Leveraging Server-Sent Events for Enterprise-Scale Real-Time Notifications: A Spring Boot Implementation

Ananth Majumdar

Email: *thisisananth[at]gmail.com*

Abstract: ***Users expect real-time updates for the information they are interested in for any web application. This doesn't align well with the request response architecture of the web. There are many solutions to address this need like HTTP Polling, long polling, server sent events and web- sockets. This paper investigates the benefits and drawbacks of these technologies with a focus on simplicity and scalability. This paper also talks about a real-world implementation of server sent events using Spring boot and the practical considerations. It also discusses the results of the implementation and any further improvements.***

Keywords: real-time updates, web application, server sent events, Spring Boot implementation, technology comparison

1. Introduction

Modern websites need real-time communication between client and servers for getting updates for time-critical things. The default request-response model is not sufficient for providing real time up- dates as the client needs to initiate the requests. Over time multiple solutions have been created for these.

Frequently polling the server provided somewhat of a real-time experience but it was wasteful of resources to create continuous requests even with empty responses. Long polling came into the pic- ture to solve the problem of continuous requests when the response is empty. Even this is wasteful due to the need to open and close multiple connections. Both these solutions don't address the basic problem of having to create multiple connections to simulate real time updates

To overcome these solutions were created where a persistent connection is created to overcome the overhead of opening and closing connections. These include server sent events for unidirectional up- dates from the server to client and web- sockets for bidirectional updates between the server and client. These are better solutions utilizing resources more efficiently. This paper provides an overview of the different solutions their benefits and drawbacks. It also details a real-time implementation of server sent events using Spring Boot and how it helped in scaling an enterprise-wise notification system. It also talks about the practical aspects of deployment of SSE in production like security, CORS request, timeouts, handling multiple connections and the architecture for deploying the application in a cloud-based infrastructure.

2. Background

2.1 The Need for Real-time updates

The web is based on a request-response architecture. A client makes a request to a server and the server sends a response back to the client with the requested resource. This works well for a world in which the requested resource changes infrequently. But if the data changes frequently, then this breaks. The server cannot send an update to the client without the client making a new request to the server. The modern web has evolved to provide a broad range of applications that require frequent updates. e.g. web dashboards, collaborative editing.

Some of the solutions for this problem are
- HTTP Polling
- Long Polling
- Server Sent Events
- WebSockets

2.2 HTTP Polling

In this method, the client that needs real-time updates, regularly sends a request to the server even if the response is empty. The server responds with empty response if there is no update and with the correct response if there is something to be sent. It requires a full HTTP handshake every time the server sends something to the client. That wastes bandwidth and increases latency. Also, most of the requests are wasted since there is no update to be sent. This also means that the updates are only available when the next request from the client is sent, so there is still some delay in the update. But nevertheless, is simple to implement.

2.3 Long Polling

In long polling the client sends a request to the server and the timeout is set to be very long or indefinite. This way the client waits until a response is available or the long timeout is reached. If a response is available the server sends the response and closes the connection. If no response is available, the response is closed when the timeout is reached. But right after the request ends, a new request is created by the client. This way it simulates an always on connection with the server and hence can get an update almost real-time. This is an improvement over HTTP polling in that the updates can be delivered almost real-time since a persistent connection is maintained with the server most of the time. Also since the requests are held for a long time compared to HTTP polling, the number of bytes sent over the wire will also be less compared to HTTP polling.

Like HTTP polling, long polling also requires a full HTTP hand- shake with the server every time the server sends something to the client wasting bandwidth and increasing latency. Scaling long polling across multiple servers means keeping track of session state in a fundamentally stateless protocol. One option is to use a third-party data store such as Redis and to reload that state with each new HTTP request. Another is to implement sticky sessions, so the same server handles the same client each time. That brings additional complexity, as you'll need a strategy to ensure load is distributed evenly across your cluster. You'll need to create your own back-pressure mechanisms to monitor server load and adjust the rate of inbound requests by delaying or temporarily rejecting connections. [1]

2.4 Server-Sent Events

Instead of polling using a request/response model, HTTP 1.1 defines a streaming data transfer mechanism using the transfer-encoding: chunked header. With this it is possible to write multiple times into the response. With an implementation of this called 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. An EventSource instance opens a persistent connection to an HTTP server, which sends events in text/event-stream format. The connection remains open until closed by calling EventSource.close(). More details about implementing Server Sent Events are presented later. Server-sent events allow a client to get updates on demand from the server. This is only useful for uni-directional communication from server to the client. EventSource has automatic error handling where it reconnects to the server upon interruptions. It also has the overhead of HTTP headers attached to each message. [1]

2.5 WebSockets

Websockets make it possible to open a two-way interactive com- munication between a web browser client and a server. Messages can be sent and received between the server and client without polling the server. The protocol consists of an opening handshake by sending a GET request with upgrade: websocket header. If the server supports websockets, it responds with HTTP code 101 Switching Protocols and a wesocket connection is cre- ated. The client and server can send messages at any time. Each websocket messages called frames contains an opcode, payload size and a masking bit in addition to the data. These have a smaller footprint compared to the HTTP headers in the previous methods. Hence these are more suited for faster bi-directional communication between the client and the server. [7] The client/server can also decide to a higher-level subprotocol to define message semantics. To keep the connection alive between the client and server, once the connection is established either the server or client can send a ping to the other party and it should respond with a pong as soon as possible.

Websocket protocol allows extensions to modify the payload. It also allow sub-protocols to structure the payload into different formats. Most clients prefer to work with websockets with a sub- protocol like JSON or STOMP for easier handling of the messages. Hence this requires special server and client libraries and higher complexity compared to other methods.

3. Understanding Server-Sent Events

To enable servers to push data over HTTP, HTML defines an inter- face called EventSource. Using EventSource a client can connect to a server and listen to events to get data from the server. The server sends messages with the text/event-stream MIME type.

```
eventSouce =
new EventSource("/events/subscribe");
eventSource.onmessage = function(event)
{
console.log("New message", event.data);
}
```

The connection is persistent. It supports auto-connect. It is also based on HTTP and doesn't need to implement a new protocol. The server writes message in the following format

```
data: Message 1
data: Message 2
data: {"key":"value"}
```

Each message is delimited by characters to send a multi-line message you can do that by sending a new data: message after a single newline character. You can also send JSON messages

EventSource supports reconnection. If a connection is broken, the client automatically tries to reconnect. The server can set a delay using retry: time in ms message. The browser should wait that much time before reconnecting. While reconnecting we can take adavantage of the id field in each message. By sending an id field in each message, the browser sets the eventSource.lastEventId to that value and sends the Last-Event-ID header with the ID so that server can start resending the information following that id.

If the server wants the browser to stop reconnecting, it was do that by sending the 204 No content status. If the client wants to close the connection it can do so by calling the close() method on the EventSource.

EventSource also supports cross origin requests. Client sends the Origin header and the server should respond with the right Access- Control-Allow-Origin header. It also allows passing credentials with the withCredentials option like this

```
let source = new
EventSource("https://another-
site.com/events",
{withCredentials: true});
```

3.1 Spring Boot Server Sent Events Support

3.1.1 Spring Framework

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. It provides numerous features that make it easy to develop all kinds of applications using Java, Kotlin, Groovy and other scripting languages. The main features are dependency injection, events, re- sources, i18n, validation, data binding, type conversion, aspect oriented programing, support for testing with mock objects, data access support like transactions, DAO support, JDBC, ORM, web frame- works like Spring MVC and Spring WebFlux, various integration technologies like remoting, JMS, JCA, JMX, email, tasks, scheduling, cache and observability. [5]

3.1.2 Spring Boot

Spring Boot makes it easy to create stand- alone, production-grade Spring based applications with an opinionated view of Spring platform and third-party libraries. It provides an embedded server so the applications can be run a java jar files. It provides opinionated starter dependencies to simplify build con- figuration. It can automatically recognize libraries in the classpath and configure Spring. It provides production ready features such as metrics, health checks and externalized configuration. [4]

To support streaming HTTP responses, Spring framework has a class called **ResponseBodyEmitter** [2] . It can be used to send multiple objects where each object is written to the response with a compatible HttpMessageConverter. It has methods to send, complete, completeWithError and also methods to handle events like onError, onTimeout, onCompletion.

It also provides SSE support with an **SseEmitter** [6] class, a subclass of ResponseBodyEmitter. It provides utilty methods to create a new SseEmitter instance with a timeout and also various methods to send the response. It also provides a builder to add a data, id, message, event and comment objects to the SseEmitter object and also to specify the reconnectTime.

4. Implementation

We will describe the implementation of server sent events for an enterprise wide real-time notification system.

4.1 Requirements and constraints

We are implementing a company wide notification system to reduce the reliance on email for systematic informational notifications. For this we built a simple UI to show the notifications for a user and set it to run always. The notifications are personalized for each user. In the initial trial phase we started with 30 users. The initial implementation to get updates was the client polling the server every 30 seconds. Most of the times the users will not have any updates but still the client needs to poll the server every 30 seconds. All this network activity was wasteful and also adding unnecessary load on the server. To avoid this load and to help scale the application to support around 3000 users, we needed to use better ways to send updates to the client.

4.2 Reasons for choosing server-sent events

Websockets were an option for this but we didn’t want to support another protocol. This also meant we had to handle keeping connection alive, connection drops and resyncing any missing mesages. All these would also increase the complexity of what we were trying to do. We also need the client to get updates from the server and there was no need for bi-directional communication. So, we decided to go with server-sent events for this use-case.

4.3 Implementation Details

We are using Spring’s SseEmitter class to send the server sent events. We create one SseEmitter for each user connection. We keep all the active connections in a Concurrent HashMap keyed by the userid. The systems that are sending notifications use a REST API to send the notification with the list of recipient userIds. We are using Ama- zon Web Services’ Simple Notification Service (SNS) and Simple Queue Service (SQS) to handle the notifications and send updates to the recipients of the messages. When an integrating service sends a notification to the notification service, the service puts it on a topic. This topic is listened to by multiple queues, one of each con- tainer running the notification service. Once each container receives a message, it goes through the list of recipients and if there is an SseEmitter connected to that container instance, it will call the send method to send the message to that client. To reduce the complexity of keeping track of sent messages, the client still connects to the server every 15 minutes to get a full copy of the messages. This makes sure that even if some messages were lost during a disconnect- reconnect cycle, they will be received in the next 15 minute period making sure nothing is missed.

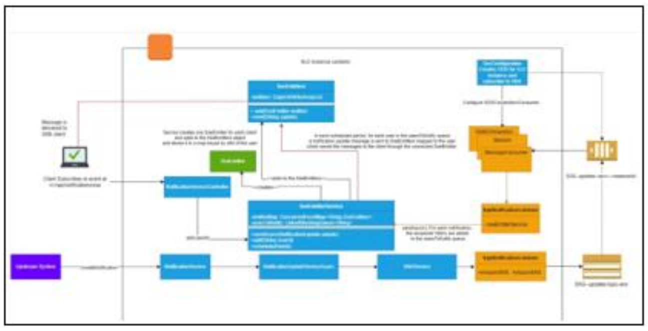
Figure 1: SSE Implementation architecture

4.4 Practical Considerations

The default EventSource object doesn’t support sending headers as part of the request. We had a need to authenticate the user before establishing a connection. For this we used a polyfill. It supports sending headers and cross domain requests. To keep the connection alive a comment message should be sent every 30 seconds which will be used to detect disconnects.

4.4.1 Idle timeouts

Our application uses a load balancer and since it is possible that a user might not get any updates, the load balancer can kill the idle connections. To overcome this we instituted a comment message that will be sent every 30 seconds to reset the idle time and keep the connection alive. By sending the heartbeat time just a few seconds less than the idle timeout, we can keep the connection alive.

4.4.2 Authorization

Since we want to enable authentication for the SSE connection, we use the polyfill to send the authorization header. With the polyfill the authorization header can be sent like this

```
var es = new EventSourcePolyfill('/events', { headers: {
'Authorization': 'Bearer token'
}
});
```

4.4.3 Handling multiple connections

This application is de- ployed with multiple instances for high-availability. So as users connect to the application from different places, it is possible that each user has multiple connections open. To avoid multiple con- nections from a single user, we have set up a check to not create a connection if a user already has 4 connections open.

4.4.4 Handling disconnected clients

Once the SSE connection is established, user will receive all the updates through the con- nection. For some reason if the client disconnects by closing the browser or they loose network connection, the connection becomes stale. Server can discover this when sending a message or a heartbeat.

When this happens the SseEmitter instance can be removed from the state by using the onError or onTimeout methods exposed by the ResponseBodyEmitter interface.

5. Benefits and Results

Server sent events helped scale the system and improved the user experience. After introducing SSE, the users are getting notifications faster (5s vs 30s) without adding any new system resources thereby improving the user experience. Here are the connection count, CPU and memory usage graphs with SSE implemented for 4000 users which is well within 60% range. This gave us the confidence to release notification system enterprise wide.

This is the active connection count with 4000 users connected through the load balancer to the server using SSE

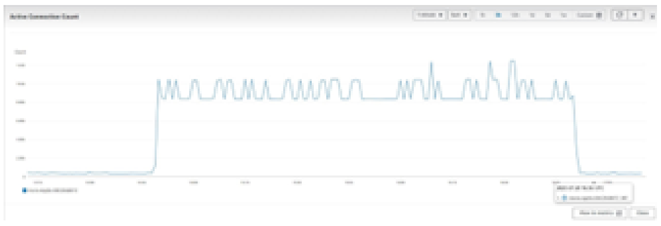

Figure 2: Active Connection Count
30 Nov 2020).

Spring Boot. URL: https://spring.io/projects/spring-boot.

Below is the memory usage with 4000 users where it peaks at below 25%

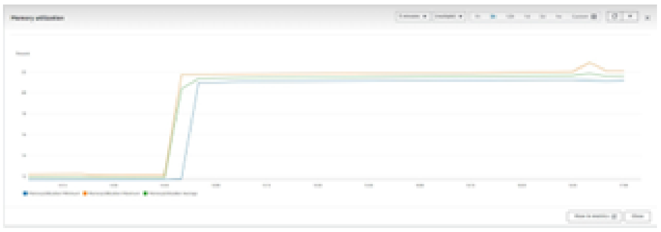

Figure 3: Memory Usage

Below is the CPU usage with 4000 users where it spikes momentarily to 10% and comes back to about 2%

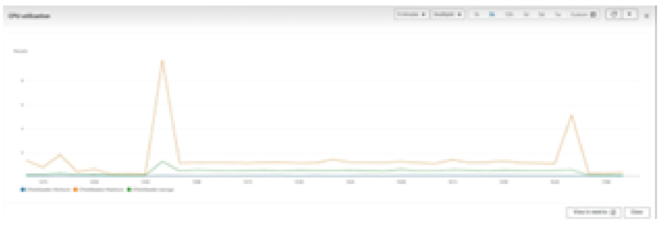

Figure 4: CPU Usage

We could also do it without introducing any new infrastructure or dependencies on the server side as everything was already available in Spring Boot with a familiar HTTP protocol.

6. Conclusion

This paper reviewed the need for real-time updates from servers to clients and the various techniques available to do that from HTTP polling, long polling, Server Sent Events to Websockets. It identified the motivation for these technologies and also the use-cases best suited for them. This paper also goes in detail about an implementation of server sent events for real-time notifications for an enterprise-wide notification system. It gives a detailed overview of server sent events and about the technologies and libraries used in the implementation. It also goes into detail of the practical consideration in making the server sent event system production ready. It then talks about the results of the implementation along with CPU, memory usage and active connection metrics. It makes the case that for unidirectional updates from server to the client, the complexity of websockets is not required and server sent events is a simpler alternative in that case.

References

[1] Paul Murley Zane Ma Joshua Mason Michael Bailey Amin Kharraz. "Web-socket adoption and the lanscape of realtime web". In: *WWW 21: Proceedings of the Web Conference* Apr 2021 (2021), pp. 1192–1203. DOI: https://doi.org/10.1145/3442381.3450063.

[2] *ResponseBodyEmitter*. URL: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/ ResponseBodyEmitter.html.

[3] *Server Sent Events*. URL: https://javascript.info/server-sent-events. (Last Updated *Spring Framework*. URL: https://spring.io/projects/spring-framework.

[4] *SseEmitter*. URL: https://docs.spring.io/spring-

framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/SseEmitter.html.

[5] *WebSockets Living Standard*. URL: https : / / websockets . spec . whatwg . org / #network-intro.