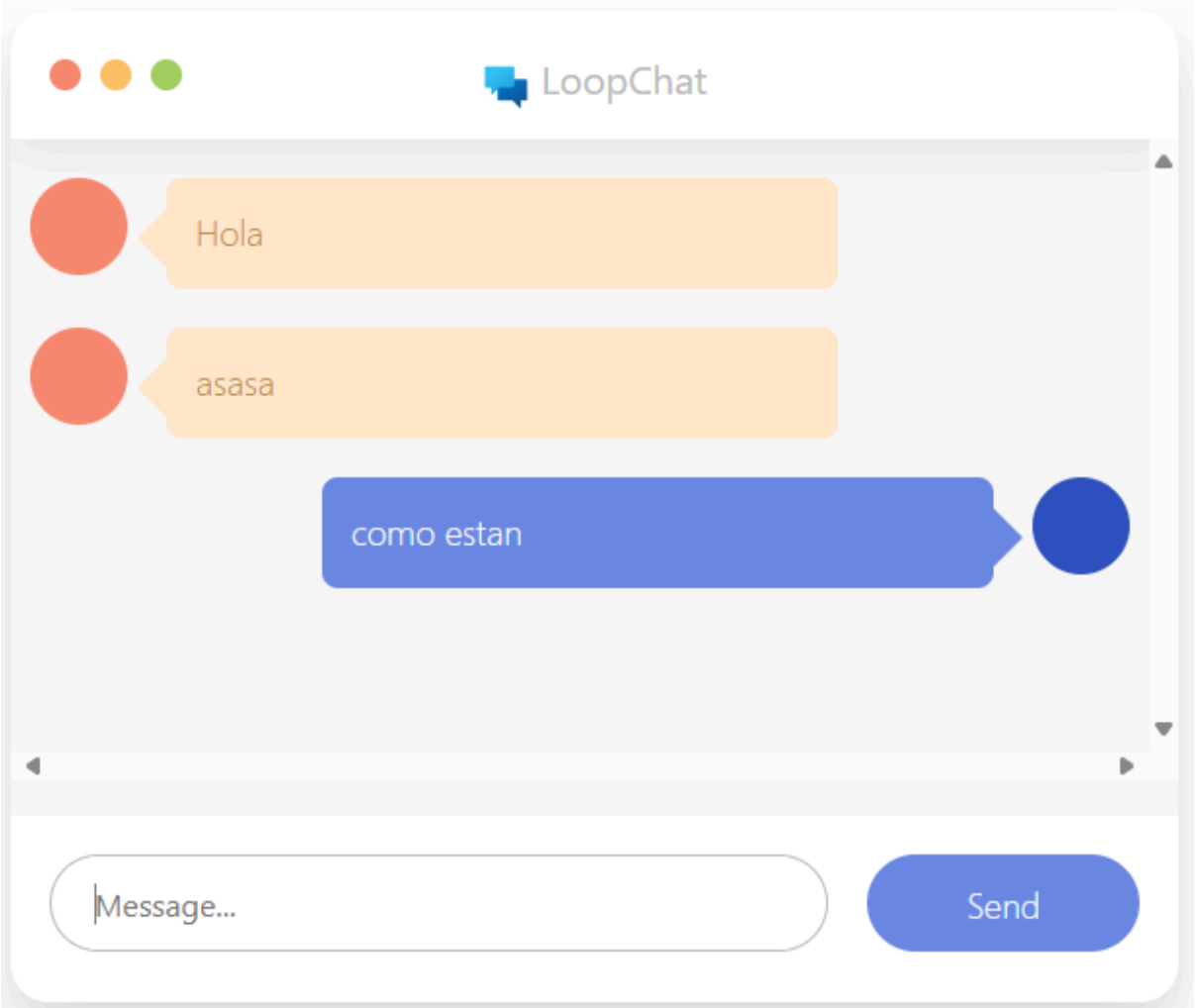
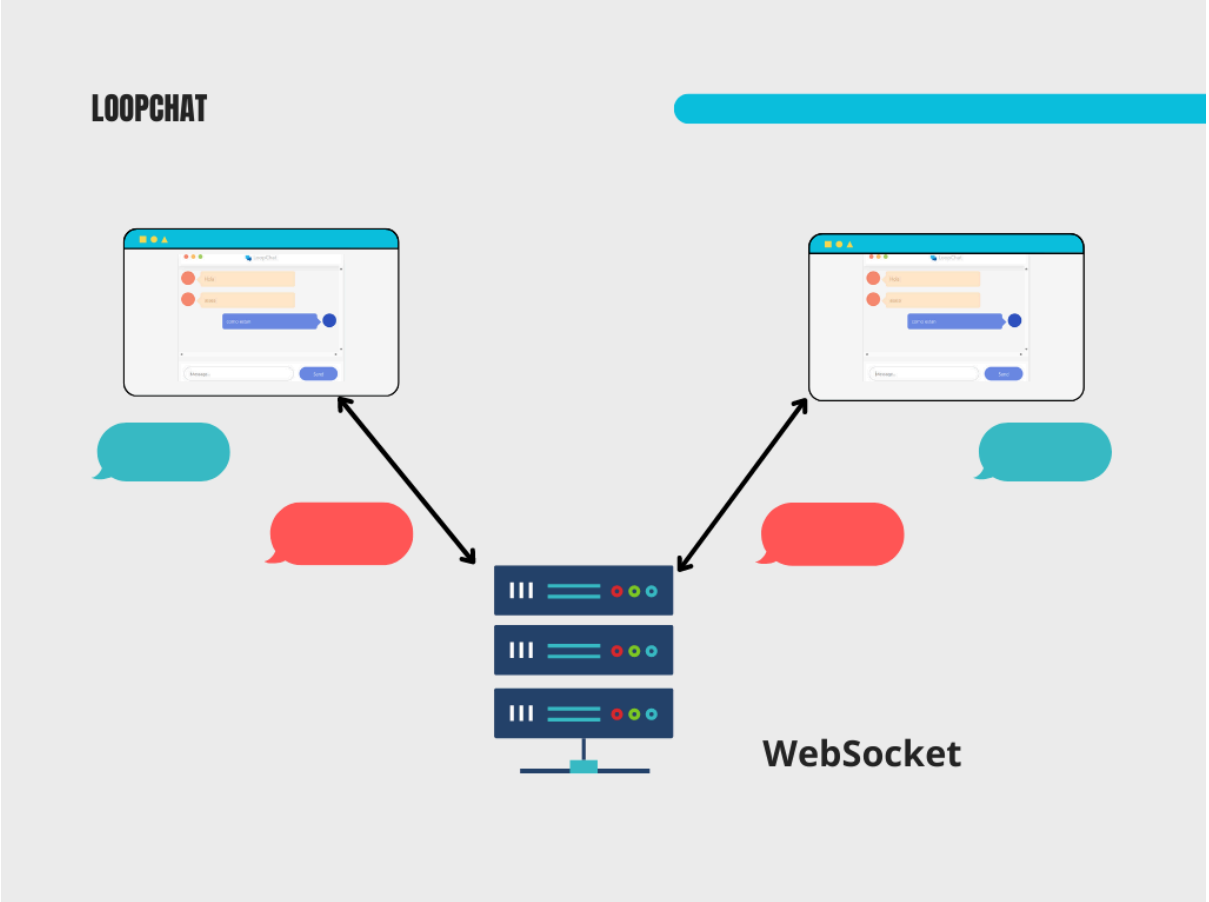
LoopChat

LoopChat es una aplicación de mensajería en tiempo real que utiliza tecnologías modernas como Angular, Spring Boot y WebSocket. Facilita la comunicación instantánea en salas de chat, permitiendo a los usuarios interactuar en una misma sala y recibir mensajes de forma simultánea. Utiliza STOMP para gestionar la comunicación, asegurando una experiencia de usuario fluida y responsiva. Ideal para aplicaciones de chat internas o plataformas de colaboración.



Como funciona el programa



LoopChat es una aplicación de chat en tiempo real que permite que varios usuarios se conecten a una misma sala y envíen mensajes. Utiliza STOMP sobre WebSocket para manejar la comunicación. Cuando un usuario envía un mensaje, este se envía al servidor Spring Boot, que retransmite el mensaje a todos los clientes conectados a la sala.

Diagrama General de LoopChat

1. **Inicia la Conexión WebSocket (initConnectionSocket)**
    - Se establece una conexión WebSocket usando SockJS y se configura el cliente STOMP.
    - Se define un comportamiento de reconexión automática en caso de desconexión.
    - Si la conexión se establece correctamente, se imprime "WebSocket connected" en la consola.

2. **Unirse a una Sala de Chat (joinRoom)**
    - Se verifica que el cliente STOMP esté conectado.
    - Si la conexión es exitosa, se suscribe al canal específico de la sala de chat mediante el identificador roomID.
    - Los mensajes recibidos de la sala son procesados y emitidos para ser mostrados en el frontend.
3. **Enviar Mensajes (sendMessage)**
    - Se verifica que el cliente STOMP esté conectado.
    - Se envía el mensaje al backend especificando el destino como /app/chat/{roomID}.
    - El mensaje se envía en formato JSON.
4. **Recibir Mensajes (onMessageReceived)**
    - Retorna un Observable que permite a los componentes de Angular suscribirse a los mensajes.
    - Cuando se recibe un mensaje, se actualiza la lista de mensajes en el frontend.

Descripción detallada de cada función en el servidor

1. configureMessageBroker(MessageBrokerRegistry registry)

- **Inicio**: Se ejecuta automáticamente cuando la aplicación inicia debido a que es un método sobrescrito de WebSocketMessageBrokerConfigurer.
- **Proceso**:
  - Activa un "simple broker" para gestionar los destinos con prefijo /topic.
  - Establece un prefijo de destino /app para mensajes de clientes, que los clientes usarán al enviar mensajes.
- **Resultado**: La configuración permite a los clientes enviar mensajes a /app, y recibir mensajes de los destinos con prefijo /topic.

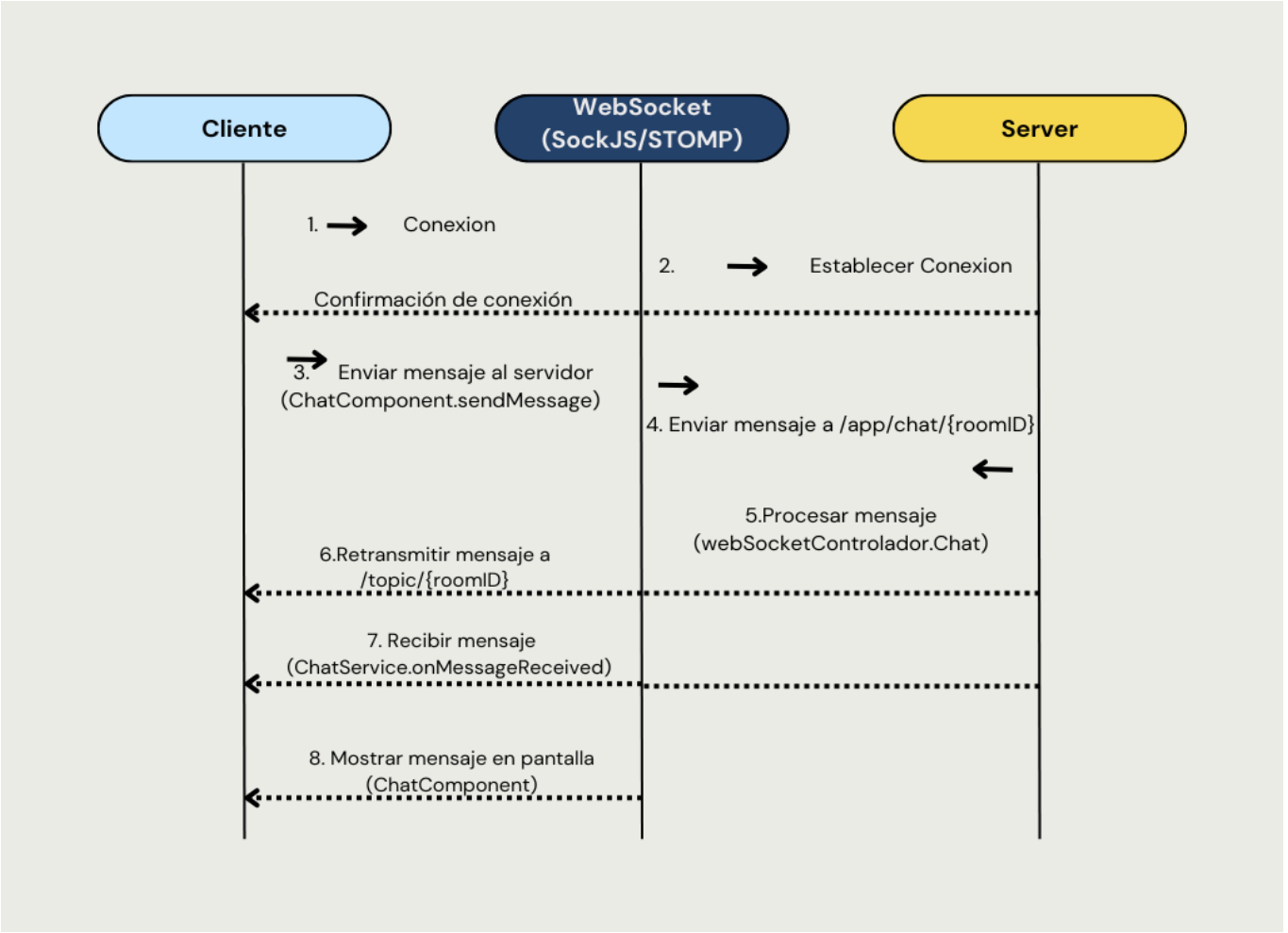
2. registerStompEndpoints(StompEndpointRegistry registry)

- **Inicio**: Se ejecuta durante el inicio de la aplicación.
- **Proceso**:
  - Registra el endpoint /chat-socket y habilita SockJS para navegadores sin soporte nativo de WebSockets.
  - Define http://localhost:4200 como origen permitido para conectar al WebSocket.
- **Resultado**: Se establece el endpoint donde los clientes podrán conectarse para establecer la comunicación en tiempo real.

3. Chat(String roomID, ChatMensaje mensaje)

- **Inicio**: Se ejecuta cuando un cliente envía un mensaje a /app/chat/{roomID}.
- **Proceso**:
    - Recibe el mensaje enviado por el cliente y lo imprime en la consola del servidor.
    - Devuelve el mensaje recibido, el cual será enviado automáticamente a todos los clientes suscritos a /topic/{roomID} gracias a la anotación @SendTo.
- **Resultado**: El mensaje se envía de vuelta a los clientes suscritos a la sala de chat especificada, permitiendo la comunicación en tiempo real entre los usuarios.

Diagrama de Secuencias del Programa



Descripción del Diagrama de Secuencias

1. **Conexión del Cliente**: El cliente (Angular) se conecta al WebSocket a través de SockJS/STOMP cuando inicia la aplicación. Esto se realiza en el servicio de chat (ChatService), en el método initConnectionSocket.
2. **Establecer Conexión WebSocket**: SockJS establece una conexión con el servidor a través del WebSocket, y el servidor responde con una confirmación de conexión exitosa.
3. **Enviar Mensaje**: Cuando el usuario envía un mensaje a través de la interfaz de usuario, se invoca el método sendMessage en el componente ChatComponent. Este método llama a ChatService.sendMessage para enviar el mensaje a través de STOMP.
4. **Mensaje al Servidor**: El ChatService envía el mensaje al servidor a través del endpoint /app/chat/{roomID}, que es gestionado por el método Chat en el controlador webSocketControlador.
5. **Procesar Mensaje**: El controlador en el servidor recibe el mensaje en la función Chat, lo procesa (en este caso, simplemente lo imprime en consola), y luego lo retransmite a los suscriptores en /topic/{roomID}.
6. **Retransmitir Mensaje**: El mensaje es enviado de vuelta a todos los clientes suscritos a la sala de chat especificada en roomID. Este paso permite que todos los clientes conectados a la misma sala reciban el mensaje.
7. **Recibir Mensaje**: El cliente recibe el mensaje a través del método onMessageReceived del ChatService, que se suscribió previamente al canal /topic/{roomID}.
8. **Mostrar Mensaje**: Finalmente, el mensaje recibido se agrega a la lista de mensajes en el ChatComponent, y se muestra en la interfaz del usuario para que todos los participantes de la sala puedan verlo.

Este diagrama describe cómo el cliente y el servidor interactúan para permitir la comunicación en tiempo real a través de mensajes de chat.

Git Hub

fabriciorioseexe/LoopChat (github.com)