Spring Server Sent Events

And

Websockets

**What are Server Push Messages?**

A server push message (also known as a push notification) is the delivery of information to a computing device from an application server where the request for the transaction is initiated by the server rather than by an explicit request from the client.

**Session Agenda**

Introduction to Server Sent Events

How Server Sent Events Work

Server Sent Events in Spring

ResponseBodyEmitter and SseEmitter

Introduction to Spring Websockets

Websocket configuration to target clients

**What are Server Push Messages?**

A server push message (also known as a push notification) is the delivery of information to a computing device from an application server where the request for the transaction is initiated by the server rather than by an explicit request from the client.

**Need for Server Push Messages**

Server push messages are designed to implement real-time communication between the web application and application server, and to send real-time data and updates from the application server to your web application.

For example, Gmail instantaneously adds incoming mail, Facebook shows chat messages, etc.

Such applications subscribe to certain topics and keeps open channel for server updates.

**Implementations of Push Technology**

Webpush, Pushlet , Long polling , Flash XMLSocket relays, Reliable Group Data Delivery (RGDD), Push notification.

HTTP server push -- Also known as HTTP streaming is used for sending asysnchronous data from server to client. WebSocket API, MIME's multipart/x-mixed-replace, Server-sent Events are mechanisms that use this protocol.

As mentioned, there are many ways a server can communicate with clients, but Server-sent Events and Websockets are most popular communication standards.

**Server Sent Events**

SSE is a mechanism that allows the server to asynchronously push the data to the client once the client-server connection is established.

The server can then decide to send data whenever a new stream of data is available.

It allows a web application to handle a unidirectional event stream and receive updates whenever server emits data.

**How Server Sent Events Work**

Client requests a HTTP connection establishment to the server over a particular URL. If the server allows, connection gets established.

URL is basically a topic address, which is a category of messages. Server sends different types of events under different topics eg. news updates, stock exchange etc.

When the server has some updates, the server pushes the messages as events under a topic.

All the clients which have open connection upon that particular topic, receive the updates.

Server can close the connection at any time or it can be closed because of some network error or any exception at the server-side.

If a connection is closed due to server or network error, the client tries automatically to re-establish the connection.

**Server Sent Events in Spring**

The SseEmitter class can deliver events from the server to the client. The messages have a Content-Type header of text/event-stream.

SseEmitter is a specialization of ResponseBodyEmitter for sending Server-Sent Events.

**ResponseBodyEmitter**

A ResponseBodyEmitter return type can send multiple objects, where each object is written with a compatible HttpMessageConverter.

It is a controller method return value type for asynchronous request processing where one or more objects are written to the response.

**ResponseBodyEmitter Example**

@RestController

@RequestMapping("/rbe")

public class RbeController {

private final ExecutorService executor

= Executors.newSingleThreadExecutor();

@GetMapping("/response")

public ResponseEntity<ResponseBodyEmitter> getRbeResponse() {

ResponseBodyEmitter emitter = new ResponseBodyEmitter();

executor.execute(() -> {

try {

for (int i = 1; i <= 5; i++) {

String message = " Message object. ";

randomDelay();

emitter.send(i + message);

}

emitter.complete();

} catch (Exception ex) {

emitter.completeWithError(ex);

}

});

return new ResponseEntity<ResponseBodyEmitter>(emitter, HttpStatus.OK);

}

private void randomDelay() {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

**Events in Spring**

The events are pretty simple and have only four fields.

id - The ID of the event

event - the type of event

data - The event data

retry - Reconnection time for the event stream

To add this info to the event, we use the SseEventBuilder . The event() factory-method of the SseEmitter creates an instance.

SseEventBuilder eventBuilder = SseEmitter.event();

emitter.send(

eventBuilder

.data(dataObject)

.name("dataObject-created")

.id(String.valueOf(dataObject.hashCode()))

);

**SseEmitter Example**

@RestController

@RequestMapping("/sse")

public class SseController {

private final ExecutorService executor

= Executors.newSingleThreadExecutor();

@GetMapping("/response")

public ResponseEntity<SseEmitter> getSseResponse() {

SseEmitter emitter = new SseEmitter();

executor.execute(() -> {

try {

for (int i = 1; i <= 5; i++) {

randomDelay();

emitter.send("/sse" + " @ " + new Date()

, MediaType.APPLICATION\_JSON);

}

emitter.complete();

} catch (Exception ex) {

emitter.completeWithError(ex);

}

});

return new ResponseEntity(emitter, HttpStatus.OK);

}

private void randomDelay() {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

**SseEmitter With Multiple Clients**

Keep a track of all client connections by having a list of all SseEmitter objects.

For every client, when a new connection is established return a new object of SseEmitter type. Before returning from the method, add its reference to a list.

When the server has an update to send as an event, then process the list and send the event to every emitter object for every client.

If any SseEmitter object fails to send an event, that means the client connection is closed. So remove that SseEmitter object from the list.

**Create a service class which will hold a list of SseEmitter objects.**

private final CopyOnWriteArrayList<SseEmitter> emitters = new CopyOnWriteArrayList<>();

**Create a method to register new clients.**

public SseEmitter registerClient(){

SseEmitter emitter = new SseEmitter();

emitters.add(emitter);

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

emitter.onTimeout(() -> {

emitter.complete();

emitters.remove(emitter);

});

return emitter;

}

**Create a POJO class, for sending as event to clients.**

@Data

@Builder

public class Notification {

private String user;

private String message;

@Builder.Default

private LocalDateTime localDateTime = LocalDateTime.now();

}

Processing and sending event logic.

public void process(String message, String user) throws IOException {

Notification notification = Notification.builder()

.user(StringUtils.isBlank(user) ? "Guest" : user)

.message(message)

.build();

sendEventToClients(notification);

}

public void sendEventToClients(Notification notification) {

List<SseEmitter> deadEmitters = new ArrayList<>();

emitters.forEach(emitter -> {

try {

emitter.send(notification);

} catch (Exception e) {

deadEmitters.add(emitter);

}

});

emitters.remove(deadEmitters);

}

Create a scheduled task to generate and sent events in every 5 seconds.

@Scheduled(cron = "\*/5 \* \* ? \* \*")

public void scheduleTask() throws IOException {

sseService.process("Scheduled job run", "Job");

}

Create end point that will trigger events to all clients.

@GetMapping("/message")

public @ResponseBody void sendMessages(@RequestParam String message,

@RequestParam(required = false) String user) throws IOException {

sseService.process(message, user);

}

Create end point for clients.

@GetMapping("/receive")

public @ResponseBody

SseEmitter getEmitter() {

return sseService.registerClient();

}

**Create a Thymeleaf file for creating EventSource object at client.**

<!doctype html>

<html>

<body>

<script src="/webjars/jquery/3.5.1/jquery.min.js"></script>

<div>

<h3> SSE events - Client</h3>

<div id="events"></div>

</div>

<script type="application/javascript">

var subscribeEvents = function () {

var eventSource = new EventSource('/sse/receive');

eventSource.onmessage = function (e) {

var notification = JSON.parse(e.data);

var d = new Date(notification.localDateTime);

var html = "<span><b>New event</b> on <i>" + d.toLocaleDateString() + " " + d.toLocaleTimeString() + "</i>: "

+ "[" + notification.user + "'s Action] " + notification.message + "</span>";

document.getElementById("events").innerHTML = html + "<br/>" + document.getElementById("events").innerHTML;

};

}

window.onload = subscribeEvents;

window.onbeforeunload = function () {

eventSource.close();

}

</script>

</body>

</html>

**Finally create an endpoint to return thymeleaf view.**

@GetMapping("/client")

public ModelAndView getClient(ModelAndView modelAndView) {

modelAndView.setViewName("sseClient.html");

return modelAndView;

}