Arquitecturas de Software

Stomp y Websockets

Clone el proyecto que stomp-websockets

Crear una clase de representación de un recurso

Una vez que se ha clonado el proyecto puede crear el servicio de mensajes STOMP.

Para comenzar, el servicio pretende aceptar mensajes STOMP que contienen un objeto JSON con el atributo 'messaje'. Por ejemplo:

```
{
    "message": "hola"
}
```

Para modelar el mensaje es necesario crear un objeto Java (POJO) con el atributo mensaje y el metodo get correspondiente:

```
package edu.eci.arsw.model;

public class ClientMessage {

        private String message;

        public String getMessage() {
                return message;
        }

        public void setMessage(String message) {
                this.message = message;
        }

}
```

Una vez que se recibe el mensaje, el servicio lo procesa creando un mensaje de respuesta y publicandolo en una cola donde se suscriben los clientes:

```
{
    "content": "Message: hola!"
}
```

Para modelarlo se crea otro objeto Java (POJO):

```
package edu.eci.arsw.model;

public class ServerMessage {

        private String content;

        public ServerMessage(String content) {
                this.content = content;
        }

        public String getContent() {
                return content;
        }

}
```

Spring utliza la libería Jackson JSON para convertir automaticamente las instancias de ServerMessage a JSON.

Controlador

Despues de es necesario crear un controlador para recibir los mensajes del cliente y procesar el mensaje de respuesta del servidor:

En Spring para procesar los mensajes STOMP, pueden ser enrutados a las clases anotadas con @Controller.

Por ejemplo el controlador MessageController procesa los mensajes dirigidos al destino "/message".

```
package edu.eci.arsw.controller;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;
import edu.eci.arsw.model.*;

@Controller
public class MessageController {

    @MessageMapping("/message")
    @SendTo("/topic/messages")
    public ServerMessage serverMessage(ClientMessage message) throws Exception {
```

```
            Thread.sleep(3000); // simulated delay
            return new ServerMessage("Message: " + message.getMessage() + "!");
    }

}
```

La anotación @MessageMapping("/message") se utiliza para invocar el método serverMessage() cuando se recibe un mensaje en la dirección "/message"

La conversión al objeto Java se realiza de manera automática. Para simular asincronía, se esperan 3 segundos para mandar un mensaje al cliente. La anotación @SendTo("/topic/messages") se utiliza para enviar el valor de retorno a todos los suscriptores de: "/topic/messages"

Configurando la mensajería STOMP

Cree la clase Java llamada WebSocketConfig:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.AbstractWebSocketMessageBrokerConfig
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws").withSockJS();
    }
}
```

La anotacion @Configuration se utiliza para indicar que es una clase de configuración de Spring. Las anotaciones @EnableWebSocketMessageBroker habilita la mensajeria para que sea publicada a todos los suscriptores.

Crear el cliente

Cree el archivo src/main/resources/static/index.htm

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello WebSocket</title>
    <script src="sockjs-0.3.4.js"></script>
    <script src="stomp.js"></script>
    <script src="stomp-ws.js"></script>
</head>
<body>
  <noscript><h2 style="color: #ff0000">Seems your browser doesn't support Javascript! Websoc
                Javascript and reload this page!</h2></noscript>
  <div>
    <div>
      <button id="connect">Connect</button>
      <button id="disconnect" disabled="disabled">Disconnect</button>
    </div>
    <div id="conversationDiv">
      <label>Message to send?</label><input type="text" id="message" />
      <button id="send">Send</button>
      <p id="response"></p>
    </div>
  </div>
</body>
</html>
```

y el archivo src/main/resources/static/stomp-ws.js

```
var stompClient = null;

function setConnected(connected) {
    document.getElementById('connect').disabled = connected;
    document.getElementById('disconnect').disabled = !connected;
    document.getElementById('conversationDiv').style.visibility = connected ? 'visible' : 'h
    document.getElementById('response').innerHTML = '';
}

function connect() {
    var socket = new SockJS('/ws');
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {
    setConnected(true);
```

```
    console.log('Connected: ' + frame);
    stompClient.subscribe('/topic/messages', function(serverMessage){
        showServerMessage(JSON.parse(serverMessage.body).content);
    });
    });
}

function disconnect() {
    if (stompClient != null) {
    stompClient.disconnect();
    }
    setConnected(false);
    console.log("Disconnected");
}

function sendMessage() {
    var message = document.getElementById('message').value;
    stompClient.send("/app/message", {}, JSON.stringify({ 'message': message }));
}

function showServerMessage(message) {
    var response = document.getElementById('response');
    var p = document.createElement('p');
    p.style.wordWrap = 'break-word';
    p.appendChild(document.createTextNode(message));
    response.appendChild(p);
}

function init() {
   var btnSend = document.getElementById('send');
   btnSend.onclick=sendMessage;
   var btnConnect = document.getElementById('connect');
   btnConnect.onclick=connect;
   var btnDisconnect = document.getElementById('disconnect');
   btnDisconnect.onclick=disconnect;
   disconnect();
}

window.onload = init;
```

Las partes principales del código son las funciones connect y sendMessage: * la función connect utiliza SockJS and stomp.js para abrir la conección a "/stomp-websocket/ws", una vez que la conección este establecida se suscribe a la cola /topic/messages * la función sendMessage, envía los mensajes que se colocan en el cuadro de texto a la dirección /app/hello.

Volver la aplicación ejecutable

Para poder ejecutar la aplicación es necesario crear una clase con el método main, en este caso src/main/java/edu/eci/arsw/Application.java.

```
package edu.eci.arsw;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

La anotación @SpringBootApplication se utiliza las clases anotadas con @Configuration para crear los contextos de la aplicación, @EnableAutoConfiguration para adicionar automaticamente los beans basados en el classpath y otras inicializaciones, @EnableWebMvc para crear una aplicación Spring MVC y habilita la aplicación como una aplicación web. Y la anotación @ComponentScan para buscar otras

Crear un ejecutable JAR

Para crear el ejecutable utilice

```
mvn clean package
```

y puede ejecutar la aplicación empaquetada en un jar:

```
java -jar target/stomp-websockets-0.1.0.jar
```

o utilizar el comando

```
mvn spring-boot:run
```

Verifique el servicio

Incluir otros mensajes

Se va a crear un servicio REST para incluir mensajes enviados a la aplicación mediante el metodo POST. Para lo cual es necesario crear la clase edu/eci/arsw/controller/MyRestController.java .

```
package edu.eci.arsw.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import edu.eci.arsw.model.ClientMessage;
import edu.eci.arsw.model.ServerMessage;

@RestController
@RequestMapping("/rest")
public class MyRestController {


    @Autowired
    private SimpMessagingTemplate template;

    @RequestMapping(value = "/msg",method = RequestMethod.POST)
    public ResponseEntity<?> addMessage(@RequestBody ClientMessage p) {
        template.convertAndSend("/topic/messages",new ServerMessage(p.getMessage()));
        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }

    @RequestMapping(value = "/check",method = RequestMethod.GET)
    public String check() {
        return "REST API OK";
    }

}
```

Para injectar el bean de mensajería de STOMP automáticamente se utiliza la anotación @Autowired.

Las anotación @RequestMapping habilita las urls “/rest/msg” y “/rest/check” en la aplicación y habilitan los métodos addMessage y check para que las procesen.

Los mensajes procesados por el método addMessage son de tipo POST y son convertidos de JSON automáticamente al objeto ClientMessage. El método addMessage utiliza template.convertAndSend para enviar a la cola "/topic/messages" un mensaje ServerMessage que contiene el mensaje enviado en el atributo 'mensaje' del mensaje del objeto ClientMessage.

Pruebe enviar mensajes utilizando el comando curl:

```
curl -H "Content-Type: application/json" -X POST -d '{ "message" : "hello people" }' http://
```