UNIVERSIDAD DE LAS FUERZAS

ARMADAS – ESPE

DEPARTAMENTO DE CIENCIAS DE LA

COMPUTACIÓN INGENIERÍA DE

SOFTWARE

APLICACIONES DISTRIBUIDAS

CHAT CON WEBSOCKETS

Pamela Montenegro

2025

Contents

Introducción

En la actualidad, los servicios de mensajería son fundamentales en diversas aplicaciones, como pedir comida, solicitar un taxi, realizar compras en línea o recibir soporte en vivo de empresas. Este proyecto tiene como propósito crear un sistema de mensajería en tiempo real que permita una comunicación instantánea y eficiente entre los usuarios, utilizando tecnologías como WebSockets para asegurar una interacción bidireccional y fluida entre el cliente y el servidor.

El sistema a desarrollar estará basado en tecnologías populares como Spring Boot para gestionar todo el backend, cubriendo funciones como la conexión, mensajería, manejo de excepciones y la lógica de negocio, similar a la utilizada en aplicaciones como WhatsApp. Además, contará con un frontend construido en React, lo que facilitará la gestión de interfaces de usuario mediante componentes reutilizables. También se incorporará la librería React Redux para mantener un estado global en la aplicación, garantizando la sincronización adecuada de los datos.

Objetivos Específicos

- Crear un sistema de mensajería en tiempo real que facilite la comunicación bidireccional entre varios usuarios a través de WebSockets, con una interfaz fácil de usar y características avanzadas, como mensajes entre usuarios y notificaciones de conexión o desconexión.

- Establecer una comunicación instantánea que permita a los usuarios enviar y recibir mensajes en tiempo real.

- Diseñar una interfaz sencilla que permita a los usuarios enviar y recibir mensajes, consultar su historial de conversaciones y recibir alertas sobre nuevos mensajes, conexiones o desconexiones.

Marco Teórico

- **Definición y funcionamiento de WebSockets.**

  WebSocket es un protocolo de comunicación que establece una conexión bidireccional y persistente entre el cliente y el servidor, permitiendo el intercambio de datos en tiempo real sin la necesidad de realizar múltiples solicitudes HTTP. Una vez establecida la conexión, tanto el cliente como el servidor pueden enviar y recibir mensajes de manera simultánea y continua. Este protocolo es especialmente útil en aplicaciones que requieren actualizaciones en tiempo real, como chats en línea, juegos interactivos y plataformas de trading financiero (Navitsky, M.,2024).

- **Diferencias entre WebSockets y otros protocolos de comunicación como HTTP.**

La principal diferencia entre WebSockets y HTTP radica en la naturaleza de la comunicación:

HTTP: Opera bajo un modelo de solicitud-respuesta unidireccional. Cada vez que el cliente necesita información, envía una solicitud al servidor, que responde con los datos requeridos. Una vez que se envía la respuesta, la conexión se cierra.

WebSockets: Establecen una conexión bidireccional y persistente, permitiendo que tanto el cliente como el servidor envíen y reciban datos en cualquier momento sin necesidad de reestablecer la conexión (Navitsky, M.,2024).

- **Casos de uso comunes de WebSockets, especialmente en chatbots y aplicaciones interactivas.**

  WebSockets son ideales para aplicaciones que requieren comunicación en tiempo real, tales como:

  Chatbots: Permiten una interacción instantánea entre el usuario y el sistema, mejorando la experiencia del usuario al recibir respuestas inmediatas.

Aplicaciones de mensajería: Facilitan la transmisión instantánea de mensajes entre usuarios, como en aplicaciones de chat en línea (Navitsky, M.,2024).

Juegos en línea: Soportan la comunicación en tiempo real entre jugadores y servidores, esencial para juegos multijugador interactivos.

Plataformas de trading financiero: Proporcionan actualizaciones en tiempo real de los mercados financieros, permitiendo a los usuarios tomar decisiones informadas rápidamente (Navitsky, M.,2024).

- **Conceptos básicos de React y su enfoque basado en componentes.**

  React es una biblioteca de JavaScript desarrollada por Facebook para construir interfaces de usuario. Su enfoque basado en componentes permite dividir la interfaz en unidades reutilizables y autónomas, facilitando el desarrollo y mantenimiento de aplicaciones complejas. Cada componente puede tener su propio estado y lógica, lo que promueve una arquitectura modular y escalable (Navitsky, M.,2024).
- **Gestión de estados y manejo de eventos en React para una experiencia interactiva.**

  En React, el estado se refiere a los datos que determinan el comportamiento y la apariencia de un componente. La gestión eficiente del estado es crucial para crear interfaces interactivas y dinámicas. React proporciona un sistema de estado local para componentes individuales y herramientas como Context API y Redux para manejar estados globales. El manejo de eventos en React se realiza mediante funciones que responden a acciones del usuario, como clics o entradas de teclado, permitiendo una interacción fluida y reactiva (Navitsky, M.,2024).
- **Integración de WebSockets en React para una comunicación bidireccional.**

Integrar WebSockets en una aplicación React permite establecer una comunicación bidireccional en tiempo real entre el cliente y el servidor. Esto se logra creando una instancia de WebSocket en el componente React y gestionando los eventos de apertura, recepción de mensajes y cierre de la conexión. Esta integración es especialmente útil en aplicaciones que requieren actualizaciones en tiempo real, como chats en línea o notificaciones en vivo (Navitsky, M.,2024)..

- **Optimización de la UI para interacciones en tiempo real y experiencia del usuario.**

  Integrar WebSockets en una aplicación React permite establecer una comunicación bidireccional en tiempo real entre el cliente y el servidor. Esto se logra creando una instancia de WebSocket en el componente React y gestionando los eventos de apertura, recepción de mensajes y cierre de la conexión. Esta integración es especialmente útil en aplicaciones que requieren actualizaciones en tiempo real, como chats en línea o notificaciones en vivo (Navitsky, M.,2024)..

- **Uso de tecnologías de bases de datos relacionales o no relacionales para persistir información.**

  La elección entre bases de datos relacionales y no relacionales depende de las necesidades específicas de la aplicación:

  Bases de datos relacionales: Utilizan un esquema estructurado con tablas y relaciones entre ellas. Son ideales para aplicaciones que requieren integridad referencial y operaciones complejas de consulta. Ejemplos incluyen MySQL y PostgreSQL.

Bases de datos no relacionales: Ofrecen flexibilidad en el esquema y son adecuadas para manejar grandes volúmenes de datos no estructurados o semi-estructurados. Son útiles en aplicaciones que requieren escalabilidad horizontal y alta disponibilidad. Ejemplos incluyen MongoDB y Cassandra.

La elección entre estas tecnologías debe basarse en los requisitos específicos de la aplicación, como la complejidad de las consultas, la necesidad de escalabilidad y la naturaleza de los datos a manejar.

Desarrollo

Requisitos funcionales

Comunicación en tiempo real

Interfaz de usuario

Para ingresar al sistema de un Login.jsx, el cual se encarga de realizar la conexión con el websocket, así también como el diseño principal del chat con sus componentes acoplados. Dicho archivo se lo muestra en el ChatRoom.jsx.

Login.jsx y ChatRoom.jsx

```
import { useState, useEffect } from 'react';
import { Form, Button, Container, Row, Col } from 'react-bootstrap';
import './Login.css';
import './ChatRoom.css';
import PropTypes from 'prop-types';
import useWebSocket from './useWebSocket';


const Login = ({ connect }) => {
  const [isConnected, setIsConnected] = useState(false);
  const [users, setUsers] = useState([]);
  const [selectedUser, setSelectedUser] = useState(null);
  const { messages, sendMessage, sendGeneralMessage, setMessages, nickname, fullname, generalMessages, disconnect } =
useWebSocket();


  const handleSubmit = (event) => {
    event.preventDefault();
    const nickname = event.target.elements.nickname.value;
```

```
  const fullname = event.target.elements.fullname.value;
  connect(nickname, fullname);
  setIsConnected(true);
};

useEffect(() => {
  if (isConnected) {
    const fetchUsers = () => {
      fetch('http://localhost:2173/users')
        .then(response => response.json())
        .then(data => {
          const filteredUsers = data.filter(user => user.nickName !== nickname);
          setUsers(filteredUsers);
        })
        .catch(error => console.error('Error fetching users:', error));
    };

    fetchUsers();
    const intervalId = setInterval(fetchUsers, 1000);
    return () => clearInterval(intervalId);
  }
}, [isConnected, nickname]);

const handleUserSelect = (user) => {
  setSelectedUser(user);
  fetch(`http://localhost:2173/messages/${nickname}/${user.nickName}`)
    .then(response => response.json())
    .then(data => setMessages(data))
    .catch(error => console.error('Error fetching messages:', error));
};

const handleSendMessage = (event) => {
```

```
  event.preventDefault();
  const messageContent = event.target.elements.message.value;
  if (messageContent.trim()) {
    if (selectedUser) {
      sendMessage(selectedUser.nickName, messageContent);
    } else {
      sendGeneralMessage(messageContent);
    }
    event.target.elements.message.value = '';
  }
};

const handleLogout = () => {
  disconnect();
  setIsConnected(false);
};

return (
  isConnected ? (
    <div className="chat-room-container d-flex">
      <div className="user-list p-3">
        <h5>Contacts</h5>
        <ul className="list-unstyled">
          <li
            className={`user-item py-2 ${selectedUser === null ? 'active' : ''}`}
            onClick={() => setSelectedUser(null)}
          >
            General Chat
          </li>
          {users.map(user => (
            <li
              key={user.nickName}
```

```
            className={`user-item py-2 ${selectedUser && selectedUser.nickName === user.nickName ? 'active' : ''}`}
            onClick={() => handleUserSelect(user)}
          >
            {user.fullName}
          </li>
        ))}
      </ul>
      <div className="user-footer">
        <span>{fullname}</span>
        <button onClick={handleLogout} className="btn btn-secondary mt-2">Salir</button>
      </div>
    </div>
    <div className="chat-window p-3">
      <h5>{selectedUser ? selectedUser.fullName : 'General Chat'}</h5>
      <div className="chat-area">
        {(selectedUser ? messages : generalMessages).map((msg, index) => (
          <div key={index} className={`message ${msg.senderId === nickname ? 'sent' : 'received'}`}>
            {selectedUser ? msg.content : <><strong>{msg.senderId}</strong>: {msg.content}</>}
          </div>
        ))}
      </div>
      <form className="chat-input mt-3" onSubmit={handleSendMessage}>
        <input type="text" className="form-control" placeholder="Type your message" name="message" />
        <Button type="submit" className="btn btn-primary mt-2">Send</Button>
      </form>
    </div>
  </div>
) : (
  <Container className="d-flex justify-content-center align-items-center vh-100">
    <Row>
      <Col md={12} className="text-center">
        <h1 className="text-white">WEB CHAT</h1>
```

```
          <h2 className="text-white">LOGIN</h2>
        </Col>
        <Col md={12}>
          <Form className="p-4 rounded" onSubmit={handleSubmit}>
            <Form.Group controlId="formNickname" className="mb-3">
              <Form.Label className="text-white">Nickname</Form.Label>
              <Form.Control type="text" placeholder="Enter nickname" name="nickname" required />
            </Form.Group>
            <Form.Group controlId="formFullname" className="mb-3">
              <Form.Label className="text-white">Fullname</Form.Label>
              <Form.Control type="text" placeholder="Enter fullname" name="fullname" required />
            </Form.Group>
            <Button variant="primary" type="submit" className="w-100">Sign In</Button>
            <Form.Text className="text-white d-block text-center mt-3">
              <a href="#" className="text-white">Forgot password?</a>
            </Form.Text>
          </Form>
        </Col>
      </Row>
    </Container>
  )
 );
};

Login.propTypes = {
  connect: PropTypes.func.isRequired,
};
```

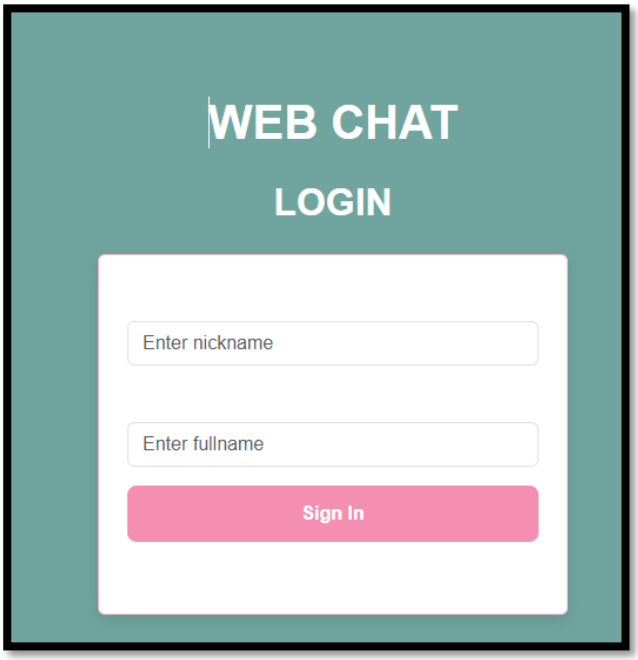
```
export default Login;
import './ChatRoom.css';

const ChatRoom = () => {
  return (
```

```
    <div className="chat-room-container d-flex">
      <div className="user-list p-3">
        <h5>Contacts</h5>
        <ul className="list-unstyled">
          <li className="user-item py-2">User 2</li>
          <li className="user-item py-2">User 3</li>
        </ul>
      </div>
      <div className="chat-window p-3">
        <h5>Chats</h5>
        <div className="chat-area"></div>
        <form className="chat-input mt-3">
          <input type="text" className="form-control" placeholder="Type your message" />
          <button type="submit" className="btn btn-primary mt-2">Send</button>
        </form>
      </div>
    </div>
  );
};
```

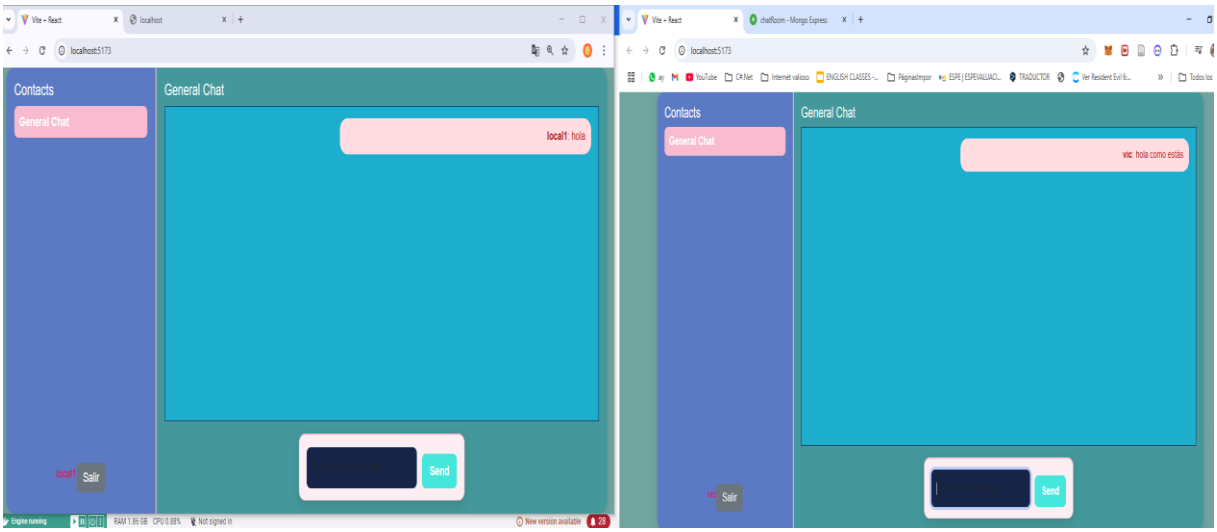
export default ChatRoom ;La interfaz gráfica consta de un Login, en el cual permite ingresar a un usuario registrado en el sistema. Tal como se muestra en la Figura 1.

Figura 1. Login del sistema



Dentro del sistema, se puede observar que se encuentran los usuarios y chats de todos los usuarios. En la figura 3, se observa como se muestra el chat general.

Figura 3. Chats generales



Estas partes se las puede observar el archivo, que se encargan del diseño como Login.css.

En el proyecto se usaron tecnologías tanto como para el Backend como para el Frontend, las cuales se detallan a continuación:

Backend: Spring Boot para implementar el servidor WebSocket

El backend está utilizando Spring Boot para manejar la comunicación WebSocket.

Configuración de WebSockets:

- Se ha configurado WebSocketConfig.java con @EnableWebSocketMessageBroker, lo que permite la comunicación en tiempo real mediante WebSockets.
- Se define el endpoint /ws para permitir que los clientes se conecten.
- Se usa STOMP (Simple Text Oriented Messaging Protocol) como protocolo de mensajería.
- setApplicationDestinationPrefixes("/app"): Define prefijos para las rutas de mensajes enviados por los clientes.
- enableSimpleBroker("/group", "/user"): Permite la suscripción a estos prefijos para recibir mensajes.
- setUserDestinationPrefix("/user"): Permite mensajes privados a usuarios.

Frontend: Cliente en JavaScript (React)

- Se usa SockJS como cliente para WebSocket.
- stompClient.subscribe("/group/{chatId}", onMessageReceive); permite suscribirse a

mensajes en tiempo real.

Protocolo: Comunicación bidireccional con WebSockets

La comunicación se establece de la siguiente manera:

- Conexión del Cliente al Servidor: El frontend se conecta al WebSocket de Spring Boot en connect(), usando STOMP y SockJS.
- Envío de Mensajes desde el Cliente: Cuando un usuario escribe un mensaje y lo envía, se llama a handleCreateNewMessage():

Recepción de Mensajes en el Servidor y Redistribución:

- El mensaje es recibido en RealTimeChat.java en @MessageMapping("/message").
- Luego, el servidor reenvía el mensaje a los clientes suscritos a /group/{chatId}.

Recepción de Mensajes en el Cliente:

- El frontend escucha los mensajes usando stompClient.subscribe("/group/{chatId}", onMessageReceive);.
- Cuando un nuevo mensaje llega, React actualiza el estado y lo muestra en la UI:

Comunicación en Tiempo Real

La aplicación de chat permite la comunicación en tiempo real utilizando Spring Boot en el backend y React con Redux en el frontend. Para esto se usa WebSockets con STOMP para establecer canales de comunicación entre los clientes y el servidor.

El backend en Spring Boot gestiona WebSockets usando WebSocketConfig.java. Este archivo configura la comunicación y define los canales a los cuales los clientes pueden suscribirse.

WebSocketConfig.java

```
package com.whatsappClone.config;




import org.springframework.context.annotation.Configuration;

import org.springframework.messaging.simp.config.MessageBrokerRegistry;


import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
```

```
@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {




   @Override

   public void registerStompEndpoints(StompEndpointRegistry registry) {
      registry.addEndpoint("/ws").setAllowedOrigins("*").withSockJS();
   }



   @Override
```

La activación de WebSockets en el backend se realiza mediante la anotación @EnableWebSocketMessageBroker, la cual habilita la comunicación en tiempo real utilizando STOMP. Se registra el endpoint /ws para aceptar conexiones WebSocket y se configura con setAllowedOrigins("*") y SockJS, asegurando la compatibilidad con navegadores que no soportan WebSockets de manera nativa. Además, el broker de mensajes gestiona la distribución de los mensajes: aquellos enviados por los clientes deben empezar con el prefijo /app, mientras que los mensajes dirigidos a un grupo se envían a /group/{chatId} y los privados a /user/{userId}, garantizando una correcta asignación de los mensajes según su destinatario.

El backend procesa los mensajes en tiempo real a través de la clase RealTimeChat.java.

En el archivo RealTimeChat.java, el método marcado con @MessageMapping("/message") actúa como un controlador para los mensajes entrantes enviados por el cliente a /app/message. Dentro de este método, se determina si el mensaje es grupal o privado. Si es un mensaje grupal, se publica en el destino /group/{chatId}, de modo que todos los usuarios

suscritos al chat lo reciban. Si es privado, se envía a /user/{chatId}, asegurando que solo el destinatario específico lo reciba. Esto permite una distribución eficiente de los mensajes según su tipo en el sistema de WebSocket. El frontend utiliza SockJS y STOMP.js para gestionar WebSockets. La conexión se establece en el siguiente código:

```
console.log("WebSocket conectado");

    },

    (error) => console.error("Error en WebSocket:", error)

  );

};
```

```
useEffect(() => {

  if (isConnected && stompClient) {
    stompClient.subscribe("/group/1", onMessageReceive);
  }

}, [isConnected]);
```

```
const onMessageReceive = (payload) => {

  const receivedMessage = JSON.parse(payload.body);

  dispatch({ type: "CREATE_NEW_MESSAGE", payload: receivedMessage });
  setMessages((prev) => [...prev, receivedMessage]);
};
```

En el frontend, la conexión WebSocket se establece utilizando SockJS, que crea un canal de comunicación con el servidor. A continuación, stompClient.connect() inicia la conexión con el backend y, una vez establecida, el cliente se suscribe a un canal específico, como /group/1, para recibir mensajes en tiempo real. Cada vez que llega un mensaje nuevo, stompClient.subscribe() ejecuta la función onMessageReceive(), que actualiza tanto el estado global en Redux como el estado local de los mensajes.

En el frontend, Redux se encarga de gestionar el estado global de los mensajes a través de acciones (Action.js) y reducers (Reducer.js). Las acciones se encargan de la lógica necesaria para enviar y recibir mensajes. Por ejemplo, cuando un usuario envía un mensaje, la acción createMessage() realiza una solicitud al backend, y una vez que se recibe la respuesta, se despacha una acción para actualizar el estado global.

Persistencia de datos

En este caso, se ha optado por utilizar una base de datos MySQL para gestionar las tablas relacionadas con los usuarios, chats y mensajes. Se ha creado una base de datos denominada "chatbot" y, en el archivo application.properties del backend en Spring Boot, se ha incluido la configuración necesaria para conectar con la base de datos, especificando el usuario, la contraseña y otros parámetros relevantes.

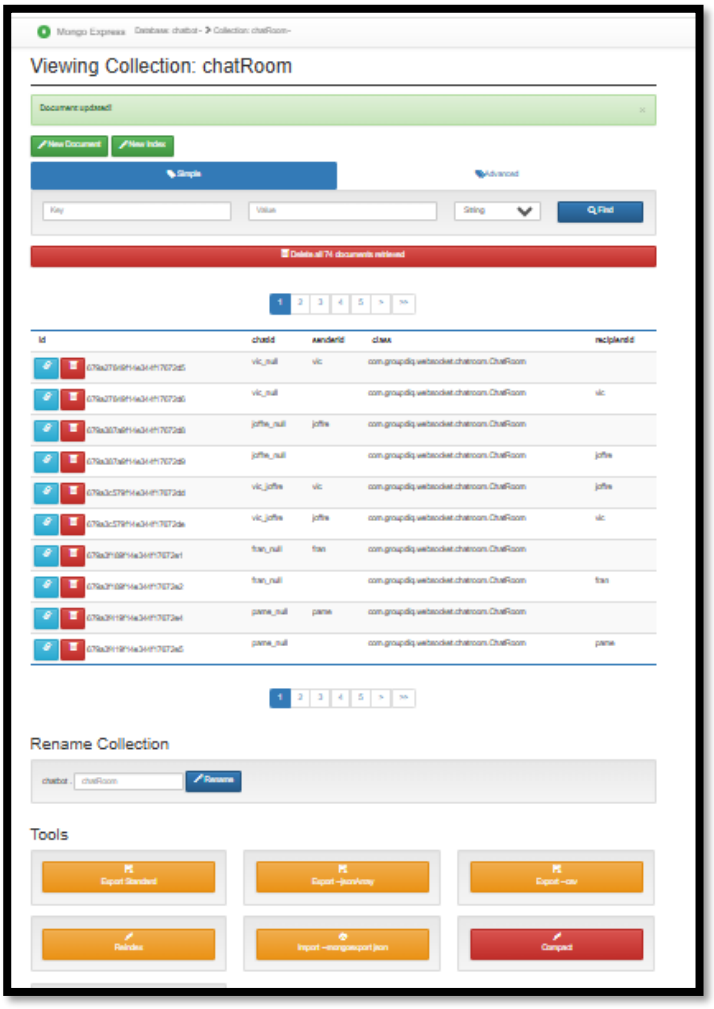
Application.propierties

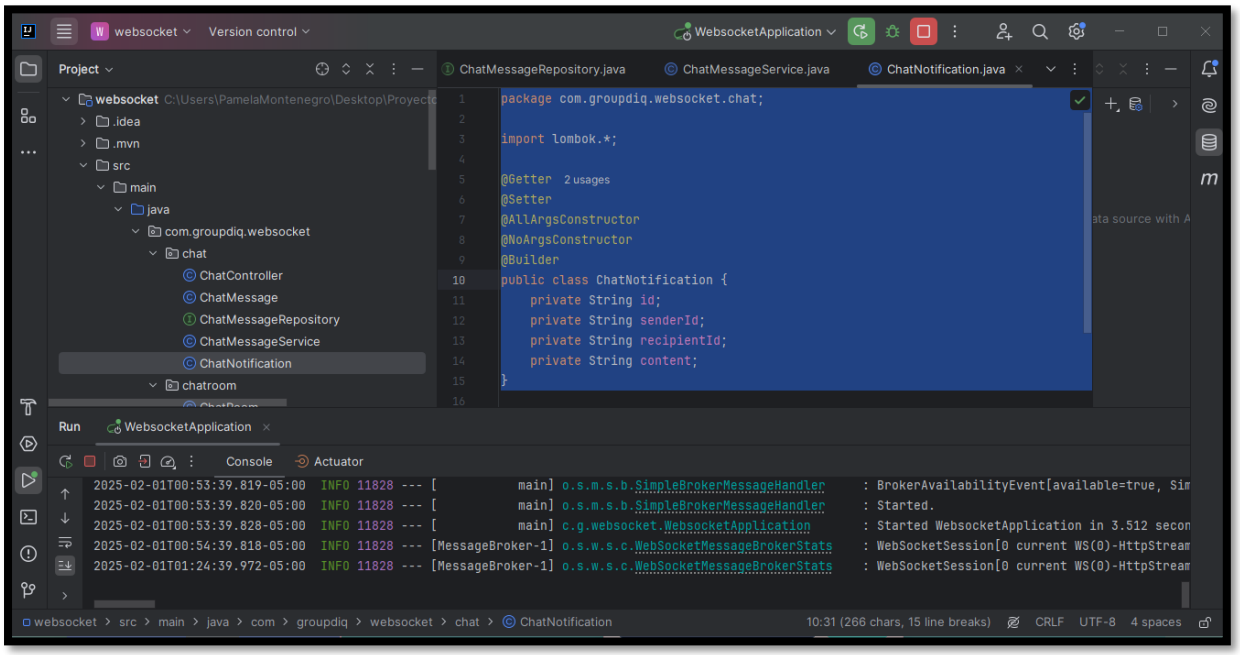
```
spring:
  data:
    mongodb:
      username:                                                    pamela
      password:                                                 pamela123
      host:                                                     localhost
```

Al ejecutar la aplicación, se crea las tablas correspondientes, tal como se observa en la figura 1.

Figura 1. Creación de base de datos

Bases de datos Mongo DB Express





Para comprender la creación de estas tablas, a continuación se muestra las entidades presentes en el backend.

ChatNotification.java

```
package com.groupdiq.websocket.chat;

import lombok.*;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ChatNotification {
    private String id;
    private String senderId;
    private String recipientId;
    private String content;
}
```

ChatMessage.java

```
 @Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Document
public class ChatMessage {
    @Id
    private String id;
    private String chatId;
    private String senderId;
    private String recipientId;
    private String content;
    private Date timestamp;
```

User.java

```
package com.groupdiq.websocket.user;

import lombok.Getter;
import lombok.Setter;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Getter
@Setter
@Document
public class User {
    @Id
    private String nickName;
    private String fullName;
    private Status status;
}
```

Mensajería

Desarrollar una sala de chat general (broadcast).

La funcionalidad del chat general se gestiona a través de WebSocket y se implementa en la API dentro del ChatController. A continuación, se presenta el código de la implementación en el ChatController, en el cual el chat se realiza entre un usuario específico y otro.

Conclusiones

- El desarrollo de este sistema de mensajería instantánea, utilizando tecnologías como Spring Boot, React, Websockets y SockJS, permitió una mejor comprensión de cómo manejar la comunicación en tiempo real entre cliente y servidor. Durante el proyecto, se lograron cumplir algunas de las funcionalidades planteadas, como la comunicación en tiempo real, a través de Websockets y SockJS, lo que garantizó la compatibilidad entre navegadores y aseguró que todos los usuarios pudieran acceder al sistema sin inconvenientes.

- La utilización de Spring Boot en este proyecto resultó ser una excelente opción, ya que ofrece una configuración sencilla, especialmente para la base de datos, permitiendo así la persistencia de los datos de los usuarios y los chats en la base de datos MySQL. Esto facilita la recuperación de las conversaciones dentro del sistema.

- Además del chat general, el sistema incorporó mensajería privada entre usuarios, demostrando su capacidad para adaptarse a escenarios de uso más complejos.

- Como posibles mejoras futuras, se sugiere implementar la autenticación de usuarios mediante OAuth2 y añadir soporte para archivos como imágenes, videos y

documentos, con el objetivo de mejorar la seguridad y funcionalidad del sistema.

Referencias

Haidi Zakelšek. (2024, June 24). *Advanced Redux Patterns - Haidi Zakelšek - Medium*. Medium. https://medium.com/@haidizakelek/advanced-redux-patterns-00837555bdf3

GeeksforGeeks. (2024, April 29). *How to Test WebSocket APIs With Postman?* GeeksforGeeks. https://www.geeksforgeeks.org/how-to-test-websocket-apis-with-postman/?ref=ml_lbp

Navitsky, M. (2024, May 10). *Spring Boot App Development with WebSockets and SockJS*. Elinext. https://www.elinext.com/blog/developing-spring-boot-app-with-websockets-and-sockjs/