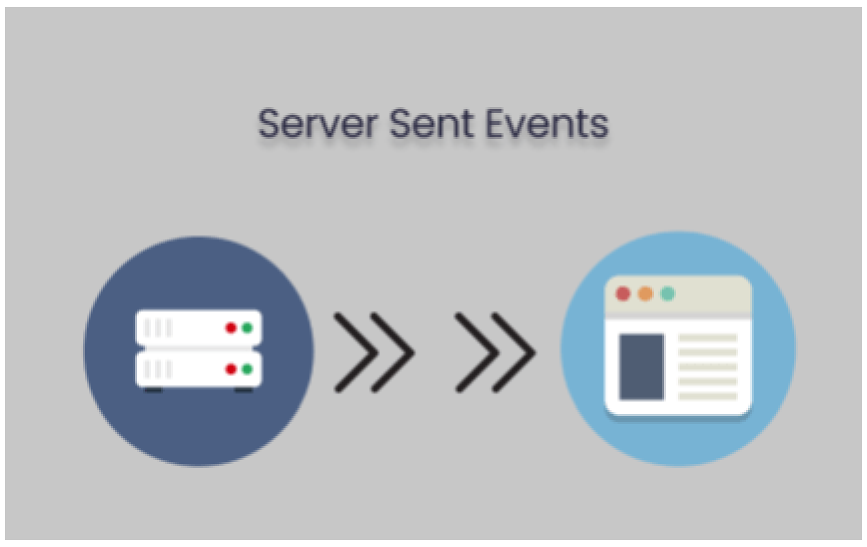
ABHISHEK MEHANDIRATTA 9 JAN 2021

An intro to server sent events



Server sent events allow us to establish a unidirectional communication channel between clients and the server. Over this channel, the server can push events(which are just plain text messages) that the client can receive and process in real time.

We'll be experimenting with server sent events in this article. You can refer to the github repo here.

In order to use server sent events, we need to establish a long lived HTTP connection from the client with the server. The client must accept `text/event-stream` in response to this request. Remember that we don't want the server to send back the response immediately because the connection would be closed in that case. This will be clear when we look at an example in `node.js`.

If you've heard of websockets, SSE is very similar to that. There are a few key differences though.

1. Websockets are bidirectional, SSE is unidirectional.
2. Websockets can send binary or plain text messages. SSE only supports plain text messages.
3. Both are supported in almost all major browsers. Websockets have received more attention though.
4. Websockets are lower level and more complex to set up as compared to SSE. There are less pitfalls when using SSE.

Let's jump into the code. We'll be using `typescript and express js`. Start off by installing all the dependencies.

```
mkdir sse-test && cd sse-test
yarn init -y # init the project
yarn add express nanoid # install deps
yarn add -D @types/express @types/node nodemon typescript # install types
```

In order for this to be a typescript project, we need to have a tsconfig.json. To generate it easily, run

```
npx tsconfig.json # then select node and press enter
```

The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree

```
import { join } from 'path'

const app = express()
app.use(express.json())

const PORT = +process.env.PORT! || 3000

app.use(express.static(join(__dirname, '../public')))

app.listen(PORT, () => console.log(`> http://localhost:${PORT}`))
```

Now make a file in the root of the project named `public/index.html`.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>SSE App</title>
</head>

<body>
  <p>Home page</p>

  <script>
    console.log('hello world')
  </script>

</body>

</html>
```

Now open `package.json` and make the following scripts to easily start the server

```
// ...
"scripts": {
    "watch": "tsc -w",
    "dev": "nodemon dist/index.js"
},
// ...
```

Open 2 terminal windows and run `yarn watch` (leave it running) in one and `yarn dev` (leave it running as well) in the other window. Open localhost:3000 and you'll be good to go.
Let's set up our express route to handle SSE. In the file `src/index.ts`, paste

```
app.get('/events', (req, res) => {
  console.log(req.headers)
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive',
    'Cache-Control': 'no-cache'
  })
  res.write(`data: “Hello”\n\n`)
  req.on('close', () => {
    console.log(`connection closed by client`)
  })
  // see that we don't do res.json() or res.send() here
```

The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree

be an event stream and the connection header is set to keep alive so that the connection is not terminated. Event format in SSE follows a specific pattern. It is as follows:

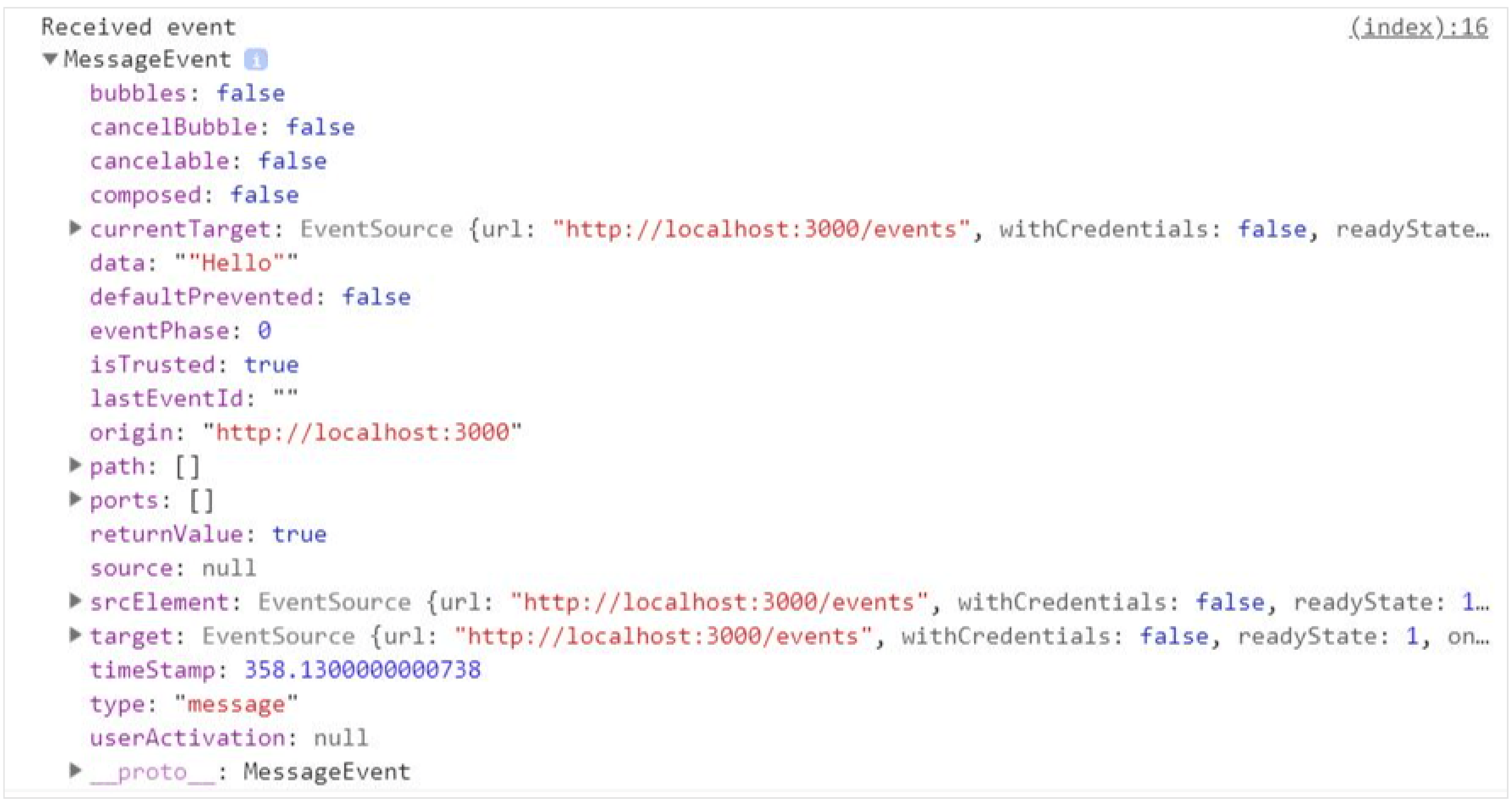
```
event: <name of your event>\n
data: <string, event data>\n
retry: <in case connection is closed, after how many ms to retry this>\n
id: <id of the message>\n\n
```

Every message is terminated by `\n\n`. An event usually has some data associated with it. All other fields are optional.

Now let's make sure frontend subscribes to this stream and receives events.
In `public/index.html`

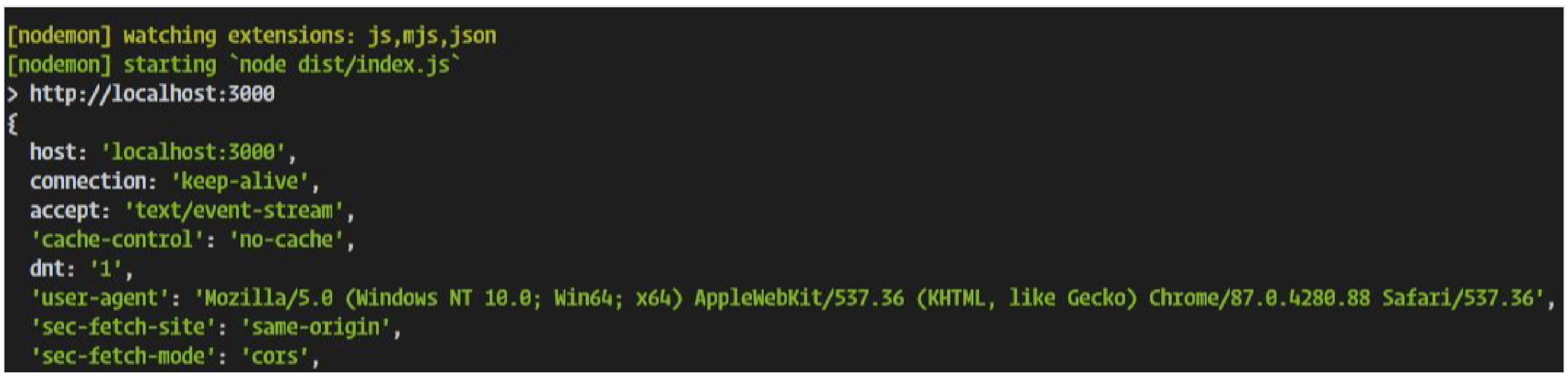
```
<script>
    const eventSource = new EventSource('/events')
    eventSource.onmessage = event => {
      console.log('Received event', event)
    }
</script>
```

When you run this, you will see some message on the console like so:



```
Received event                                                                    (index):16
▼MessageEvent
    bubbles: false
    cancelBubble: false
    cancelable: false
    composed: false
  ▶ currentTarget: EventSource {url: "http://localhost:3000/events", withCredentials: false, readyState…
    data: ""Hello""
    defaultPrevented: false
    eventPhase: 0
    isTrusted: true
    lastEventId: ""
    origin: "http://localhost:3000"
  ▶ path: []
  ▶ ports: []
    returnValue: true
    source: null
  ▶ srcElement: EventSource {url: "http://localhost:3000/events", withCredentials: false, readyState: 1…
  ▶ target: EventSource {url: "http://localhost:3000/events", withCredentials: false, readyState: 1, on…
    timeStamp: 358.1300000000738
    type: "message"
    userActivation: null
  ▶ __proto__: MessageEvent
```

On the server console, you can see the headers sent by the client:



```
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node dist/index.js`
> http://localhost:3000
{
  host: 'localhost:3000',
  connection: 'keep-alive',
  accept: 'text/event-stream',
  'cache-control': 'no-cache',
  dnt: '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/87.0.4280.88 Safari/537.36',
  'sec-fetch-site': 'same-origin',
  'sec-fetch-mode': 'cors',
```

The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree

The `EventSource` API used in `index.html` sets the client headers to accept event stream. You can also use the `fetch API` as well for SSE by manually setting these headers.

Let's see how we can broadcast messages across multiple clients. Make a file `src/types.ts`

```
import { Response } from 'express'

export type CustomEvent = {
  data: Object
  id: string
  type: string
  retry?: number
}
export type Client = {
  id: string
  res: Response
}
```

Then in `src/index.ts`

```
import { CustomEvent, Client } from './types'
import { nanoid } from 'nanoid'

let clients: Client[] = []

app.get('/events', (req, res) => {
  console.log(req.headers)
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Connection': 'keep-alive',
    'Cache-Control': 'no-cache'
  })

  const client: Client = {
    res,
    id: nanoid(10)
  }
  sendEventToAll({
    type: 'join', // event type is join
    data: {
      joined: client.id
    },
    id: nanoid(10)
  })
  clients.push(client)
  req.on('close', () => {
    console.log(`connection closed by client ${client.id}`)
    clients = clients.filter(c => c.id !== client.id)
  })
})
// send to all connected clients
```

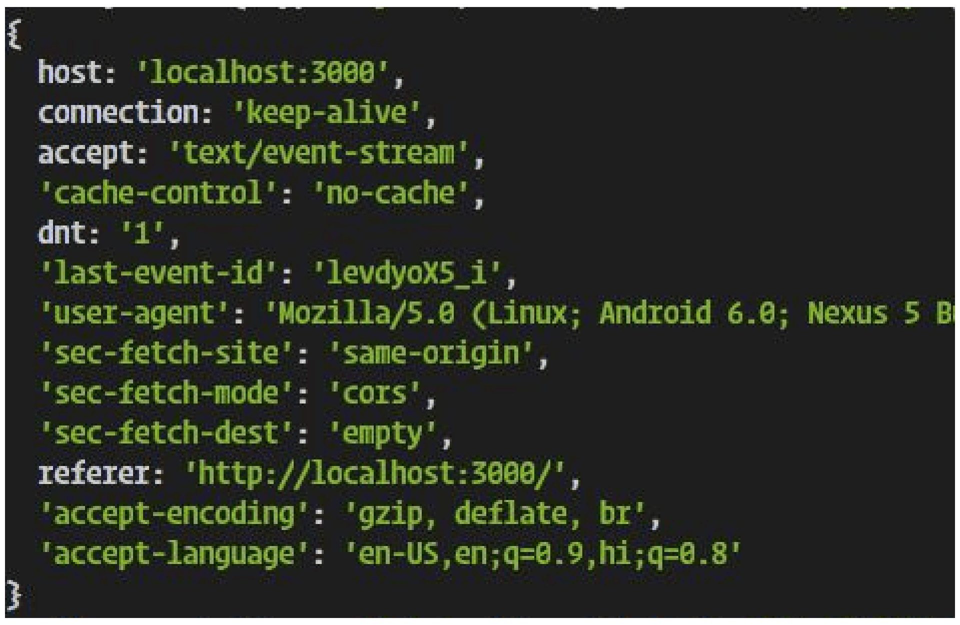
The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree

```
      eventString += `retry: ${event.retry}\n`
    }
    eventString += `id: ${event.id}\n\n`
    c.res.write(eventString)
  })
}
```

It is pretty self explanatory. We store every client in an array of clients, where every client has an id and an express response handle so that we can write out events to them.
The most notable thing here is the use of custom event names. We specify these in the `event: ${event.type}` line. To receive custom events on frontend, you can use `addEventListener` on eventSource object like so:

```
eventSource.addEventListener('join', event => {
    console.log('Join received', event)
})
```

Now open two browser windows to see this in action.
The cool thing about SSE is that when you associate an id with every event, the browser automatically sends the last event id to the server in case the connection is dropped by the server. This way, the server can send all remaining messages to the client so it can catch up. To see this in action, open two browser windows again. After that, restart the server by typing `rs` and then enter in `nodemon`. As soon as the client reconnects to the server, you'll see the header `last-event-id` sent with `/events` request.



```
{
  host: 'localhost:3000',
  connection: 'keep-alive',
  accept: 'text/event-stream',
  'cache-control': 'no-cache',
  dnt: '1',
  'last-event-id': 'levdyoXS_i',
  'user-agent': 'Mozilla/5.0 (Linux; Android 6.0; Nexus 5 B
  'sec-fetch-site': 'same-origin',
  'sec-fetch-mode': 'cors',
  'sec-fetch-dest': 'empty',
  referer: 'http://localhost:3000/',
  'accept-encoding': 'gzip, deflate, br',
  'accept-language': 'en-US,en;q=0.9,hi;q=0.8'
}
```

You can use SSE in situations when the server needs to communicate to the client in real time. For example, in-app unread notification count, live update of likes/comments on posts, real-time flow of data in admin dashboards etc. However be wary of the biggest limitation of SSE, which is unidirectional communication.

I hope you found this introduction informative. If you have any doubts, feel free to comment below.

Share : Facebook Twitter Linkedin Whatsapp

Tags : #Node.js #Express.js #SSE

The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree

3 Comments Sort by Newest

Sebastian Neumair 3 days ago

Thanks for the insightful intro. Do you have any idea why all the listeners I am getting from the EventSource (instance of EventSource) are null?

onerror: null
onmessage: null
onopen: null
readyState: 1
url: "http://localhost:8000/events"
withCredentials: false

Reply

Abhishek Mehandiratta Admin Sebastian Neumair 3 days ago

If this is the case, you must not be receiving any events from the server. Have a look at the github repo associated with this post
https://github.com/abhi12299/sse-intro
console.log your EventSource object once you've registered all event listeners. It should hopefully work. If it doesn't, let me know.

Reply

Sebastian Neumair Abhishek Mehandiratta 2 days ago

thank you, the culprit interestingly was the html... thanks for the great article btw :)

Reply +1

ADD WIDGETPACK TO YOUR SITE POWERED BY **WIDGET PACK™**

Home About

Blog Contact

The website uses cookies 🍪. If you continue to use the website, I assume you are okay with it. I agree