Server Sent Events — Architecture and Implementation

Teja Swaroop Mylavarapu Follow 5 min read · Apr 20, 2020

79 3

SSE — An interesting architecture that has been making rounds for quite some time.

So, what is SSE?

In layman terms, SSE is a Server Sent Event where the UI first initiates the call and requests for data, then the Server sends back the data to the caller (UI) when the data is available.

When an HTTP connection is made from the UI, the server holds the connection and responds back with the data when and ever it gets an event/data from the downstream APIs/sources.

Unlike a normal HTTP call where the Server sends back the response immediately and close the connection, in the case of SSE, the Server holds the connection for a period of time and emits back the response as Streams when it's available. It can send packets of data as a stream one after the other with a single HTTP Connection. Once all the data is sent, the Server then closes the connection.

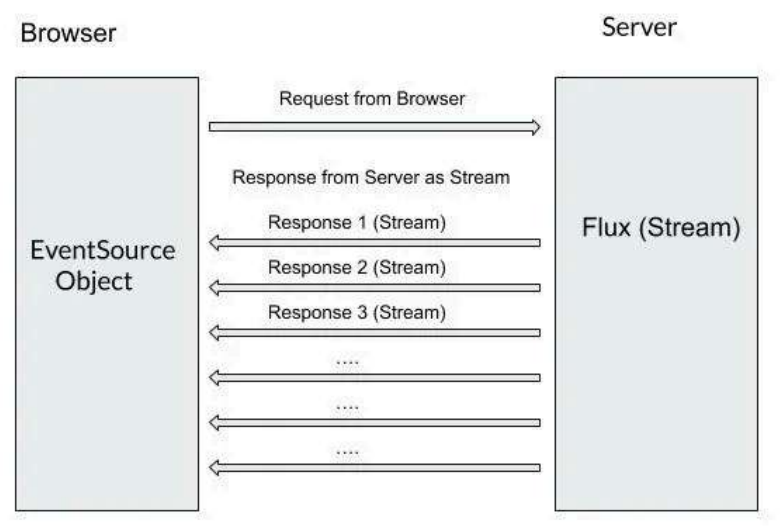


Image 1. Server Sent Events Request-Response Diagram

As depicted in the diagram above, the Server sends the responses as a Stream of data in the form of Flux. One single request from the Browser will emit multiple responses (JSON type) back in the same HTTP (HTTPS) call.

The Browser however has to implement the request as an EventSource Object from JavaScript. This Object is present in all the major browsers by default except for Internet Explorer (I know what you feel about this…!!).

Why Server Sent Events?

Server Sent Events (SSE) are used in scenarios where the browser constantly looks for updates from the Server and the Server emits out the responses when and if anything is available on its side. Instead of the Browser making continuous requests (Long Polling) again and again, the Browser just makes 1 request and the Server sends back the response/responses whenever it is available.

Long Polling Scenario: If the Server does not have the data available when the Browser request comes in, usually it sends back empty data and the Browser makes the request again after some time to check if the data is available and the server responds appropriately and this loop can go on forever if the Server does not have the data.

Trust me, I have seen around 25–30 Round Trip calls being made in these scenarios where the Browser constantly kept checking for updates on the Server with no luck. This scenario is pretty draining and proves to be very cumbersome on the Browser's resources with multiple calls being made repeatedly.

The architecture and SSE completely eliminates the concept of Long Polling where the Browser keeps making multiple requests to the Server.

Now, with a basic understanding of what is SSE, let us dig a little bit on the coding perspectives and how this can be achieved.

Tech Stack used:

1. JavaScript for the UI
2. Java for the Backend
3. Apache Kafka as data consumer

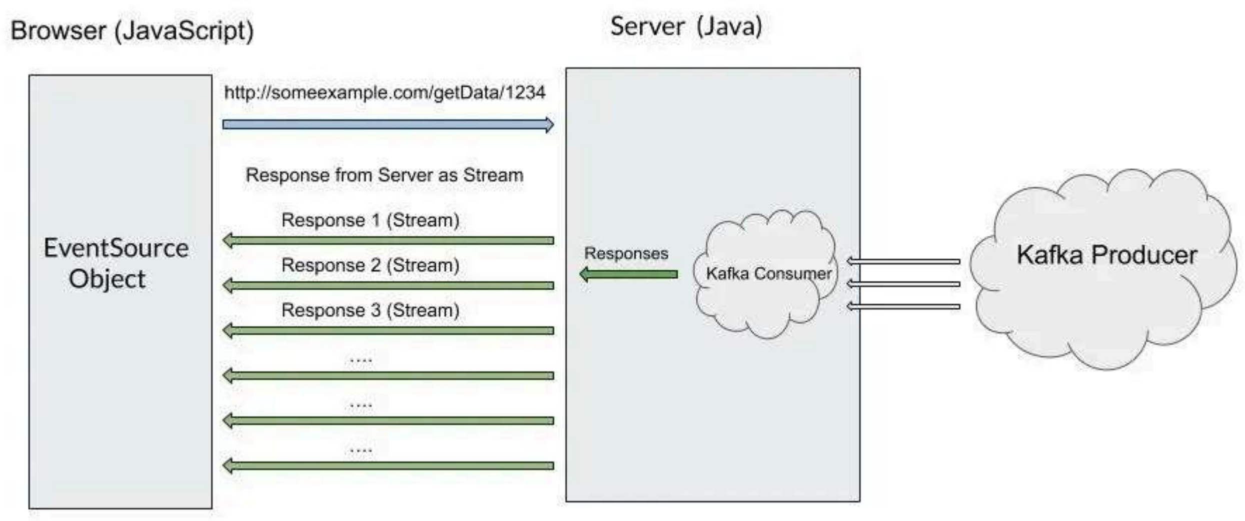


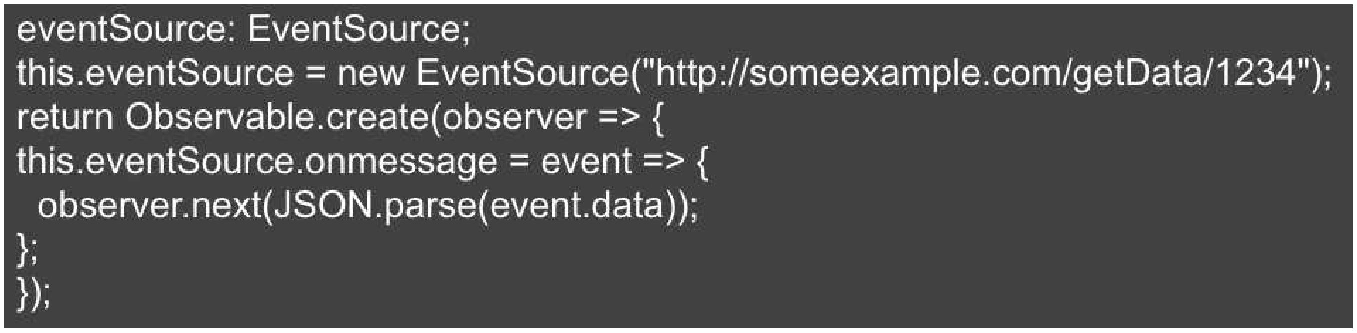
Image 2: Real time SSE Implementation

SSE Implementation:

Step 1: **JavaScript for Browser**

A Browser makes the call to the SSE Server through the **EventSource Object.** EventSource Object is the JavaScript implementation of SSE. With an EventSource Object, the browser makes the call to the Server and holds on to the call till the Server terminates the call. For any reason (ALB Timeout, Network disruption), if the connection from the UI to the backend breaks, the EventSource object automatically reconnects to the Server. Let me tell you, this is automatic, no extra manual or coding intervention needed.

JavaScript Snippet for SSE:



```
eventSource: EventSource;
this.eventSource = new EventSource("http://someexample.com/getData/1234");
return Observable.create(observer => {
this.eventSource.onmessage = event => {
  observer.next(JSON.parse(event.data));
};
});
```

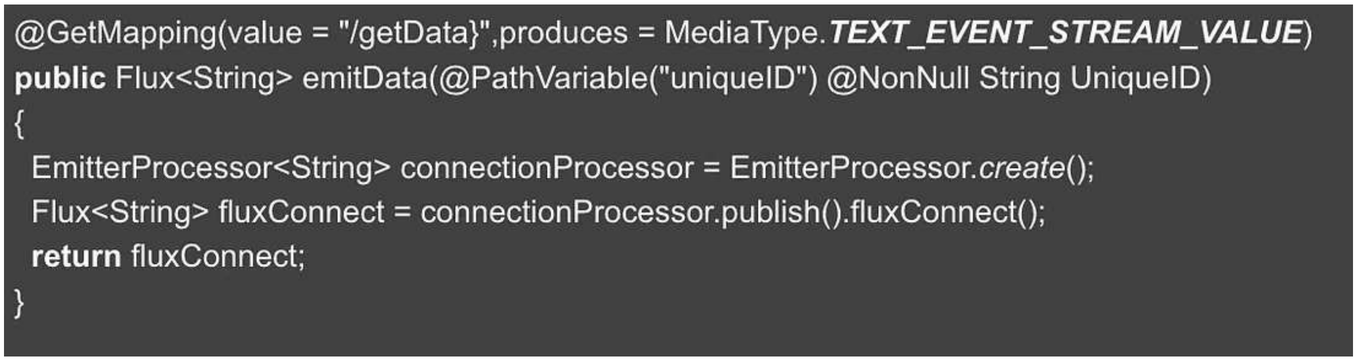
Step 2: **SSE on the Server (Java)**

a. SSE events are sent back in the form of Streams. So we have to make sure that our code/API responds back with the response type as “text/event-stream”.

b. SSE is implemented based on the connection from the UI. When a connection request comes in, the Server needs to hold on to the connection till a response is obtained. It can be implemented in Java using “EmitterProcessor”. It is a type of processor that can be used with several subscribers. This EmitterProcessor is the processor which holds the unique one time connection with the Browser.

c. The response type of the data sent back to the Browser is of type Flux. Flux is the object that is responsible in streaming the data back to the user.

Both Flux and EmitterProcessor are built on Java’s Reactor core Publisher API. Reactor Core is a Java library which implements the reactive programming model.

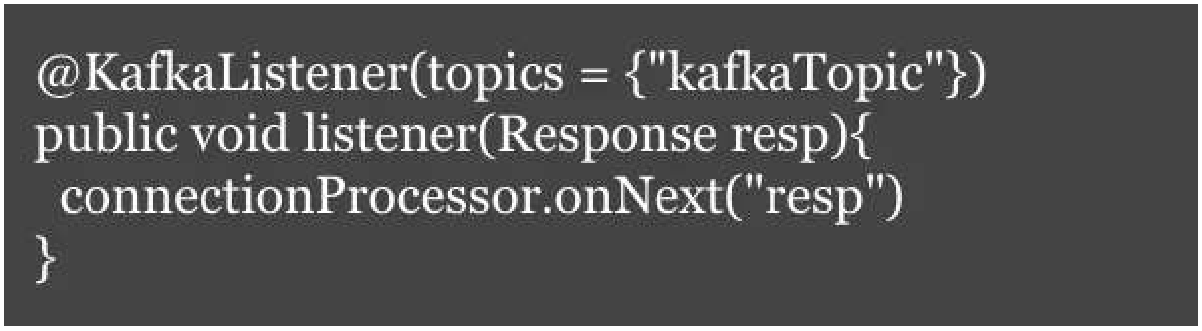


```
@GetMapping(value = "/getData}",produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> emitData(@PathVariable("uniqueID") @NonNull String UniqueID)
{
  EmitterProcessor<String> connectionProcessor = EmitterProcessor.create();
  Flux<String> fluxConnect = connectionProcessor.publish().fluxConnect();
  return fluxConnect;
}
```

The above code depicts on how the GET call is built in Java and how the response type is of event-stream value, the connection received and the Flux events getting sent back to the caller (Browser).

Step 3: **Kafka Consumer, sending SSE events**

Once the connection is established, the Server holds the connection and waits for data. Here, we are reading the events from an Apache Kafka Consumer. Once the Kafka Producer produces an event for the topic which our Kafka Consumer subscribed, the data/events will be consumed by our Consumer.



```
@KafkaListener(topics = {"kafkaTopic"})
public void listener(Response resp){
  connectionProcessor.onNext("resp")
}
```

The data is read and if there is an SSE connection that exists, the data is emitted back to the Browser. This process is repeated till the Server terminates the connection.

Things to be aware of:

While using AWS, the connection from the Browser to the Server gets disconnected based on the Application Load Balancer's (ALB) timeout. If the timeout is configured for 180 seconds, the connection gets disconnected after 180 seconds and the browser automatically reconnects to the Server.

I hope this article of mine has helped you understand the concepts of Server Sent Events. Happy Coding…!!

If you like this article, please do Clap hands at the bottom of the page as many times as you can. It encourages me to write more.

Sse Server Sent Events Java Browsers Event Source

Published in Level Up Coding

257K followers · Last published 4 days ago

Follow

Coding tutorials and news. The developer homepage gitconnected.com && skilled.dev && levelup.dev

Written by Teja Swaroop Mylavarapu

79 followers · 18 following

Follow

Lead Software Engineer at Capital One

Responses (3)

Pankaj Chopra

S Ghosh
Aug 10, 2020

What happens when the server is about to send the event and the browser gets disconnected at that moment? Will the browser miss that event?

1 1 reply Reply

Antonino Barila
Sep 26, 2023

Hello,

in context with multiple POD where the message of kafka can be processd from a pod that don't have open a SSE, How do you manage this ?

Reply

Barathvasu
Jan 16, 2021

Good Article explaining about SSE!! When we take a real time scenario, we will have multiple servers hosting the same application, if we use kafka how will be able to consume from kafka to the same request/jvm from client unless we create multiple