Laboratório WebSockets com Spring

Neste laboratório, criaremos um aplicativo da web simples que implementa mensagens usando os novos recursos WebSocket introduzidos com Spring Framework 4.0.

WebSockets é uma conexão bidirecional , full-duplex e persistente entre um navegador da web e um servidor. Uma vez que uma conexão WebSocket é estabelecida, a conexão permanece aberta até que o cliente ou servidor decida fechar esta conexão.

Um caso de uso típico pode ser quando um aplicativo envolve a comunicação de vários usuários entre si, como em um bate-papo. Construiremos um cliente de chat simples em nosso exemplo.

Dependências Maven

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-messaging</artifactId>
    <version>5.2.2.RELEASE</version>
</dependency>
```

Além disso, como usaremos *JSON* para construir o corpo de nossas mensagens, precisamos adicionar as dependências de *Jackson* . Isso permite que o Spring converta nosso objeto Java de / para *JSON* :

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.10.2</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version>
</dependency>
```

Ative o WebSocket no Spring

A primeira coisa a fazer é habilitar os recursos do WebSocket. Para fazer isso, precisamos adicionar uma configuração ao nosso aplicativo e anotar essa classe com *@EnableWebSocketMessageBroker* .

Como o próprio nome sugere, ele permite o manuseio de mensagens WebSocket, apoiado por um agente de mensagens:

```
@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("/chat");
         registry.addEndpoint("/chat").withSockJS();
    }
}
```

Aqui, podemos ver que o método *configureMessageBroker* é usado para **configurar o intermediário de mensagem** . Primeiro, habilitamos um agente de mensagens na memória para transportar as mensagens de volta para o cliente em destinos prefixados com "/ tópico".

W e completar a nossa simples configuração, designando o prefixo "/ aplicação" para destinos de filtro de segmentação métodos de aplicação anotado (via *@MessageMapping*).

Crie o model de mensagem

Para modelar a mensagem que transporta o texto, podemos criar um objeto Java simples com as propriedades *from* e *text* :

```
public class Message {

    private String from;
    private String text;

    // getters and setters
}
```

Crie um controlador de tratamento de mensagens

Como vimos, a abordagem do Spring para trabalhar com mensagens STOMP é associar um método do controlador ao endpoint configurado. Isso é possível por meio da anotação *@MessageMapping* .

A associação entre o endpoint e o controlador nos dá a capacidade de lidar com a mensagem, se necessário:

```
@MessageMapping("/chat")
@SendTo("/topic/messages")
public OutputMessage send(Message message) throws Exception {
    String time = new SimpleDateFormat("HH:mm").format(new Date());
    return new OutputMessage(message.getFrom(), message.getText(),
time);
}
```

Para fins de exemplo, vamos criar um outro objeto modelo chamado *OutputMessage* para representar a mensagem de saída enviada para o destino configurado . Preenchemos nosso objeto com o remetente e o texto da mensagem retirado da mensagem recebida e o enriquecemos com um carimbo de data / hora.

Depois de tratar nossa mensagem, nós a enviamos para o destino apropriado definido com a anotação *@SendTo* . Todos os assinantes do destino “ */ tópico / mensagens* ” receberão a mensagem.

Crie um cliente de navegador

Depois de fazer nossas configurações no lado do servidor, usaremos a **biblioteca sockjs-client** para construir uma página HTML simples que interage com nosso sistema de mensagens.

Em primeiro lugar, precisamos importar as bibliotecas cliente *sockjs* e *stomp* Javascript. A seguir, podemos criar uma função *connect ()* para abrir a comunicação com nosso endpoint, uma função *sendMessage ()* para enviar nossa mensagem STOMP e uma função *disconnect ()* para fechar a comunicação:

```
<html>
    <head>
        <title>Chat WebSocket</title>
        <script src="resources/js/sockjs-0.3.4.js"></script>
        <script src="resources/js/stomp.js"></script>
        <script type="text/javascript">
            var stompClient = null;

            function setConnected(connected) {
                document.getElementById('connect').disabled = connected;
                document.getElementById('disconnect').disabled =
!connected;

document.getElementById('conversationDiv').style.visibility
                  = connected ? 'visible' : 'hidden';
                document.getElementById('response').innerHTML = '';
            }

            function connect() {
                var socket = new SockJS('/chat');
                stompClient = Stomp.over(socket);
                stompClient.connect({}, function(frame) {
                    setConnected(true);
                    console.log('Connected: ' + frame);
                    stompClient.subscribe('/topic/messages',
function(messageOutput) {

showMessageOutput(JSON.parse(messageOutput.body));
                    });
                });
            }
```

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

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

            function showMessageOutput(messageOutput) {
                var response = document.getElementById('response');
                var p = document.createElement('p');
                p.style.wordWrap = 'break-word';
                p.appendChild(document.createTextNode(messageOutput.from
+ ": "
                  + messageOutput.text + " (" + messageOutput.time +
")"));
                response.appendChild(p);
            }
        </script>
    </head>
    <body onload="disconnect()">
        <div>
            <div>
                <input type="text" id="from" placeholder="Choose a
nickname"/>
            </div>
            <br />
            <div>
                <button id="connect"
onclick="connect();">Connect</button>
                <button id="disconnect" disabled="disabled"
onclick="disconnect();">
                    Disconnect
                </button>
            </div>
            <br />
            <div id="conversationDiv">
                <input type="text" id="text" placeholder="Write a
message..."/>
```

```
                <button id="sendMessage"
onclick="sendMessage();">Send</button>
                <p id="response"></p>
            </div>
        </div>

    </body>
</html>
```

Testando o Exemplo

Para testar nosso exemplo, podemos abrir algumas janelas do navegador e acessar a página de bate-papo em:

```
http://localhost:8080
```

Feito isso, podemos entrar no chat inserindo um apelido e clicando no botão de conexão. Se escrevermos e enviarmos uma mensagem, podemos vê-la em todas as sessões do navegador que aderiram ao chat.