UNIVERSIDAD DE GUAYAQUIL

Chat en Tiempo Real Utilizando Sockets

Proyecto del segundo parcial

Integrantes:

Grizzly Alcivar Zambrano

Pintag Sanga Glen

Jesús Zuña Pacheco

López Moreno Adiel Stalin

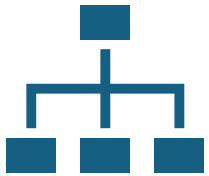
Lindao Herrera Alan

Tecnologías utilizadas en el proyecto

Angular 19

Bun

Tailwind CSS



Angular 19:

Framework frontend utilizado para la construcción de la interfaz de usuario, la gestión de componentes, el enrutamiento de la aplicación y la comunicación con el backend mediante servicios.

Bun:

Entorno de ejecución de JavaScript utilizado para el backend, encargado de implementar el servidor de WebSockets que permite la comunicación en tiempo real entre los clientes conectados.

Tailwind CSS:

Framework de estilos basado en utilidades, utilizado para la maquetación y el diseño visual de las pantallas de login y chat, permitiendo un desarrollo rápido y consistente de la interfaz.

Actividades realizadas durante el desarrollo

1. Creación del proyecto Angular mediante el comando ng new client --package-manager bun , con el objetivo de inicializar la aplicación frontend utilizando Bun como gestor de paquetes.

2. Estructuración del proyecto creando dos carpetas principales (login y chat), destinadas a organizar la lógica y la interfaz del inicio de sesión y del chat respectivamente.

3. Creación de los componentes principales login.component.ts y chat.component.ts , encargados de representar las vistas principales de la aplicación.

4. Configuración del enrutamiento de la aplicación Angular, permitiendo la navegación entre el login y el chat mediante rutas definidas en el archivo de rutas, utilizando carga dinámica de componentes.

5. Instalación y configuración de Tailwind CSS, con el fin de aplicar estilos modernos y responsivos a la interfaz del proyecto de manera eficiente

6. Maquetación de la interfaz de usuario, diseñando las vistas de login y chat, incluyendo campos de entrada, botones, encabezados y contenedores para los mensajes.

Actividades realizadas durante el desarrollo

7. Creación del componente message.component.ts , utilizado para representar visualmente cada mensaje del chat y diferenciar los mensajes propios de los recibidos de otros usuarios.

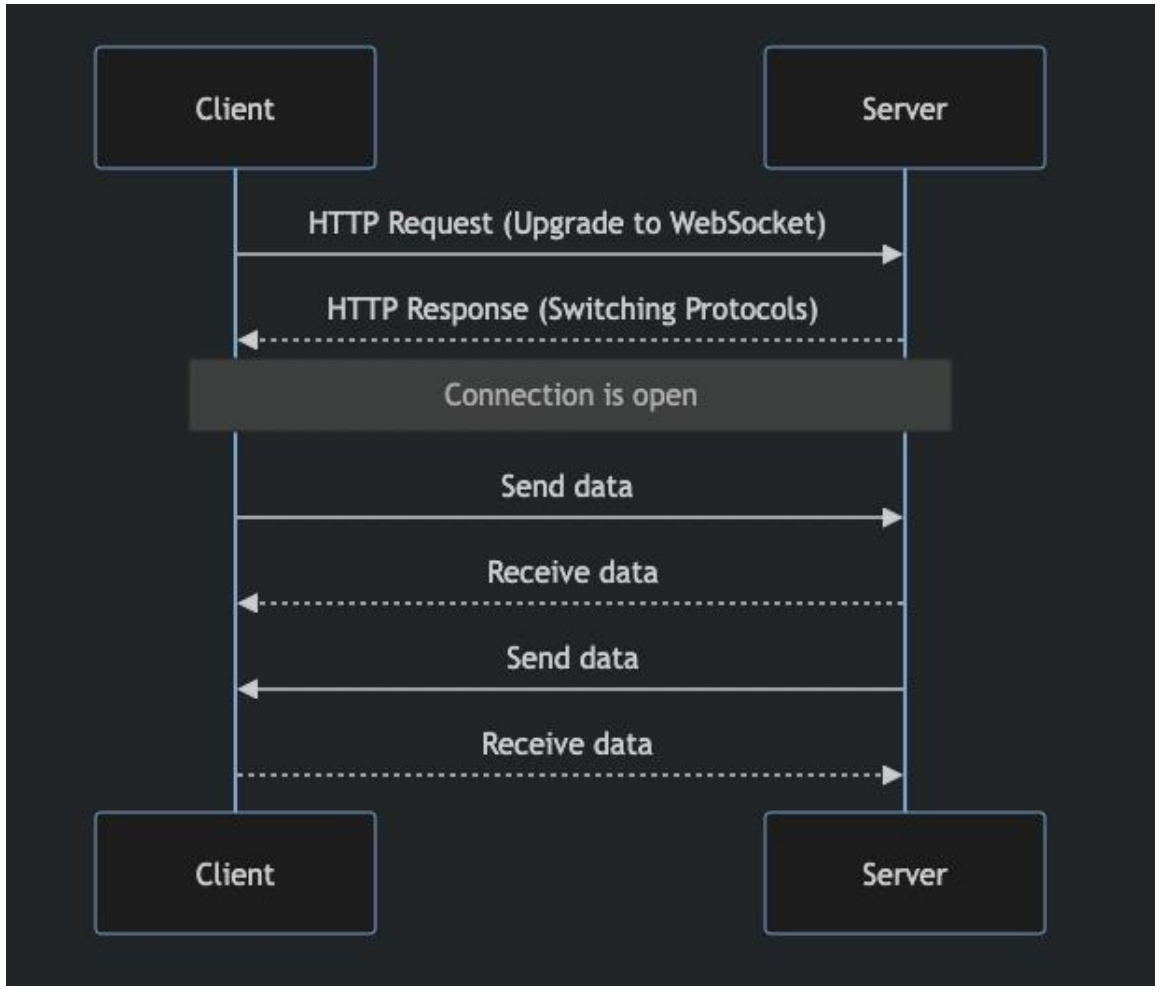
8. Se creó el proyecto backend utilizando Bun mediante el comando bun init , sobre el cual se implementó el servidor WebSocket en el archivo index.ts , encargado de gestionar las conexiones, desconexiones y la transmisión de mensajes en tiempo real entre los clientes conectados.

9. Desarrollo del servicio websocket.service.ts en Angular, encargado de manejar la conexión con el servidor WebSocket, enviar mensajes, recibir eventos y mantener el estado de la comunicación.

10. Finalmente, se realizó la integración completa entre el frontend y el backend para permitir la comunicación en tiempo real entre los usuarios del chat.

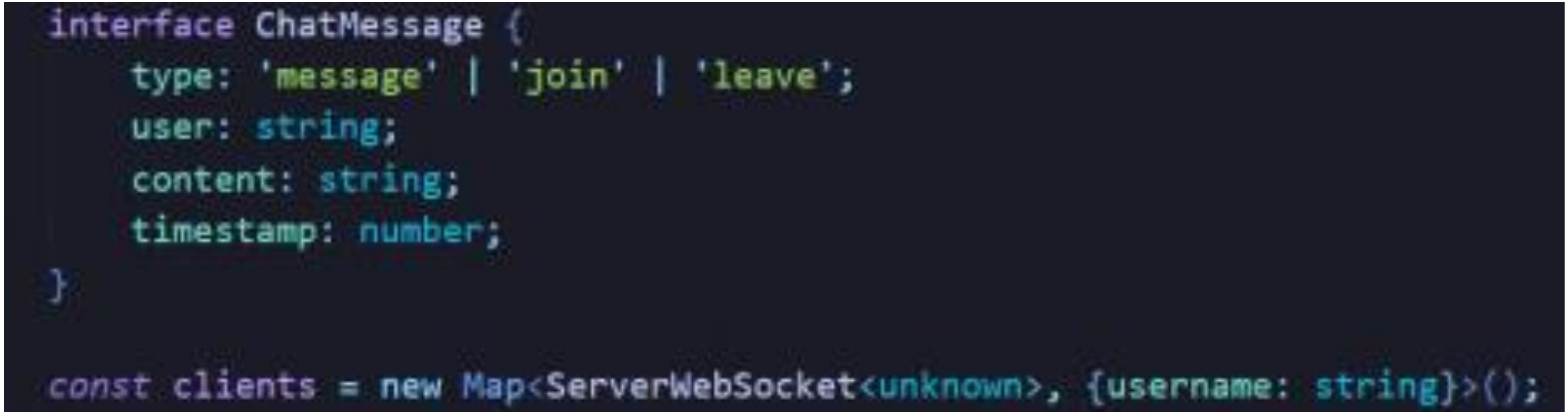
Además, se implementó una persistencia básica de datos mediante localStorage para conservar el nombre de usuario y el historial de mensajes.

así como la funcionalidad de cierre de sesión (logout), encargada de eliminar la información almacenada y redirigir al usuario nuevamente a la pantalla de inicio de sesión.



Explicación del archivo index.ts (Servidor WebSocket con Bun)

El archivo index.ts constituye el núcleo del backend, ya que gestiona las conexiones WebSocket, el envío y recepción de mensajes, y la comunicación en tiempo real entre los usuarios del chat, utilizando Bun como entorno de ejecución



1. Definición del formato de los mensajes y gestión de clientes

En este bloque se define la estructura estándar de los mensajes que se intercambian entre cliente y servidor, así como el almacenamiento de las conexiones activas mediante un Map . Esto permite identificar a cada usuario conectado y gestionar múltiples clientes de forma simultánea.

```
const sendMessageToClients = (message: ChatMessage) => {
    clients.forEach((_, client) => {
        client.send(JSON.stringify(message));
    })
}
```

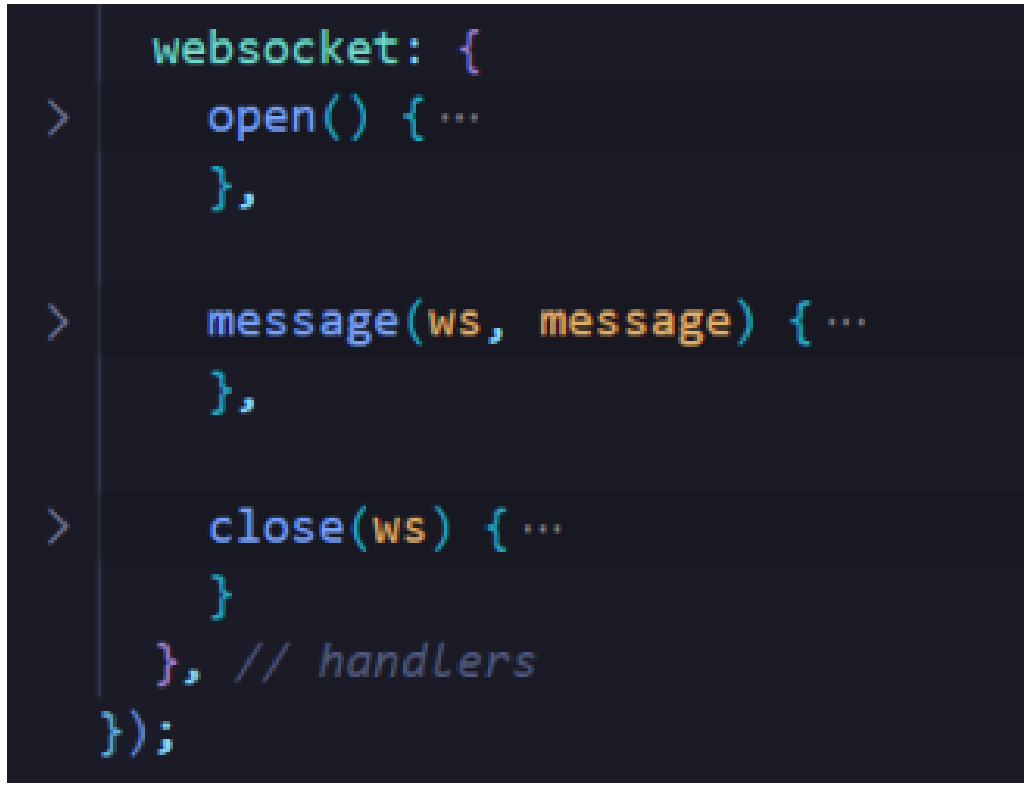
2. Envío de mensajes a todos los clientes conectados (broadcast)

Esta función centraliza el envío de mensajes en tiempo real a todos los clientes conectados, utilizando el método send() propio de los WebSockets. Se emplea para notificar eventos como la unión de usuarios, mensajes enviados y desconexiones

```
Bun.serve({
  fetch(req, server) {
    // upgrade the request to a WebSocket
    if (server.upgrade(req)) {
      return; // do not return a Response
    }
    return new Response("Upgrade failed", { status: 500 });
  },
```

3. Inicialización y actualización de la conexión a WebSocket

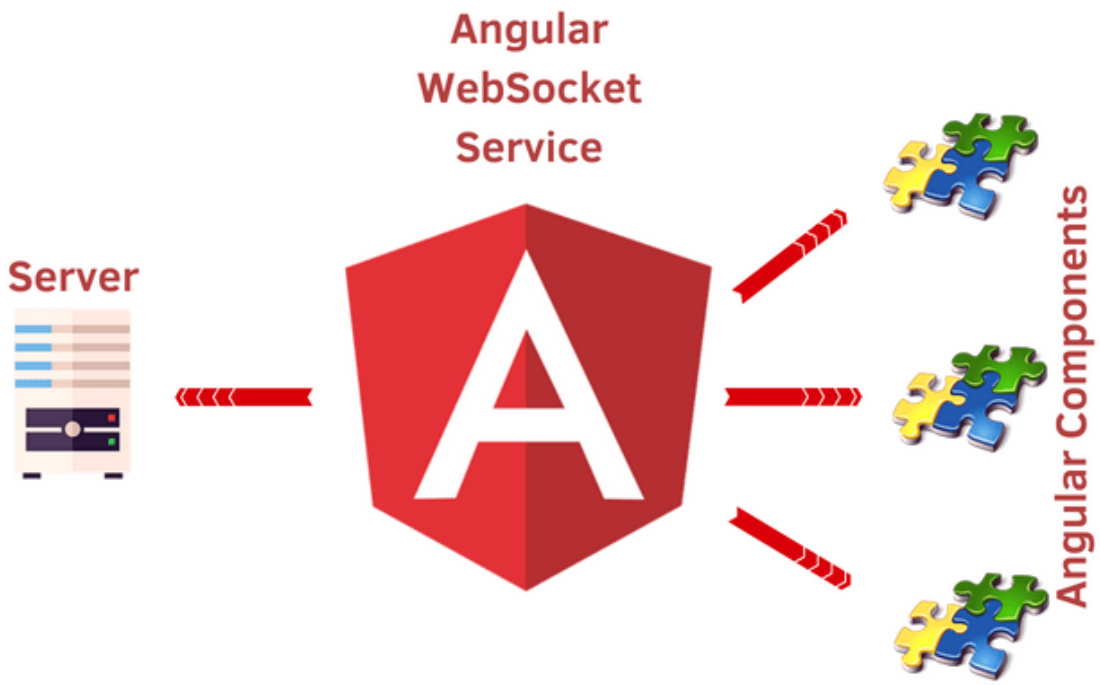
- En este bloque se inicializa el servidor con Bun.serve() y se actualiza la conexión HTTP a WebSocket, permitiendo establecer una conexión persistente entre cliente y servidor.



4. Manejo de eventos WebSocket (open , message , close)

- Este bloque gestiona el ciclo de vida del WebSocket:
- open : registra nuevas conexiones.
- message : procesa mensajes entrantes (join y message) y los reenvía a todos los clientes.
- close : notifica la salida de un usuario y elimina su conexión.

Aquí se implementa la lógica principal del chat en tiempo real, permitiendo la interacción entre múltiples usuarios conectados simultáneamente.



Explicación del archivo websocket.service.ts – Servicio WebSocket en Angular

El archivo websocket.service.ts actúa como el intermediario principal entre Angular y el servidor WebSocket, gestionando la conexión, el intercambio de mensajes en tiempo real, la persistencia de la sesión y la actualización reactiva de la interfaz del chat.

```ts
export interface ChatMessage {
  type: "message" | "join" | "leave";
  user: string;
  content?: string;
  timestamp: number;
}
```

```ts
  username = signal<string>('');
  messages = signal<ChatMessage[]>([]);
```

1. Definición del contrato del mensaje y estado global

- Este bloque define la estructura de los mensajes intercambiados con el servidor y el estado global de la aplicación, utilizando Signal para almacenar el usuario actual y el historial de mensajes.

```
constructor() {
  this.loadSession();
}

private loadSession() {
  const savedUsername = localStorage.getItem('username');
  if (savedUsername) {
    this.connect(savedUsername);
    // obtener los mensajes
    this.loadChatMessages();
  } else {
    // redirigir al login
    this.router.navigate(['/']);
  }
}
```

2. Inicialización del servicio y carga de sesión

- Aquí se implementa la persistencia de sesión, permitiendo: reconectar automáticamente al usuario si existe información guardada, redirigir al login si no hay sesión activa.

```
this.socket = new WebSocket('ws://localhost:3000'); // 3000 por defecto en Bun

this.socket.onopen = () => {
  // mandar mensaje de que se unio alguien
  this.joinChat();
};
```

3. Conexión al servidor WebSocket

- Este bloque establece la conexión WebSocket con el servidor en Bun y envía automáticamente un mensaje de tipo join al abrirse la conexión.

```
this.socket.onmessage = (event) => {
  const message = JSON.parse(event.data) as ChatMessage;

  this.messages.update((oldMessages) => {
    const messages = [...oldMessages, message];
    localStorage.setItem('messages', JSON.stringify(messages));
    return messages;
  });
};
```

4. Recepción de mensajes y actualización del estado

- Este bloque gestiona los mensajes entrantes desde el servidor, ya que actualiza el estado reactivo, persiste el historial en localStorage y permite la actualización automática de la interfaz.

```
sendChatMessage(content: string) {
  const message: ChatMessage = {
    type: 'message',
    user: this.username(),
    content,
    timestamp: Date.now(),
  };
  this.sendMessage(message);
}

private sendMessage(message: ChatMessage) {
  if (this.socket && this.socket.readyState === WebSocket.OPEN) {
    this.socket.send(JSON.stringify(message));
  }
}
```

5. Envío de mensajes al servidor

- Aquí se construyen y envían los mensajes del usuario al servidor WebSocket, verificando previamente que la conexión esté activa.

```
logOut() {
  if (this.socket) {
    this.socket.close();
    this.username.set('');
    this.router.navigateByUrl('/');
    this.messages.set([]);
    localStorage.removeItem('username');
    localStorage.removeItem('messages');
  }
}
```

6. Envío de mensajes al servidor

- Este bloque gestiona el **cierre de sesión**, cerrando la conexión WebSocket, limpiando el estado local y redirigiendo al usuario al login.

Explicación del archivo login.component.ts – Componente de inicio de sesión

El archivo login.component.ts cumple la función de puerta de entrada a la aplicación, capturando el nombre del usuario, iniciando la conexión con el servidor WebSocket y controlando la transición hacia la vista principal del chat.

1. Definición del componente y formulario reactivo

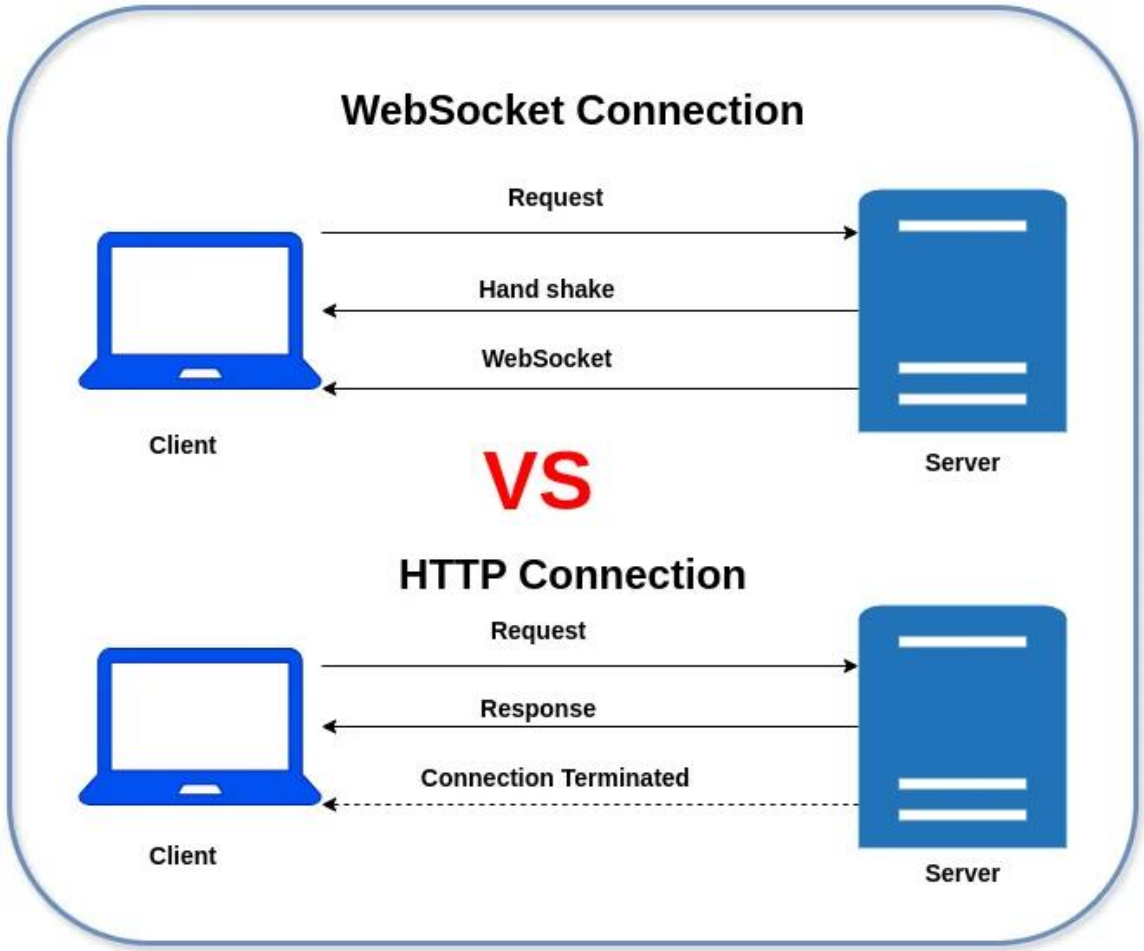
Se define el componente de login utilizando formularios reactivos, los cuales permiten capturar y validar el nombre del usuario

2. Inyección de dependencias y servicios :

Aquí se inyectan el servicio WebSocket y el Router, permitiendo: iniciar la conexión con el servidor, controlar la navegación entre vistas.

3. Inicio de sesión y navegación al chat

Se establece la conexión WebSocket mediante el servicio y redirige al usuario a la vista del chat.



Explicación del archivo chat.component.ts – Componente principal del chat

El archivo chat.component.ts integra la interfaz del chat con la lógica de comunicación en tiempo real, permitiendo la visualización dinámica de mensajes, el envío de información al servidor WebSocket y la gestión de la sesión del usuario de forma eficiente.

1. Definición del componente e integración de dependencias

- Se define el componente del chat y se importan los módulos necesarios para su funcionamiento. Se incluye MessageComponent para delegar la representación visual de cada mensaje y ReactiveFormsModule para gestionar la entrada de texto del usuario de forma controlada.

2. Inyección del servicio WebSocket y estado compartido

- Se inyecta el WebsocketService , el cual actúa como fuente central de datos en tiempo real. A través de este servicio, el componente accede: al listado de mensajes del chat, al nombre del usuario autenticado. El uso de Signal permite que la interfaz se actualice automáticamente cuando llegan nuevos mensajes desde el servidor.

3. Renderizado dinámico de mensajes en tiempo real

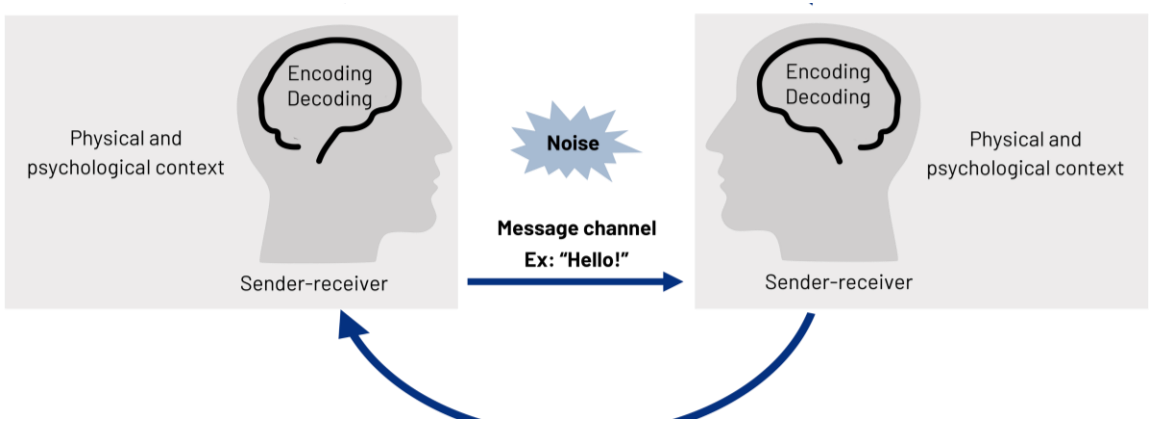
- Se encarga de renderizar dinámicamente los mensajes recibidos. Cada mensaje es enviado al componente MessageComponent , donde se determina si el mensaje pertenece al usuario actual, permitiendo una diferenciación visual clara dentro del chat.

4. Envío de mensajes al servidor WebSocket

- Se captura el mensaje escrito por el usuario, valida su contenido y lo envía al servidor mediante el servicio WebSocket. Posteriormente, se limpia el campo de texto para permitir el envío de nuevos mensajes.

5. Cierre de sesión y finalización de la conexión

- Permite al usuario cerrar su sesión, lo que implica: cerrar la conexión WebSocket, limpiar los datos almacenados, regresar a la pantalla de login.



Explicación del archivo message.component.ts – Componente de visualización de mensajes

El archivo message.component.ts implementa un componente reutilizable encargado de la presentación visual de cada mensaje dentro del chat, separando la lógica de diseño de la lógica principal del componente chat .

1. Definición del componente y propiedades de entrada

- Este bloque define el componente y declara las propiedades de entrada que recibe desde chat.component.ts :
  - message : contiene la información del mensaje (usuario y contenido).
  - myMessage : indica si el mensaje pertenece al usuario actual.
- Esto permite que el componente sea totalmente reutilizable y desacoplado de la lógica del chat.

```
<div class="flex items-start gap-4 w-full"
  [class.flex-row-reverse]="myMessage()">
```

```
[ngClass]="myMessage() ? 'bg-blue-100': 'bg-gray-200'">
```

2. Diferenciación visual entre mensajes propios y ajenos

- Este bloque aplica clases dinámicas según el tipo de mensaje:
  - los mensajes propios se alinean a la derecha,
  - los mensajes de otros usuarios se alinean a la izquierda.

```
{{ message().user.charAt(0).toUpperCase() }}
```

```
@if (!myMessage()) {
  <p class="text-xs text-blue-600 mb-1">
    {{ message().user }}
  </p>
}
<p>{{ message().content }}</p>
```

3. Representación del usuario y contenido del mensaje

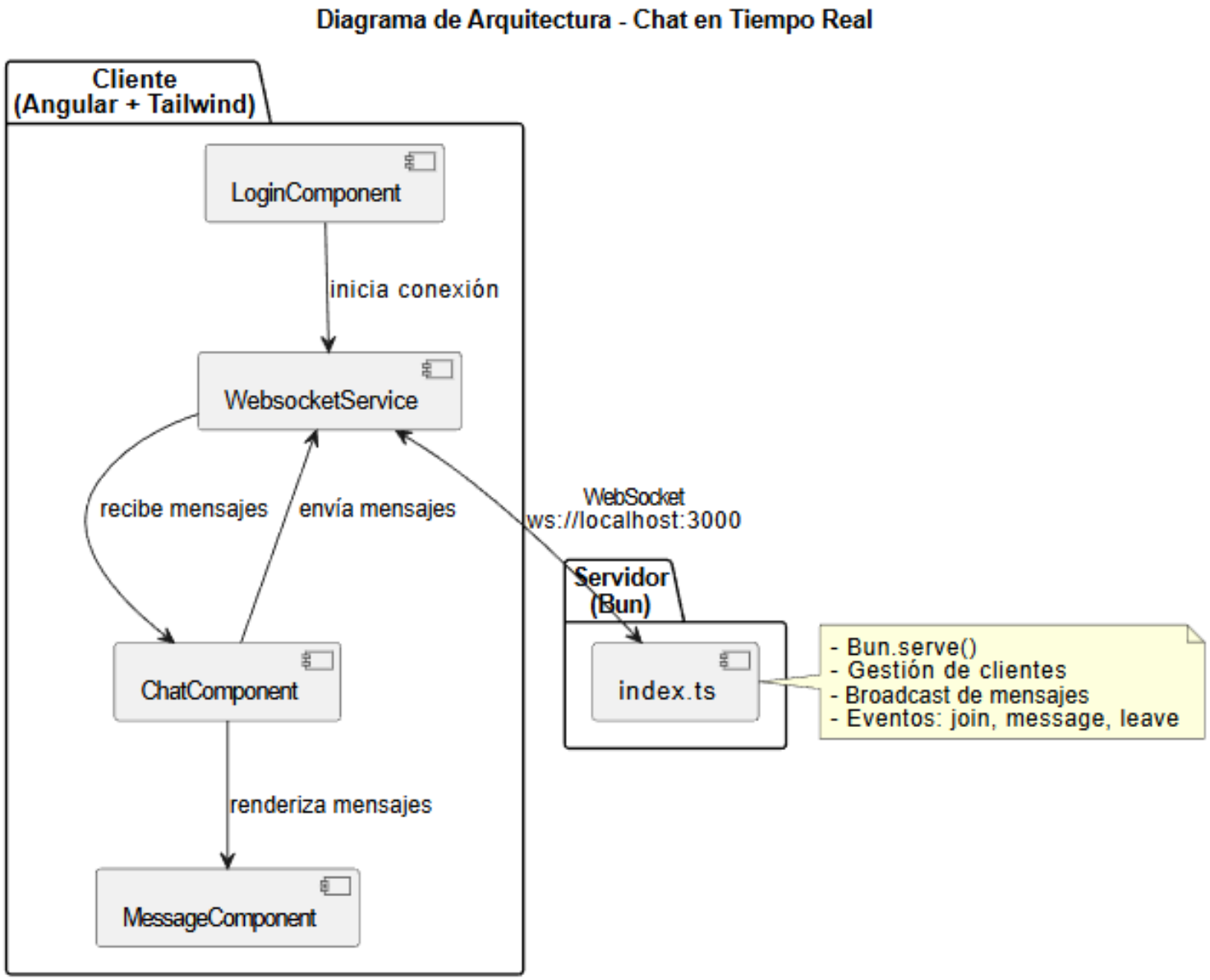
- Aquí se muestra:
  - la inicial del usuario como avatar,
  - el nombre del remitente (solo cuando no es el usuario actual),
  - el contenido del mensaje.
- Este enfoque mejora la claridad y legibilidad del chat, especialmente cuando hay múltiples usuarios conectados.

Conclusión

- Como conclusión, el proyecto desarrollado permitió implementar un chat en tiempo real funcional, integrando un backend basado en WebSockets con Bun y un frontend construido con Angular, logrando una comunicación bidireccional eficiente entre múltiples usuarios.
- La correcta separación de responsabilidades entre servicios, componentes y presentación facilitó un flujo claro desde el inicio de sesión hasta el intercambio dinámico de mensajes.
- A continuación, se procederá a realizar la muestra práctica del funcionamiento del chat, donde se evidenciará la conexión de usuarios, el envío y recepción de mensajes en tiempo real y la gestión de la sesión dentro de la aplicación.

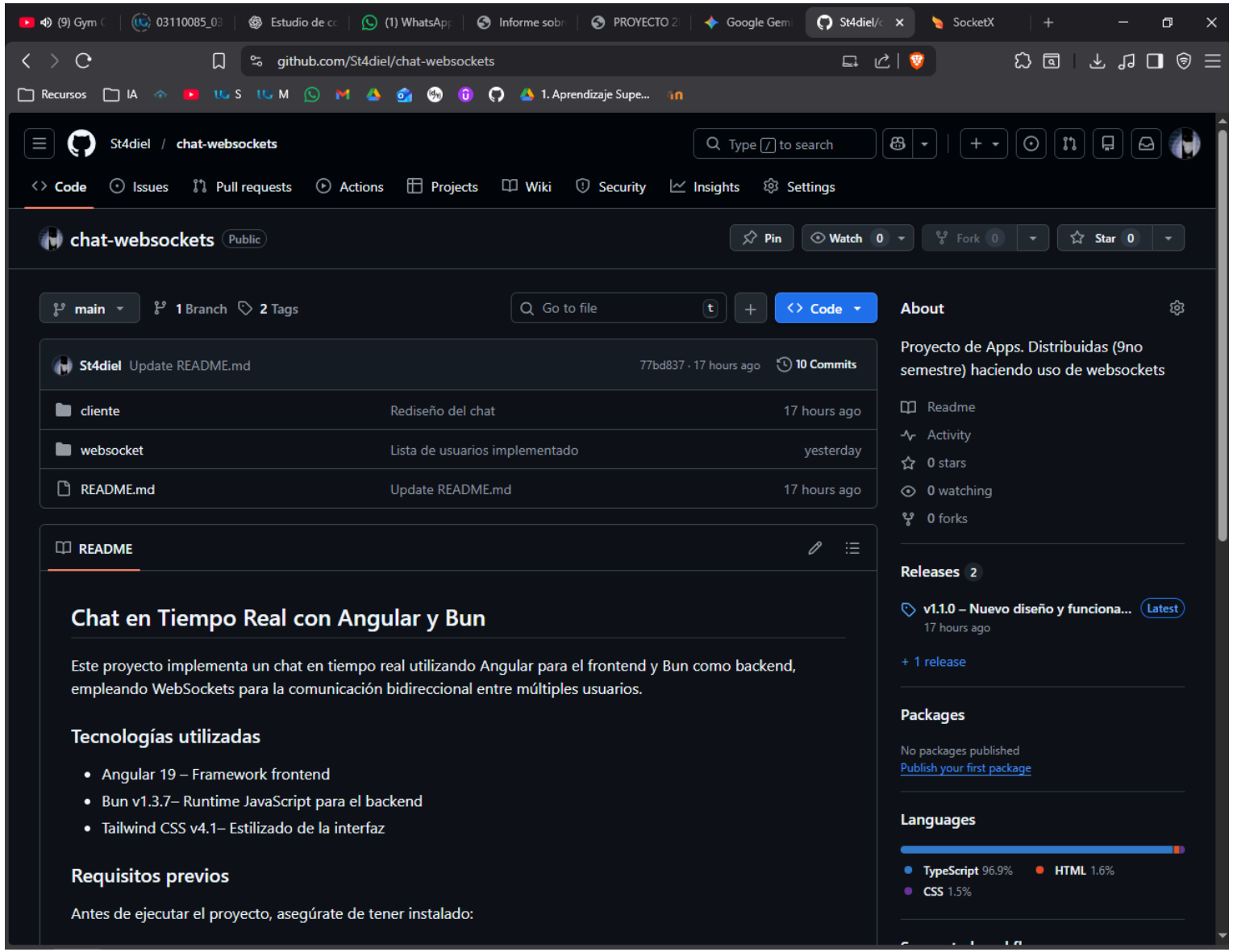
Diagrama de arquitectura

Anexos



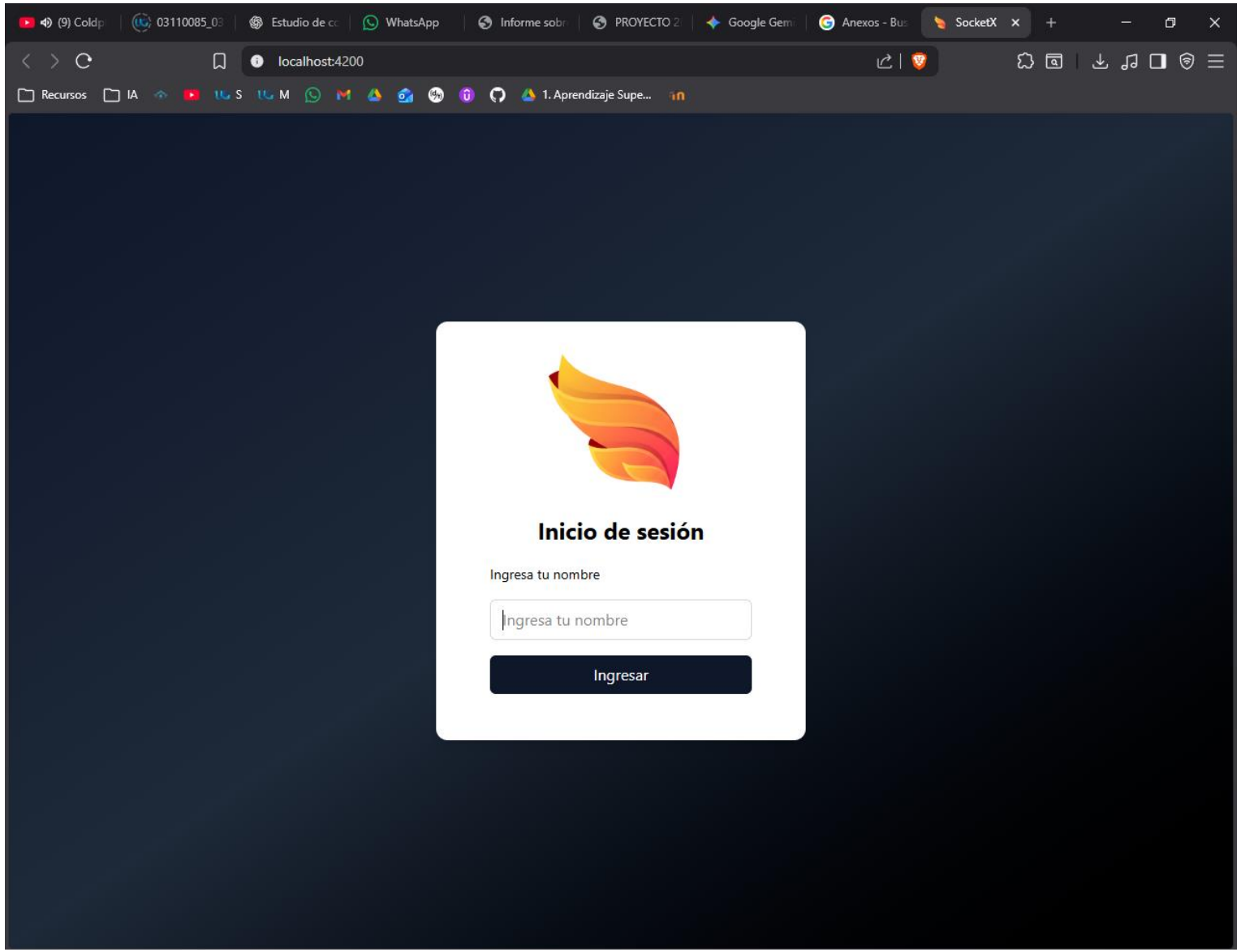
Repositorio de GitHub:
https://github.com/St4diel/chat-websockets

Anexos



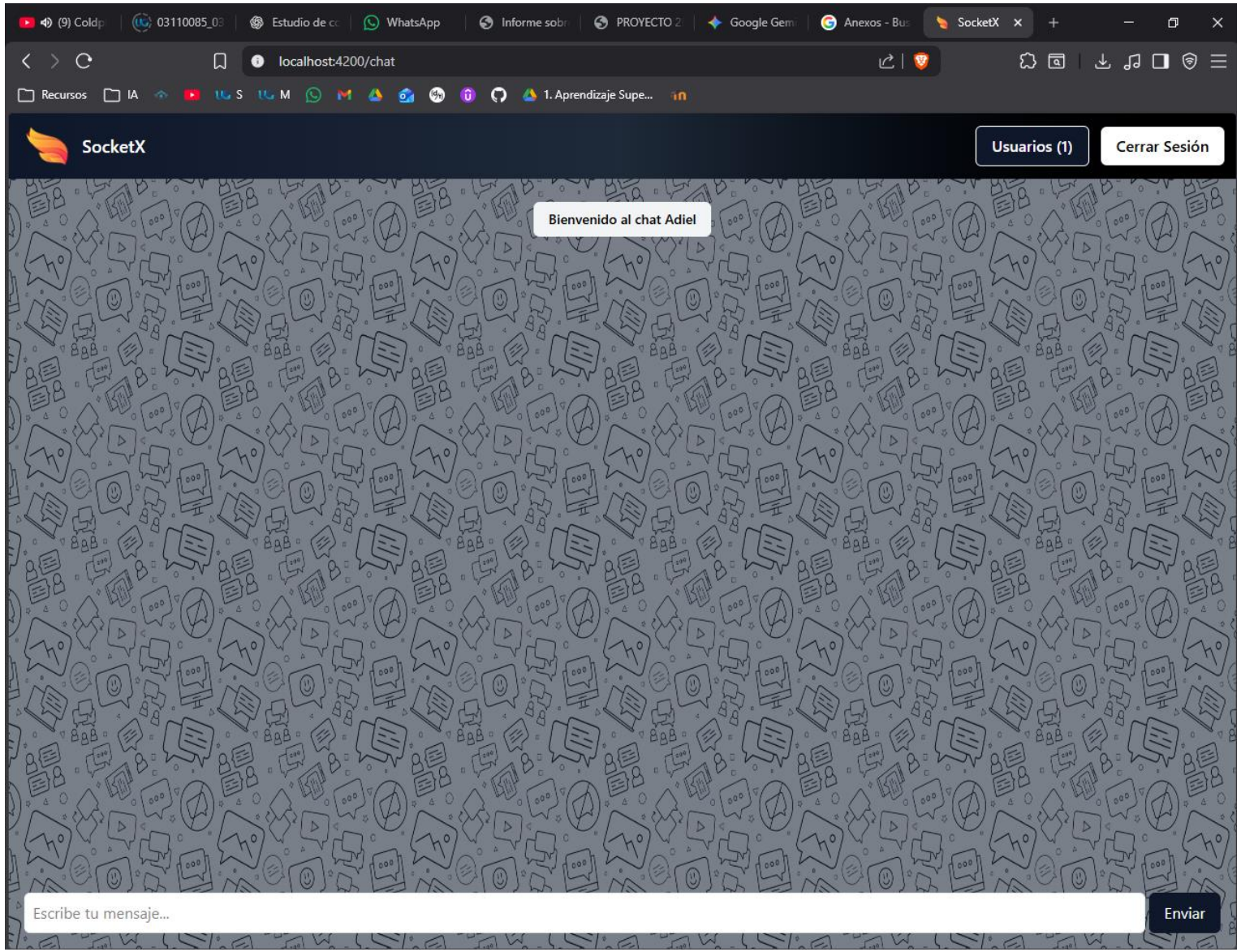
Login

Anexos



Chat principal

Anexos



Lista de usuarios

Anexos