Realtime Chat Application using Spring Boot

Introduction

This project is a realtime chat application developed using Spring Boot and WebSocket. The application allows users to join, chat, and leave chat rooms in real-time. Spring Boot provides a robust and scalable architecture for the application, while WebSocket enables real-time communication between the server and clients. The application has features such as joining chat rooms, sending messages, and leaving chat rooms, providing a seamless and interactive chatting experience for its users.

Websocket

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. WebSocket is distinct from HTTP.

The protocol enables interaction between a web browser (or other client application) and a web server with lower overhead than half-duplex alternatives such as HTTP polling, facilitating real-time data transfer from and to the server.

Once a websocket connection is established between a client and a server, both can exchange information until the connection is closed by any of the parties.

This is the main reason why WebSocket is preferred over the HTTP protocol when building a chat-like communication service that operates at high frequencies with low latency.

Stomp JS

Simple (or Streaming) Text Oriented Message Protocol (STOMP), formerly known as TTMP, is a simple text-based protocol, designed for working with message-oriented middleware (MOM). It provides an interoperable wire format that allows STOMP clients to talk with any message broker supporting the protocol.

Since websocket is just a communication protocol, it doesn't know how to send a message to a particular user. STOMP is basically a messaging protocol which is useful for these functionalities.

Sock JS

SockJS is used to enable fallback options for browsers that don't support WebSocket. The goal of SockJS is to let applications use a WebSocket API but fall back to non-WebSocket alternatives when necessary at runtime, i.e. without the need to change application code.

Under the hood, SockJS tries to use native WebSockets first. If that fails, it can use a variety of browser-specific transport protocols and presents them through WebSocket-like abstractions.

Codes

Websocket configuration

Path: config>WebSocketConfig.java

```
@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {


  @Override

  public void registerStompEndpoints(StompEndpointRegistry registry) {

    registry.addEndpoint("/websocket").withSockJS();

  }


  @Override

  public void configureMessageBroker(MessageBrokerRegistry registry) {

    registry.enableSimpleBroker("/topic");

    registry.setApplicationDestinationPrefixes("/app");

  }

}
```

This code is a Spring Framework configuration class that configures WebSocket communication for a chat application. It implements the WebSocketMessageBrokerConfigurer interface to customize the WebSocket configuration.

Dependencies

This code relies on the following Spring Framework imports:

- org.springframework.context.annotation.Configuration: Indicates that this class is a configuration class.
- org.springframework.messaging.simp.config.MessageBrokerRegistry: Allows configuration of the message broker for WebSocket communication.
- org.springframework.web.socket.config.annotation.*: Provides annotations and interfaces for configuring WebSocket endpoints.

Class Declaration

- @Configuration: Annotation indicating that this class is a configuration class.
- @EnableWebSocketMessageBroker: Annotation enabling WebSocket message handling with a message broker.

Methods

registerStompEndpoints

This method is overridden from the WebSocketMessageBrokerConfigurer interface. It configures the WebSocket endpoints that clients can connect to.

- StompEndpointRegistry: A registry for registering Stomp endpoints.

In this code, it registers the "/websocket" endpoint and enables SockJS fallback options.

configureMessageBroker

This method is overridden from the WebSocketMessageBrokerConfigurer interface. It configures the message broker used for routing messages between clients and the server.

- MessageBrokerRegistry: A registry for configuring the message broker.

In this code, it enables a simple in-memory message broker with the "/topic" destination prefix for broadcasting messages to subscribed clients. It also sets the application destination prefix to "/app" for routing messages that are sent from clients to server-side message handling methods.

This configuration class sets up the necessary configurations for WebSocket communication in the chat application. It enables the use of a message broker for handling message routing and sets up the endpoints that clients can connect to. The registerStompEndpoints method configures the endpoints, and the configureMessageBroker method configures the message broker.

Websocket Event Listener

Path: config>WebSocketEventListener.java

```
@Component

@Slf4j

@RequiredArgsConstructor

public class WebSocketEventListener {


  private final SimpMessageSendingOperations messagingTemplate;


  @EventListener

  public void handleWebSocketDisconnectListener(SessionDisconnectEvent
event) {

    StompHeaderAccessor headerAccessor =
StompHeaderAccessor.wrap(event.getMessage());

    String username = (String)
headerAccessor.getSessionAttributes().get("username");

    if (username != null) {

      log.info("user disconnected: {}", username);

      var chatMessage = ChatMessage.builder()

          .type(ChatMessage.MessageType.LEAVE)

          .sender(username)

          .build();

      messagingTemplate.convertAndSend("/topic/public", chatMessage);

    }
    }


}
```

This code is a Spring Framework component class that listens for WebSocket disconnect events and handles the disconnection by sending a leave message to the chat room. It is responsible for notifying other clients when a user disconnects from the chat.

Dependencies

This code relies on the following Spring Framework imports:

- org.springframework.stereotype.Component: Marks the class as a Spring component.
- lombok.*: Lombok library annotations for generating boilerplate code.
- org.springframework.messaging.simp.SimpMessageSendingOperations: Provides operations for sending messages to WebSocket subscribers.
- org.springframework.context.event.EventListener: Annotation for handling application events.

Class Declaration

- @Component: Annotation indicating that this class is a Spring component.
- @Slf4j: Annotation for generating a logger field using Lombok.
- @RequiredArgsConstructor: Annotation for generating a constructor with required dependencies.

Constructor

- WebSocketEventListener(SimpMessageSendingOperations messagingTemplate): Constructor that injects an instance of SimpMessageSendingOperations (used for sending messages to WebSocket subscribers) as a dependency.

Methods

handleWebSocketDisconnectListener

This method is annotated with @EventListener and is invoked when a SessionDisconnectEvent occurs, indicating that a WebSocket session has been disconnected. It retrieves the username from the session attributes, constructs a leave message using the ChatMessage class, and sends it to the "/topic/public" destination using the messagingTemplate.

- SessionDisconnectEvent: The event representing a WebSocket session disconnection.
- StompHeaderAccessor: A wrapper for the message headers.
- SimpMessageSendingOperations: Operations for sending messages to WebSocket subscribers.

This component class is responsible for handling WebSocket disconnect events. When a user disconnects from the chat,
the handleWebSocketDisconnectListener method is triggered. It retrieves the username from the session attributes and constructs a leave message using the ChatMessage class. The leave message is then sent to the "/topic/public" destination using the messagingTemplate, notifying other clients about the user's disconnection. The @Component annotation ensures that this class is automatically detected and registered as a Spring bean.

Chat Model

Chat model is the message payload which will be exchanged between the client side and server side of the application.

Path: model>ChatMessage.java

```
import lombok.*;

import java.nio.file.FileStore;

@Getter

@Setter

@AllArgsConstructor

@NoArgsConstructor

@Builder

public class ChatMessage {

   private String content;

   private String sender;

   private MessageType type;
```

```
public enum MessageType {

  CHAT, LEAVE, JOIN

}


public String getContent() {

  return content;

}


public void setContent(String content) {

  this.content = content;

}


public String getSender() {

  return sender;

}


public void setSender(String sender) {

  this.sender = sender;

}


public MessageType getType() {

  return type;

}


public void setType(MessageType type) {

  this.type = type;

}
```

```
}
```

This code defines a ChatMessage class representing a message in a chat application. It utilizes the Lombok library to automatically generate getters, setters, constructors, and builder methods.

Dependencies

This code relies on the following imports:

- import lombok.*: Lombok library annotations for generating boilerplate code.

Class Declaration

- @Getter: Annotation indicating that Lombok should generate getter methods for all fields.
- @Setter: Annotation indicating that Lombok should generate setter methods for all fields.
- @AllArgsConstructor: Annotation indicating that Lombok should generate a constructor with all fields as arguments.
- @NoArgsConstructor: Annotation indicating that Lombok should generate a default constructor with no arguments.
- @Builder: Annotation indicating that Lombok should generate a builder method for the class.

Fields

- content: Represents the content of the chat message.
- sender: Represents the sender of the chat message.
- type: Represents the type of the chat message, which is defined as an enum called MessageType.

Enum: MessageType

This enum defines the possible types of chat messages: CHAT, LEAVE, and JOIN.

Methods

The class also includes getter and setter methods for each field to access and modify their values.

- getContent(): Returns the content of the chat message.
- setContent(String content): Sets the content of the chat message.

- getSender(): Returns the sender of the chat message.
- setSender(String sender): Sets the sender of the chat message.
- getType(): Returns the type of the chat message.
- setType(MessageType type): Sets the type of the chat message.

This class serves as a model for chat messages in the application. It encapsulates the content, sender, and type of a chat message. The Lombok annotations eliminate the need to write repetitive boilerplate code for getters, setters, constructors, and builders, making the class concise and easier to work with.

Chat Controller

Path: controller>ChatController.java

```
import org.springframework.messaging.handler.annotation.*;
import org.springframework.messaging.simp.SimpMessageHeaderAccessor;
import org.springframework.stereotype.Controller;


@Controller
public class ChatController {


  @MessageMapping("/chat.register")
  @SendTo("/topic/public")
  public ChatMessage register(@Payload ChatMessage chatMessage,
SimpMessageHeaderAccessor headerAccessor) {
    headerAccessor.getSessionAttributes().put("username",
chatMessage.getSender());
    return chatMessage;
  }


  @MessageMapping("/chat.send")
  @SendTo("/topic/public")
```

```
    public ChatMessage sendMessage(@Payload ChatMessage chatMessage) {
        return chatMessage;
    }
}
```

This code is a Spring Framework controller class that handles WebSocket messages for a chat application. It contains two message handling methods: register and sendMessage.

Dependencies

This code relies on the following Spring Framework imports:

- org.springframework.messaging.handler.annotation.*: Provides annotations for handling WebSocket messages.
- org.springframework.messaging.simp.SimpMessageHeaderAccessor: Allows access to message headers and session attributes.
- org.springframework.stereotype.Controller: Marks the class as a controller component.

Class Declaration

- @Controller: Annotation indicating that this class is a controller component.

Methods

register

This method is annotated with @MessageMapping("/chat.register") and is invoked when a client sends a message to the "/app/chat.register" destination. It receives a ChatMessage object as a payload and a SimpMessageHeaderAccessor object to access message headers.

The method sets the sender's username from the received ChatMessage object as a session attribute using the headerAccessor. It returns the same ChatMessage object, which will be broadcasted to all subscribers of the "/topic/public" channel.

- @Payload: Annotation indicating that the method parameter should be bound to the payload of the received message.
- @SendTo("/topic/public"): Annotation specifying that the return value of the method should be sent as a message to the "/topic/public" destination.
- Return: The ChatMessage object received as the payload.

sendMessage

This method is annotated with @MessageMapping("/chat.send") and is invoked when a client sends a message to the "/app/chat.send" destination. It receives a ChatMessage object as a payload.

The method simply returns the received ChatMessage object, which will be broadcasted to all subscribers of the "/topic/public" channel.

- @Payload: Annotation indicating that the method parameter should be bound to the payload of the received message.
- @SendTo("/topic/public"): Annotation specifying that the return value of the method should be sent as a message to the "/topic/public" destination.
- Return: The ChatMessage object received as the payload.

This controller class is responsible for handling incoming WebSocket messages related to user registration and sending chat messages. The register method is used to register a user by storing their username in the session attributes, while the sendMessage method is used to broadcast chat messages to all connected clients. These methods are annotated to specify the destination paths for the messages and the return destinations for broadcasting.

UI

Path: resources directory

Basic HTML, CSS code and;

JavaScript code
Path: resources>js>main.js

```
"use strict";


var usernamePage = document.querySelector("#username-page");

var chatPage = document.querySelector("#chat-page");

var usernameForm = document.querySelector("#usernameForm");

var messageForm = document.querySelector("#messageForm");

var messageInput = document.querySelector("#message");

var messageArea = document.querySelector("#messageArea");
var connectingElement = document.querySelector(".connecting");


var stompClient = null;

var username = null;

//mycode

var password = null;


var colors = \[

  "#2196F3",

  "#32c787",

  "#00BCD4",

  "#ff5652",

  "#ffc107",

  "#ff85af",

  "#FF9800",

  "#39bbb0",

  "#fcba03",

  "#fc0303",

  "#de5454",

  "#b9de54",

  "#54ded7",

  "#54ded7",

  "#1358d6",

  "#d611c6",

\];


function connect(event) {

  username = document.querySelector("#name").value.trim();
  password = document.querySelector("#password").value;
  if (username) {
    // Please create a password
    if (password == "YOUR PASSWORD") {
      usernamePage.classList.add("hidden");
      chatPage.classList.remove("hidden");

      var socket = new SockJS("/websocket");
      stompClient = Stomp.over(socket);

      stompClient.connect({}, onConnected, onError);
    } else {
      let mes = document.getElementById("mes");
      mes.innerText = "Wrong password";
    }
  }
  event.preventDefault();
}

function onConnected() {
  // Subscribe to the Public Topic
  stompClient.subscribe("/topic/public", onMessageReceived);

  // Tell your username to the server
  stompClient.send(
    "/app/chat.register",
    {},
    JSON.stringify({ sender: username, type: "JOIN" })
```

```
 );

 connectingElement.classList.add("hidden");
}

function onError(error) {
 connectingElement.textContent =
  "Could not connect to WebSocket! Please refresh the page and try again or
contact your administrator.";
 connectingElement.style.color = "red";
}

function send(event) {
 var messageContent = messageInput.value.trim();

 if (messageContent && stompClient) {
  var chatMessage = {
   sender: username,
   content: messageInput.value,
   type: "CHAT",
  };

  stompClient.send("/app/chat.send", {}, JSON.stringify(chatMessage));
  messageInput.value = "";
 }
 event.preventDefault();
}
```

```
function onMessageReceived(payload) {
 var message = JSON.parse(payload.body);

 var messageElement = document.createElement("li");

 if (message.type === "JOIN") {
  messageElement.classList.add("event-message");
  message.content = message.sender + " joined!";
 } else if (message.type === "LEAVE") {
  messageElement.classList.add("event-message");
  message.content = message.sender + " left!";
 } else {
  messageElement.classList.add("chat-message");

  var avatarElement = document.createElement("i");
  var avatarText = document.createTextNode(message.sender\[0\]);
  avatarElement.appendChild(avatarText);
  avatarElement.style\["background-color"\] = getAvatarColor(message.sender);

  messageElement.appendChild(avatarElement);

  var usernameElement = document.createElement("span");
  var usernameText = document.createTextNode(message.sender);
  usernameElement.appendChild(usernameText);
  messageElement.appendChild(usernameElement);
  // \* update
  usernameElement.style\["color"\] = getAvatarColor(message.sender);
  //\* update end
 }

 var textElement = document.createElement("p");
 var messageText = document.createTextNode(message.content);
 textElement.appendChild(messageText);

 messageElement.appendChild(textElement);
 // \* update
 if (message.sender === username) {
  // Add a class to float the message to the right
  messageElement.classList.add("own-message");
 } // \* update end
 messageArea.appendChild(messageElement);
 messageArea.scrollTop = messageArea.scrollHeight;
}

function getAvatarColor(messageSender) {
 var hash = 0;
 for (var i = 0; i < messageSender.length; i++) {
  hash = 31 \* hash + messageSender.charCodeAt(i);
 }

 var index = Math.abs(hash % colors.length);
 return colors\[index\];
}

usernameForm.addEventListener("submit", connect, true);
messageForm.addEventListener("submit", send, true);
```

This code is a client-side JavaScript code that enables a user to connect to a chat application using a username and password. It utilizes the Stomp protocol to establish a WebSocket connection and exchange messages with a server.

Variables

- usernamePage: Represents the element on the page that contains the username input form.
- chatPage: Represents the element on the page that displays the chat interface.
- usernameForm: Represents the form element for submitting the username.
- messageForm: Represents the form element for submitting chat messages.
- messageInput: Represents the input field where the user enters chat messages.
- messageArea: Represents the area where chat messages are displayed.
- connectingElement: Represents the element that indicates the connection status.
- stompClient: Holds the Stomp client instance used for WebSocket communication.
- username: Stores the username entered by the user.
- password: Stores the password entered by the user.
- colors: An array of colors used for generating avatar colors.

Functions

connect(event)

This function is called when the username form is submitted. It retrieves the username and password entered by the user, checks if the password is correct, and if so, establishes a WebSocket connection to the server. It hides the username page and displays the chat page.

- event: The event object representing the form submission event.

onConnected()

This function is called when the WebSocket connection is successfully established. It subscribes to the "/topic/public" channel to receive messages from the server. It also sends a JOIN message to the server to notify the user's username.

onError(error)

This function is called when an error occurs during the WebSocket connection. It updates the connecting element to display an error message.

- error: The error object representing the connection error.

send(event)

This function is called when the chat message form is submitted. It retrieves the message content, sends the message to the server using the Stomp client, and clears the message input field.

- event: The event object representing the form submission event.

onMessageReceived(payload)

This function is called when a message is received from the server. It parses the message payload, creates the necessary DOM elements to display the message, and appends them to the message area.

- payload: The message payload received from the server.

getAvatarColor(messageSender)

This function generates an avatar color based on the message sender's username. It calculates a hash value based on the characters of the username and maps it to an index in the colors array to select a color.

- messageSender: The username of the message sender.

Event Listeners

- usernameForm.addEventListener("submit", connect, true): Listens for the submission of the username form and calls the connect function.
- messageForm.addEventListener("submit", send, true): Listens for the submission of the chat message form and calls the send function.