Server-Sent Events with Spring

Published: March 05, 2017 • Updated: February 16, 2018 • java, spring

A popular choice for sending real time data from the server to a web application is WebSocket. WebSocket opens a bidirectional connection between client and server. Both parties can send and receive messages. In scenarios where the application only needs one way communication from the server to the client a simpler alternative exists: Server-Sent Events (SSE). It's a HTML5 standard and utilizes HTTP as transport protocol, the protocol only supports text messages and it's unidirectional, only the server can send messages to the client.

Support

Server-Sent Events are supported in most modern browsers. Only Microsoft's browsers IE and Edge do not have a built in implementation. Fortunately this is not a problem because Server-Sent Events uses common HTTP connections and can therefore implemented with a polyfill. The following polyfill libraries add Server-Sent Event support to IE and Edge.

- https://github.com/remy/polyfills/blob/master/EventSource.js by Remy Sharp
- https://github.com/rwaldron/jquery.eventsource by Rick Waldron
- https://github.com/amvtek/EventSource by AmvTek
- https://github.com/Yaffle/EventSource by Yaffle

We use the Yaffle/EventSource library in our productive applications and it works so far very reliable.

With SSE the server cannot immediately start sending messages to the client. It's always the client (browser) that establishes the connection and after that the server is able to send messages. A SSE connection is basically a long streaming HTTP connection.

1. Client opens the HTTP connection

2. Server sends zero, one or more messages over this connection.
3. Connection is closed by the server or due to a network error.
4. Client opens a new HTTP connection and so on...

The nice thing about Server-Sent Events is that they have a built in reconnection feature, when the client loses the connection he tries to reconnect to the server automatically. WebSocket does not have such a built in functionality.

Despite the fact that only the server can send messages to the client over SSE, you could develop applications like a chat application with this technology. Because a client application can always open a second HTTP connection with the Fetch API or XMLHttpRequest and send data to the server.

Client API

The Server-Sent Events API in the browser is simple and consist of only one object: `EventSource`
To open a connection an application needs to instantiate the object and provide the URL of the server endpoint.

```
const eventSource = new EventSource('http://localhost:8080/stream');
```

The browser immediately sends a GET request to the URL with the accept header `text/event-stream`

```
GET /stream HTTP/1.1
Accept: text/event-stream
...
```

Because this is a normal GET request, an application could add query parameters to the URL like in any other GET request.

```
const eventSource = new EventSource('http://localhost:8080/stream?event=news');
```

Query parameters cannot be changed during the lifecycle of the EventSource object. Every time the client reconnects he uses the same URL. But an application can always close the connection with the `close()` function and instantiate a new `EventSource` object.

```
const eventSource = new EventSource('http://localhost:8080/stream?event=worldnew
...
eventSource.close();
const eventSource = new EventSource('http://localhost:8080/stream?event=sports')
```

The HTTP response to a EventSource GET request must contain the Content-Type header with the value `text/event-stream` and the response must be encoded with UTF-8.

```
HTTP/1.1 200
Content-Type: text/event-stream;charset=UTF-8
```

The protocol that SSE uses is text based, starts with a keyword then a colon (:) and a string value. The `data` keyword specifies a message for the client. Spaces before and after the message will be ignored. Every line is separated with a carriage return (0d) or a line feed (0a) or both (0d 0a).

```
data: the server message
```

The server can split a message over several lines

```
data:line1
data:line2
data:line3
```

The browser will concatenate these three lines and emit one event. To separate message from each other the server needs to send a blank line after each message.

```
data:{"heap":148713928,"nonHeap":49888752,"ts":1488640735925}
```

```
data:{"heap":149344096,"nonHeap":49889392,"ts":1488640736927}

data:{"heap":149344096,"nonHeap":49889392,"ts":1488640736929}
```

You see that the payload does not have to be a simple string. It's perfectly legal to send JSON strings and parse them on the client with `JSON.parse` .

To process these events in the browser an application needs to register a listener for the `message` event. The property `data` of the event object contains the message. The browser filters out the keyword `data` and the colon and only assigns the string after the colon to `event.data` .

```
eventSource.onmessage = event => {
  const msg = JSON.parse(event.data);
  //access msg.ts, msg.heap, msg.nonHeap
};
```

An application can listen for the `open` and `error` event. The `open` event is emitted as soon as the server sends a 200 response back. The `error` event is called whenever a network error occurs. It is also emitted when the server closes the connection.

```
eventSource.onopen = e => console.log('open');
eventSource.onerror = e => {
  if (e.readyState == EventSource.CLOSED) {
   console.log('close');
  }
  else {
    console.log(e);
  }
};
```

Named Events

A server can assign an event name to a message with the `event:` keyword. The `event:` line can precede or follow the `data:` line. In this example the server sends 4 messages. The first message is an `add` event, the second a `remove` event, then follows an `add` event and the last message is an unnamed event.

```
event:add
data:100

data:56
event:remove

event:add
data:101

data:simple event
```

Named events are processed differently on the client. They do not trigger the `message` handlers. Named events emit an event that has the same name as the event itself. For this example we need 3 listeners to process all the messages. You cannot use the `on...` syntax for registering listeners to these events, they have to be registered with the `addEventListener` function.

```
eventSource.onmessage = e => {
  // receives: 'simple event'
};
/* OR
eventSource.addEventListener('message', function(e) {
  // receives: 'simple event'
}, false);
*/

source.addEventListener('add', function(e) {
  //receives "100" and "101"
}, false);

source.addEventListener('remove', function(e) {
  //receives "56"
}, false);
```

Reconnect

Browsers keep the Server-Sent Events HTTP connection open as long as possible. When the connection is closed by the server or due to a network error, the browser waits by default 3 seconds and then opens a new HTTP connection. The browser tries to send reconnect requests forever until he gets a 200 HTTP response back. With a call to `close()` an application can stop this.

The 3 seconds wait time between connections can be changed by the server. To change it the server sends a `retry:` line together with the message. The number after the colon specifies the number of milliseconds the browser has to wait before he tries to reconnect.

```
event:add
data:100
retry:10000
```

After the browser received this message he changes the wait time between connections to 10 seconds. With `retry:0` the browser immediately tries to reconnect after the previous connection was closed.

ID

The server can assign an id to every message with the `id:` keyword. Valid values for the id are any arbitrary string.

```
id:2012-08-19T10:11:20
data:648
```

A client can access this id with the property `lastEventId` of the event object.

```
source.addEventListener('message', function(e) {
        console.log(e.data); // "648"
        console.log(e.lastEventId); // "2012-08-19T10:11:20"
}, false);
```

The primary use case whey this id exists is to keep track what messages the client successfully received. When the SSE connection was closed the browser sends a new GET request and in this request he sends the last received message id as an additional HTTP header `Last-Event-ID` to the server.

```
GET /memory HTTP/1.1
Accept: text/event-stream
Last-Event-ID: 2012-08-19T10:11:20
```

The server can then read this header and send all new created messages since this id to the client, to make sure that the client receives all messages without any gap.

SSE support in Spring

Spring introduced support for Server-Sent Events with version 4.2 (Spring Boot 1.3). In the following example we create a Spring Boot application that sends the current used heap and non-heap memory of the Java virtual machine as Server-Sent Events to the client. The client is a simple html page that displays these values.

We create the server application with the http://start.spring.io/ website and select `Web` as the only dependency.

Next we create a Pojo that holds the memory information

```
public class MemoryInfo {
  private final long heap;
  private final long nonHeap;
  private final long ts;
  //get and set methods
```

src/main/java/ch/rasc/sse/MemoryInfo.java

Then we create a scheduled service that reads the memory information every second, creates an instance of the `MemoryInfo` class and publishes it with Spring's event bus infrastructure

```
@Service
public class MemoryObserverJob {

  public final ApplicationEventPublisher eventPublisher;

  public MemoryObserverJob(ApplicationEventPublisher eventPublisher) {
    this.eventPublisher = eventPublisher;
  }

  @Scheduled(fixedRate = 1000)
  public void doSomething() {
    MemoryMXBean memBean = ManagementFactory.getMemoryMXBean();
    MemoryUsage heap = memBean.getHeapMemoryUsage();
    MemoryUsage nonHeap = memBean.getNonHeapMemoryUsage();

    MemoryInfo mi = new MemoryInfo(heap.getUsed(), nonHeap.getUsed());
    this.eventPublisher.publishEvent(mi);
  }
}
```

src/main/java/ch/rasc/sse/MemoryObserverJob.java

Next we create a RestController that handles the EventSource GET request from the client. The get handler needs to return an instance of the class `SseEmitter` . Each client connection is represented with it's own instance of `SseEmitter` . Spring does not give you tools to manage these `SseEmitter` instances. In this application we store the emitters in a simple list(`emitters`) and add handlers to the emitter's completion and timeout event to remove them from the list.

```
@Controller
public class SSEController {

  private final CopyOnWriteArrayList<SseEmitter> emitters = new CopyOnWriteArra

  @GetMapping("/memory")
  public SseEmitter handle() {
      SseEmitter emitter = new SseEmitter();
      this.emitters.add(emitter);

      emitter.onCompletion(() -> this.emitters.remove(emitter));
      emitter.onTimeout(() -> this.emitters.remove(emitter));

      return emitter;
    }

    @EventListener
    public void onMemoryInfo(MemoryInfo memoryInfo) {
      List<SseEmitter> deadEmitters = new ArrayList<>();
      this.emitters.forEach(emitter -> {
        try {
          emitter.send(memoryInfo);
        }
        catch (Exception e) {
          deadEmitters.add(emitter);
        }
      });

      this.emitters.removeAll(deadEmitters);
    }
  }
```

src/main/java/ch/rasc/sse/SSEController.java

By default, Spring Boot with the embedded Tomcat server keeps the SSE HTTP connection open for 30 seconds. An application can change that with an entry to the application.properties file

```
spring.mvc.async.request-timeout=45000
```

This setting keeps the HTTP connection open for 45 seconds. Alternatively an application can set the timeout directly on the `SseEmitter` constructor.

```
SseEmitter emitter = new SseEmitter(180_000L); //keep connection open for 180 s
```

The method `onMemoryInfo` is annotated with the `@EventListener` annotation and listens for the events that are sent from the `MemoryObserverJob` class. When a new object is emitted the method loops over all registered clients and tries to send the MemoryInfo to the clients. The send call can always fail when the client is no longer connected. Servers do not get informed when the EventSource connection is closed either normally with `close()` or due to a network error or the user just closed the browser. Because of that we add a try catch around the send method and when the send fails the application removes that client from the emitters list.

To send messages to the client the application calls the emitter's `send` method. This method expects either an object or a `SseEventBuilder` . Objects will be converted to a JSON string and sent in a `data:` line to the client. A send call with the `SseEventBuilder` allows the application to set all the previous mentioned message attributes like retry, id and event name. The static method `event()` of the `SseEmitter` class creates a new `SseEventBuilder` .

```
SseEventBuilder builder = SseEmitter.event()
                                    .data(memoryInfo)
                                    .id("1")
                                    .name("eventName")
                                    .reconnectTime(10_000L);
emitter.send(builder);
```

Every method of the builder corresponds to a keyword in the SSE message:

```
data(...) -> data:
id(...) -> id:
name(...) -> event:
reconnectTime(...) -> retry:
```

Spring provides an easy way to access the `Last-Event-ID` HTTP header when the application needs it. You have to make the parameter optional with `required=false` because the initial GET request from the client does not contain this HTTP header.

```
@GetMapping("/memory")
public SseEmitter handle(@RequestHeader(name = "Last-Event-ID", required = fals
    ...
}
```

The client opens the SSE connection with

```
const eventSource = new EventSource('http://localhost:8080/memory');
```

and registers a `message` listener that parses the JSON and sets the innerHTML of three dom elements to display the received data.

```
<!DOCTYPE html>
<html>
<head>
<title>Server Memory Monitor</title>
<script>
function initialize() {
  const eventSource = new EventSource('memory');

  eventSource.onmessage = e => {
    const msg = JSON.parse(e.data);
    document.getElementById("timestamp").innerHTML = new Date(msg.ts);
    document.getElementById("heap").innerHTML = msg.heap;
    document.getElementById("nonheap").innerHTML = msg.nonHeap;
  };
}

window.onload = initialize;
</script>
</head>
<body>
  <h1>Memory Observer</h1>
  <h3>Timestamp</h3>
  <div id="timestamp"></div>

  <h3>Heap Memory Usage</h3>
  <div id="heap"></div>
  <h3>Non Heap Memory Usage</h3>
  <div id="nonheap"></div>
</body>
</html>
```

```
</html>
```

src/main/resources/static/index.html

You find the source code for the entire project on GitHub.

More information

The following articles provide you with more information about Server-Sent Events.

- https://hpbn.co/server-sent-events-sse/
- https://www.html5rocks.com/en/tutorials/eventsource/basics/
- https://developer.mozilla.org/en-US/docs/Web/API/EventSource
- https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

Keeping track

At the end a shameless self plug. Because Spring does not provide support for keeping track of the `SseEmitter` instances I wrote a library that does that for Spring Boot applications. You can add the library with this dependency to a project.

```
<dependency>
  <groupId>ch.rasc</groupId>
  <artifactId>sse-eventbus</artifactId>
  <version>1.1.7</version>
</dependency>
```

To enable the library you need to add the `@EnableSseEventBus` annotation to an arbitrary `@Configuration` class.

```
@SpringBootApplication
@EnableSseEventBus
public class Application {
    ...
}
```

The library creates a bean of type `SseEventBus` that an application can inject into any Spring managed bean.

```
@Controller
public class SseController {
  private final SseEventBus eventBus;
  public SseController(SseEventBus eventBus) {
    this.eventBus = eventBus;
  }

  @GetMapping("/register/{id}")
  public SseEmitter register(@PathVariable("id") String id) {
    return this.eventBus.createSseEmitter(id, SseEvent.DEFAULT_EVENT)
  }
}
```

The library expects that each client sends a unique id. An application can create such an id for example with a uuid library like https://github.com/kelektiv/node-uuid. To register the client the program calls the `createSseEmitter` method with the id and the name of the events the client wants to listen to.

```
const uuid = uuid();
const eventSource = new EventSource(`/register/${uuid}`);
eventSource.addEventListener('message', response => {
    //handle the response from the server
    //response.data contains the data line
}, false);
```

To publish messages an application can either call the `handleEvent` method on the `SseEventBus` bean or publishes a `SseEvent` object with Spring's event infrastructure

```
@Service
public class DataEmitterService {
  private final SseEventBus eventBus;
  public DataEmitterService(SseEventBus eventBus) {
    this.eventBus = eventBus;
    this.eventBus = eventBus;
  }

  public void broadcastEvent() {
    this.eventBus.handleEvent(SseEvent.ofData("some useful data"));
  }
}


@Service
public class DataEmitterService {
  private final ApplicationEventPublisher eventPublisher;
  public DataEmitterService(ApplicationEventPublisher eventPublisher) {
    this.eventPublisher = eventPublisher;
  }

  public void broadcastEvent() {
    this.eventPublisher.publishEvent(SseEvent.ofData("some useful data"));
  }
}
```

Visit the GitHub project page for more information: https://github.com/ralscha/sse-eventbus
You find a demo application that uses the library in this repository:
https://github.com/ralscha/sse-eventbus-demo

Home | RSS | Atom | Send Feedback