project by Aslak Hellesøy, with additions to support the requirements of the LaunchDarkly SDKs. Note that as described in the changelog, the API is *not* backward-compatible with the original package, although it can be used with minimal changes.

Install

```
npm install launchdarkly-eventsource
```

Example

```
npm install
node ./example/sse-server.js
node ./example/sse-client.js    # Node.js client
open http://localhost:8080      # Browser client - both native and polyfill
curl http://localhost:8080/sse  # Enjoy the simplicity of SSE
```

Install

```
npm i launchdarkly-eventsource
```

Repository
github.com/launchdarkly/js-eventsource

Homepage
github.com/launchdarkly/js-eventsource

Weekly Downloads
687,523

Version
2.0.1

License
MIT

Unpacked Size
404 kB

Total Files
23

Issues
0

Pull Requests
0

Last publish
a month ago

Collaborators

The polyfill we use is actually open source. It's a fork of an existing polyfill, but as the docs note that it is primarily a Node polyfill.

Wait...

SERVER SENT EVENTS ON THE SERVER?

This brings up the point that, although server sent events are a browser API, there are a number of implementations for server-to-server communication, including the Node.js implementation that I showed a moment ago.

Let's say you need to send mostly one way updates from one server to another, server sent events can be a way to do that, provided there's an implementation for the language you wish to use. LaunchDarkly uses this across our server-side SDKs for streaming updates to flag rules from server to server so that you get the same immediate response on the server side as you do on the client.

LAUNCHDARKLY OPEN SOURCE SSE LIBRARIES

» .NET: https://github.com/launchdarkly/dotnet-eventsource

» Ruby: https://github.com/launchdarkly/ruby-eventsource

» Python: https://github.com/launchdarkly/python-eventsource

» Java: https://github.com/launchdarkly/okhttp-eventsource

» Swift: https://github.com/launchdarkly/swift-eventsource

» Rust: https://github.com/launchdarkly/rust-eventsource-client

In fact besides the JavaScript implementation, LaunchDarkly has a bunch of other server-side and native mobile implementations of server sent events all open source.

EXAMPLES

Alright, let's look at some example code

SLIDES AND CODE

https://github.com/remotesynth/server-sent-events

QUESTIONS?

Email: brinaldi@launchdarkly.com
Mastodon: https://mastodon.xyz/@remotesynth