

Tiempos largos en peticiones

En este documento se abordará como implementar soluciones a los tiempos largos en peticiones al servidor.

Tiempos largos en peticiones al servidor (Spring-boot).

Asynchronous



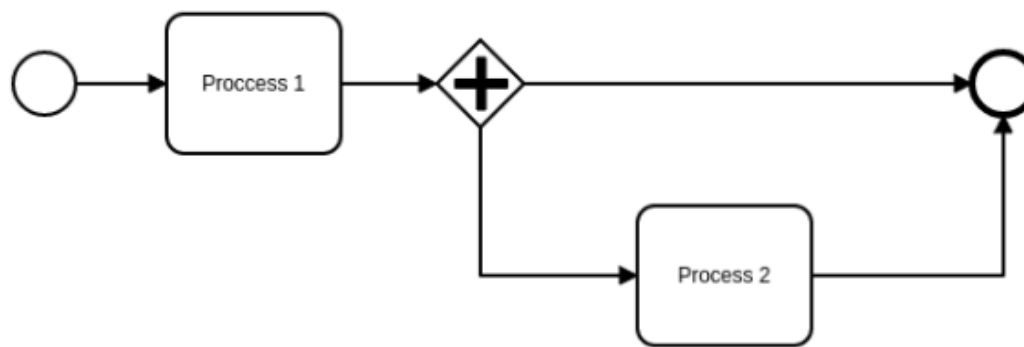
Asincronía hace referencia al suceso que no tiene lugar en total correspondencia temporal con otro suceso, En programación esto se maneja con hilos, comúnmente utilizados para aumentar la eficiencia en la programación y en esencia lo que nos permite la asincronía es ejecutar diferentes secciones de código simultáneamente donde una parte no depende de la otra para evitar fallos de dependencia.

Implementar esta tecnología de peticiones asincrónicas al servidor requiere de una lógica de negocio que permita notificar al usuario si el proceso terminó o no exitosamente, esto se debe proponer en la historia de usuario, algún ejemplo podría ser el envío de un correo electrónico o el manejo de campos en bases de datos que indiquen el estado del proceso.

Para implementar esta tecnología lo primero que se debe hacer es habilitarla en la clase principal del proyecto, a través de la anotación `@EnableAsync`, como se muestra a continuación.

```
1 | @EnableAsync
2 | publi class AdministracionPersonalApplication extends SpringBootServletInitial:
```

Una vez habilitada, podremos anotar cualquier método tipo `void` con la anotación `@Async`, esto lo que hará internamente es que abrirá otro hilo donde se ejecutará este proceso.



La anotación `@Async` se debe poner justo antes de la declaración del método.

```

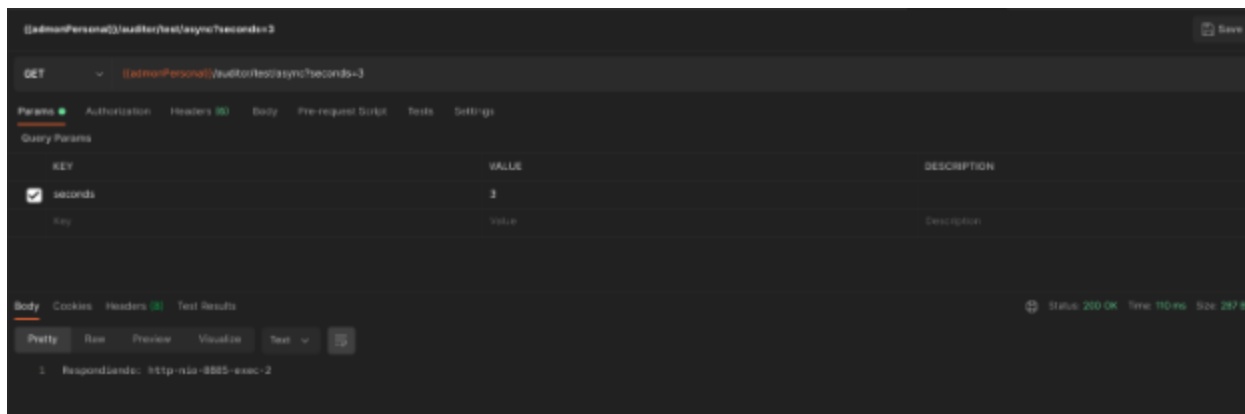
1  /**
2   * Llama a un procedimiento que dura los segundos que se le pasen,
3   * usando la anotación @Async
4   *
5   * @param seconds -
6   */
7  @Async
8  public void testAsync(Long seconds) {
9      this.fRespondeTSegundosDespues(seconds);
10     System.out.println("Respondiendo: " + Thread.currentThread().getName());
11 }
  
```

Esto hace que cuando se llame el método dentro del *Controller* libere la petición, pero internamente el servidor en el hilo que fue creado sigue corriendo el proceso.

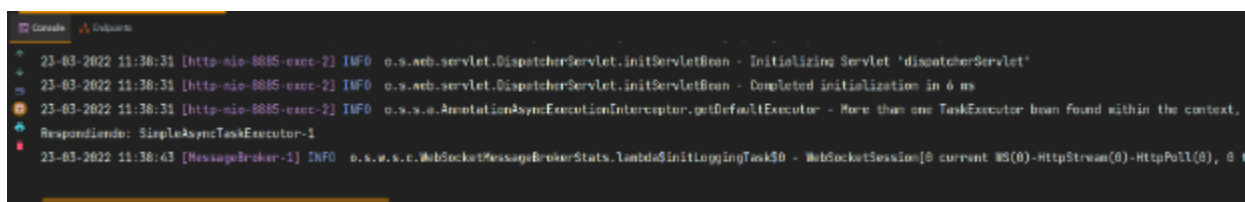
```

1  @GetMapping("/test/async")
2  public ResponseEntity<String> fRespondeTSegundosDespues(@RequestParam Long seconds) {
3      pfRecHumano.testAsync(seconds);
4      return new ResponseEntity<>("Respondiendo: " + Thread.currentThread().getName());
5  }
  
```

En la prueba mostrada anteriormente se puede evidenciar la diferencia del hilo principal y el hilo que se crea a partir de la llamada al servicio asíncrono.



Pasamos como parámetro 3 segundos para simular un tiempo de respuesta, y vemos en postman que se libera la petición de manera instantánea e internamente en el servidor se completa el procedimiento después de 3 segundos.



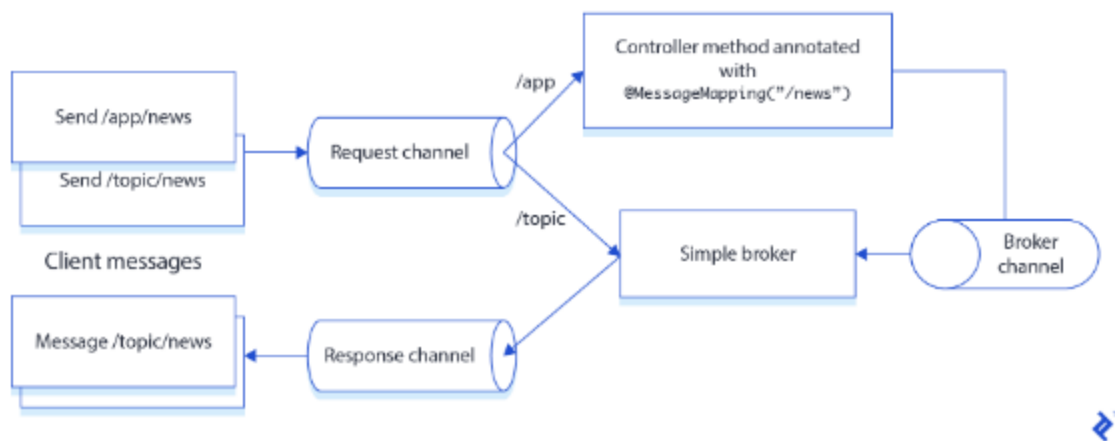
Conclusiones

La idea de esta forma de llamar procedimientos almacenados es manejar una lógica interna para notificar cuando el procedimiento termine de ejecutarse, ya sea con una bandera a nivel de back o base de datos, o con el método de envío de correos, según lo que más convenga de acuerdo al problema que se quiera abordar dando libertad al desarrollador a abordarlo como le sea conveniente.

WebSockets

Para implementar webSockets en *spring-boot* una de las opciones que tenemos es con la metodología **STOMP**, (Streaming Text Oriented Messaging Protocol) es un protocolo de transferencia de mensajes de texto simple, solo menciona algunas formas específicas de cómo se intercambian los marcos de mensajes entre el cliente y el servidor mediante WebSockets.

Para entender el funcionamiento nos podemos apoyar en el siguiente grafico tomado de un post en internet.



Cada solicitud es enviada a una dirección ya sea con prefijo `/app` o *prefijo /topic*, en este caso únicamente se establecen 2 rutas, dichas peticiones son recibidas y trabajadas como lo que son, en `/app` nos encontramos a un controlador, es decir un método con lógica de negocio y anotado con `@MessageMapping` y el nombre del método, y también tenemos otro que es `/topic` que es el prefijo de destino, en caso de que nosotros queramos tener un canal de comunicación estático pues simplemente este prefijo indica por donde el *broker* debe generar la respuesta, en casos privados como el de nuestro interés no se establece de manera predefinida dicha ruta sino que es generada dinámicamente en función del usuario.

Pero bueno a nivel de spring boot necesitamos ya sea construir una clase de configuración y anotarla con `@EnableWebSocketMessageBroker` e implementamos la interfaz `WebSocketMessageBrokerConfigurer` o como lo hicimos nosotros se agrega la anotación a la clase principal y se registra un *Bean* que implementa la interfaz y sobrescribe los métodos que necesitamos, para registrar la base del websocket y las rutas destino.

```

1  @Bean
2  public WebSocketMessageBrokerConfigurer socketConfigurer() {
3      return new WebSocketMessageBrokerConfigurer() {
4          @Override
5          public void registerStompEndpoints(StompEndpointRegistry registry) {
6              registry.addEndpoint("/ws")
7                  .setAllowedOrigins("*")
8                  .withSockJS();
9          }
10
11         @Override
12         public void configureMessageBroker(MessageBrokerRegistry registry) {
13             registry.setApplicationDestinationPrefixes("/app");
14             registry.enableSimpleBroker("/user");
15             registry.setUserDestinationPrefix("/user");
16         }
17     }
  
```

```
18     };  
19 }
```

En este caso no nos interesa por ahora un canal de ámbito general o público, por el contrario, queremos un canal dinámico que garantice seguridad y privacidad en función del usuario que accede al servicio de websocket.

Entonces cómo funciona el controlador asociado al prefijo */app* de aplicación, el que tendrá la lógica de negocio

```
1  @Controller  
2  public class GreetingController {  
3  
4      private SimpMessagingTemplate simpMessagingTemplate;  
5      private PFRecHumano pfRecHumano;  
6  
7      @MessageMapping("/private-message") //receptor de mensajes  
8      public void greetPrivate>HelloMessage message) {  
9          System.out.println(  
10             this.pfRecHumano.prcBuscaNombreCodErrorExito("1234"));  
11             simpMessagingTemplate.convertAndSendToUser(message.getIdentifier(), "/{  
12                 //return the payload  
13             }  
14  
15             @Autowired  
16             public void setSimpMessagingTemplate(SimpMessagingTemplate simpMessagingTer  
17                 this.simpMessagingTemplate = simpMessagingTemplate;  
18             }  
19  
20             @Autowired  
21             public void setPfRecHumano(PFRecHumano pfRecHumano) {  
22                 this.pfRecHumano = pfRecHumano;  
23             }  
24     }
```

Dicho controlador, no de tipo *RestController* pues al final no es una petición *Rest*; el cual tiene un método anotado con `@MessageMapping` y en este caso un nombre */private-message* para hacer alusión a que es un socket semi-privado por ahora.

Entonces este método se encarga de llamar a un procedimiento el cual tarda 10 segundos en ejecutarse y luego usando la clase *SimpMessagingTemplate* hacemos la magia de convertir y enviar dinámicamente en

función del usuario, y la ruta de salida, es decir a la que se debe suscribir el cliente (frontend) será del tipo `/prefijo/${identificadorUsuario}/destination`.

En este caso es: `/user/${nombreUsuario}/private` y ya veremos más a delante que esto cambia dinámicamente.

Veamos ahora como funciona a nivel de *front-end* básico.

```
1  function connect() {  
2      var socket = new SockJS(baseRoute+' /ws');  
3      stompClient = Stomp.over(socket);  
4      stompClient.connect({}, function (frame) {  
5          setConnected(true);  
6          console.log('Connected: ' + frame);  
7          stompClient.subscribe(`/user/${$("##identifier").val()}/private`, funct:  
8              showGreeting(greeting.body);  
9              ocultarBarra();  
10         });  
11     });  
12 }
```

Se crea un socket basado en *SockJs* el cual apunta a la ruba base registrada en el *bean* que mencionamos, de esta manera nuestro cliente ya puede interactuar con el socket del servidor y bajo la estructura *connect* le indicamos que se suscriba a `/user/identifier/private`, donde el identificador es capturado en tiempo desde un input.

Se ve de esta manera: En la consola vemos como se abre el socket en el cliente para manejar la conexión y se conecta al servidor, el servidor responde con *Connected*, luego de establecida la conexión vemos como se suscribe y se carga el destination como `:/user/Pelopincho/private` donde Pelopincho es nuestro nombre en clave para el ejemplo.

```

Live reload enabled.                                     (index):91
Opening Web Socket...                                   stomp.min.js:8
Web Socket Opened...                                   stomp.min.js:8
>>> CONNECT                                           stomp.min.js:8
accept-version:1.1,1.0
heart-beat:10000,10000

<<< CONNECTED                                         stomp.min.js:8
version:1.1
heart-beat:0,0

connected to server undefined                           stomp.min.js:8
Connected: CONNECTED                                  app.js:22
heart-beat:0,0
version:1.1

>>> SUBSCRIBE                                         stomp.min.js:8
id:sub-0
destination:/user/Pelopincho/private

>

```

De este modo podemos ahora solicitar una petición y ver su evolución en el tiempo.



Tan pronto como registramos un nombre y enviamos la llamada, él comunica al controlador /app/private-message y este ejecuta el procedimiento (como dura bastante tiempo por eso se simula una barra de progreso).

The screenshot shows a web application interface. At the top, there is a blue progress bar. Below it, there is a section labeled "WebSocket connection:" with two buttons: "Connect" and "Disconnect". To the right of these buttons, there is a label "identifier" and a text input field containing "Pelopincho". Below this, there is a label "What is your name?" and another text input field containing "Mario". Below the second input field, there is a "Send" button. At the bottom of the interface, there is a section labeled "Greetings" with a text area that is currently empty.

y al cabo de los 10 segundos obtenemos una respuesta:

The screenshot shows the same web application interface as before, but now the "Greetings" section has a response message: "hola mundo2022-03-23 11:02:26". The "Send" button is still visible below the "What is your name?" input field.

Ahora si lo hacemos en dos pestañas al tiempo con dos identificadores diferentes vemos como internamente

no se conocen entre ellos y la información recibida cambia:

WebSocket connection: Connect Disconnect

identifier
carlo

What is your name?
magnu

Send

Greetings

hola mundo2022-03-23 11:05:07

Como última prueba generaremos dos conexiones con el mismo identificador en diferentes pestañas del navegador para ver cómo se suscriben a la misma ruta y comparten comunicación.

WebSocket connection: Connect Disconnect

identifier
pelopincho

What is your name?
hey hey

Send

Greetings

hola mundo2022-03-23 10:53:15

Por esto es importante definir bien una *autorizacion*, lógicamente se debe garantizar que solo un usuario autorizado pueda suscribirse, con el uso de JWT, también que el identificador será en función del token con el *payload* el cual es único y esto garantiza que el usuario puede abandonar el portal y luego conectarse y ver si ya se resolvió su petición, además de garantizar la privacidad y seguridad de la información.

Prueba realizando una conexión y abandonando el portal, luego ingresando al cabo de un minuto y suscribirse para recibir la respuesta.

WebSocket connection: Connect Disconnect

identifier
pelopincho

What is your name?
Your name here...

Send

Greetings

hola mundo2022-03-23 11:12:28

Sin realizar ningún *send*, solo estando conectado el recibe la comunicación pues el backend ya tiene definido dicho canal para enviar información, esto solventa muchos inconvenientes...

#1 que pasa con el tiempo de vida del token? Sí el token expira durante la llamada, pero el cliente no abandona el portal es deber del front reactivar el token sin que el usuario se entere para que no expire la sesión sin embargo si el usuario se *desloguea* y luego al cabo de una hora quiere ver si el procedimiento ya acabó puede simplemente conectarse pues la ruta de mensajes depende únicamente del payload del token y el backend y JWT garantizan seguridad pues con la firma del token cualquier modificación en dicho payload dañará la

autorización.

Por seguridad será necesario establecer una lógica por si llega a conectarse posterior al tiempo de respuesta, tal vez una acción para ver el histórico de pronto con una tabla auxiliar (por definir).

Conclusión

Los websockets son una tecnología bastante utilizada actualmente para comunicación en tiempo real, la cual no es la sugerida para abordar los problemas mencionados, si bien es una posible solución y nos da una manera de abordarlo igualmente debe manejar una lógica de consulta o señal que garantiza la información al momento de realizar peticiones y de esta manera informar al usuario y facilitar el control en todo momento del procedimiento, pero la opción asíncrona es más sencilla y por ahora es más cómoda para ser trabajada.