Instituto Tecnológico

y de Estudios Superiores de Occidente

Programación de Aplicaciones de Escritorio

“Documentación”

Integrantes:

José Andrés Cota Samaue - 733387

Ivanna Joselyn Haro Zuno – 727788

David Koch Torres Septien

Profesor: Francisco Javier Sevilla Medina

Viernes 17 de noviembre de 2023

Descripción:

TuneHub será una aplicación web que combina características de reproducción de música, administración de listas de reproducción, calendario de eventos, y la posibilidad de establecer comunicación limitada con artistas, todo en un solo entorno digital de música que tendrá las siguientes características:

- Reproducción de música: Los usuarios tendrán acceso a una biblioteca de música en línea, permitiéndoles explorar y disfrutar de una amplia variedad de géneros musicales.
- Listas de reproducción: Los usuarios podrán crear, personalizar y gestionar sus propias listas de reproducción, lo que les permitirá organizar sus canciones favoritas de acuerdo a sus preferencias.
- Calendario de eventos: La aplicación ofrecerá información detallada sobre eventos musicales, conciertos y festivales. Los usuarios podrán buscar eventos de su interés, ver detalles como fechas, lugares y artistas participantes, y agregar estos eventos directamente a su calendario de Google para mantenerse informados y no perderse ningún espectáculo.
- Comunicación con artistas: Los usuarios tendrán la oportunidad de establecer contacto limitado con sus artistas favoritos a través de un chat. Esta función permitirá a los usuarios interactuar con los artistas en un entorno exclusivo y controlado, lo que fomentará la comunidad y la conexión entre fanáticos y músicos.

Tecnologías:

Front: Angular, Handlebars, CSS, Bootstrap

Back: Node.js, Express, JavaScript

Base de datos: MongoDB

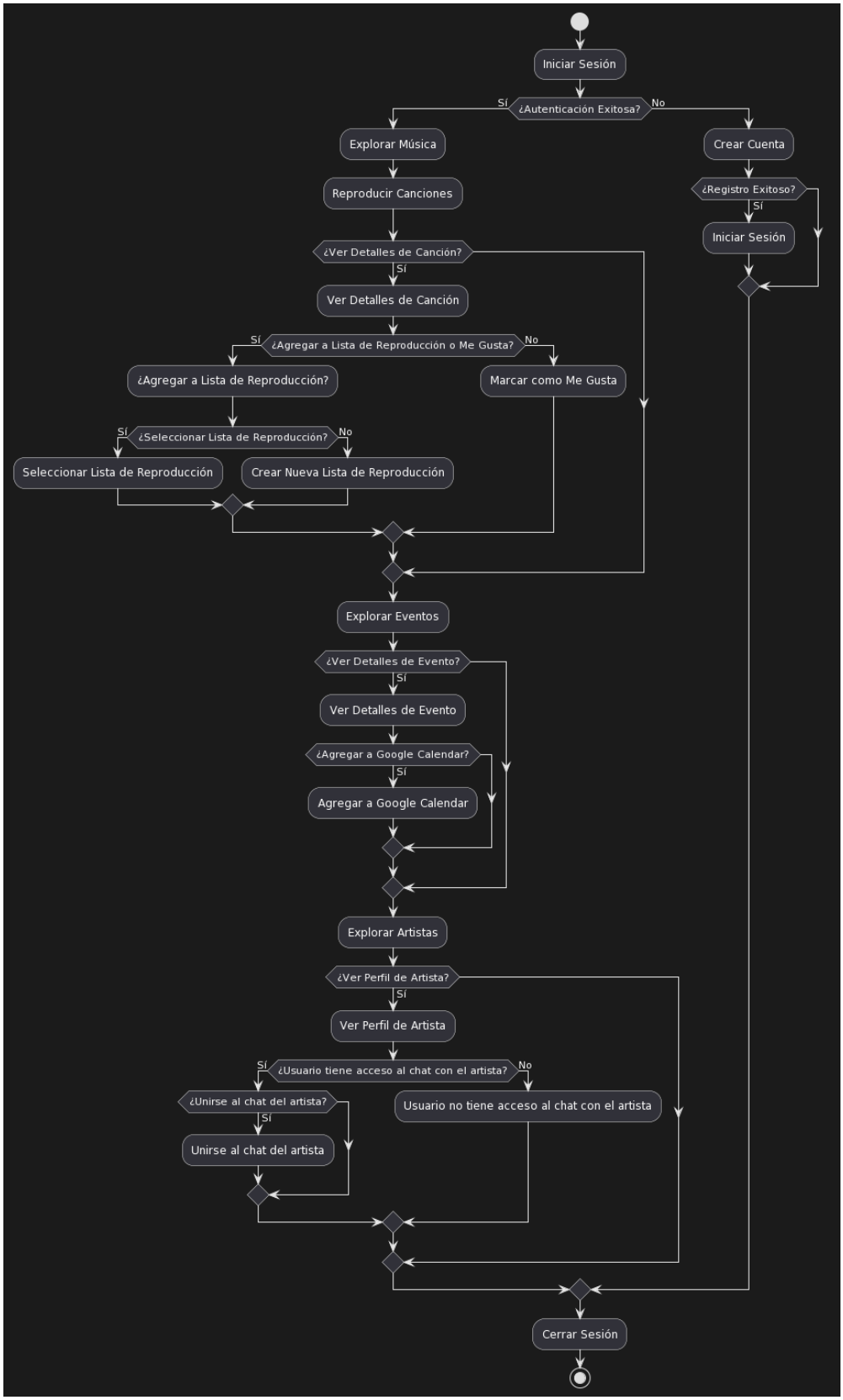
API’s externas: Soundcloud, Genius, GoogleCalendar

Alcance:

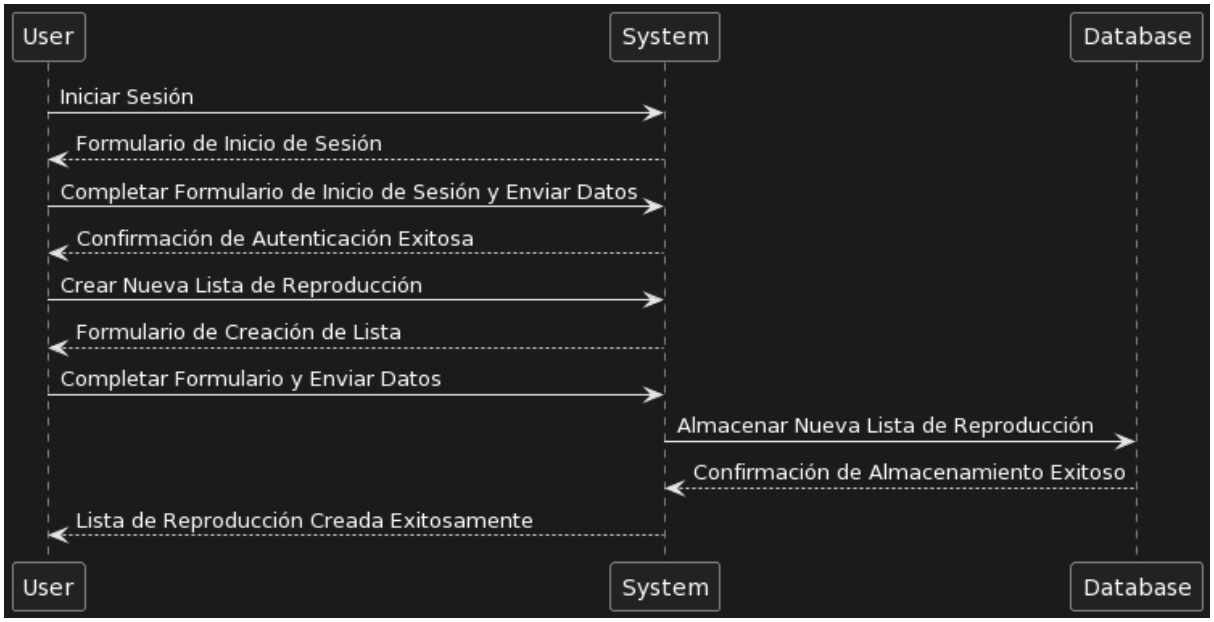
- Usuarios: Pueden crear una sesión con su correo y contraseña. Autenticación con cuentas de Gmail.
- Canciones: Se pueden subir canciones con formato mp3 a través de multer o puede ser importadas de SoundCloud vía su API.
- Eventos: Se pueden crear eventos (conciertos, meet and greets, etc.) del lado del artista y del lado del usuario lo pueden agregar a su google calendar.
- Chats: Los artistas se pueden comunicar con sus fans a través de chats con acceso limitado.
- Tags: Las canciones pueden tener tags para ayudar a determinar la categoría, filtrar y facilitar la búsqueda.
- Playlist: Los usuarios pueden crear playlists con canciones de artistas. Pueden modificarlas, filtrarlas y compartirlas con otros usuarios

Diagramas

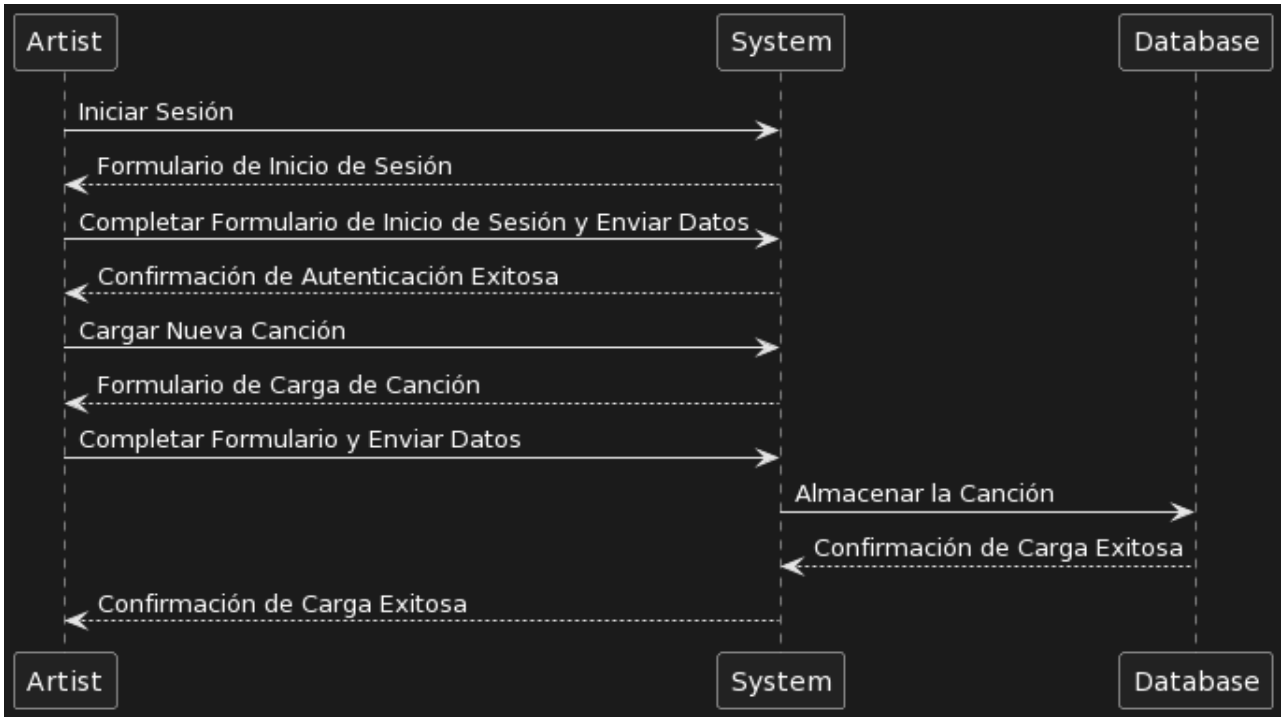
- Diagrama general de la aplicación



- Crear playlist por un usuario



- Carga de Nueva Canción por un Artista



Describir las entidades que se van a administrar dentro de la aplicación y las acciones que se realizarán sobre cada una de estas.

Usuario:

- Registro como cliente
- Iniciar sesión

- Ver y editar perfil
- Crear playlist
- Reproducir música
- Unirse a chats de sus músicos
- Explorar música
- Explorar eventos

Artista:

- Registro como artista: Los artistas pueden registrarse como tal, proporcionando información sobre su música y perfil
- Iniciar sesión
- Ver y editar perfil
- Subir canciones o ligar canciones de SoundCloud
- Crear chat con fans
- Recibir y mandar mensajes con fans
- Publicar eventos
- Poner tag a canciones

Acciones para el Usuario Normal:

- Explorar y escuchar música.
- Agregar canciones a listas de reproducción personales.
- Buscar y agregar eventos a Google Calendar.
- Iniciar y responder a chats con artistas.
- Ver y editar su propio perfil.

Acciones para el Artista:

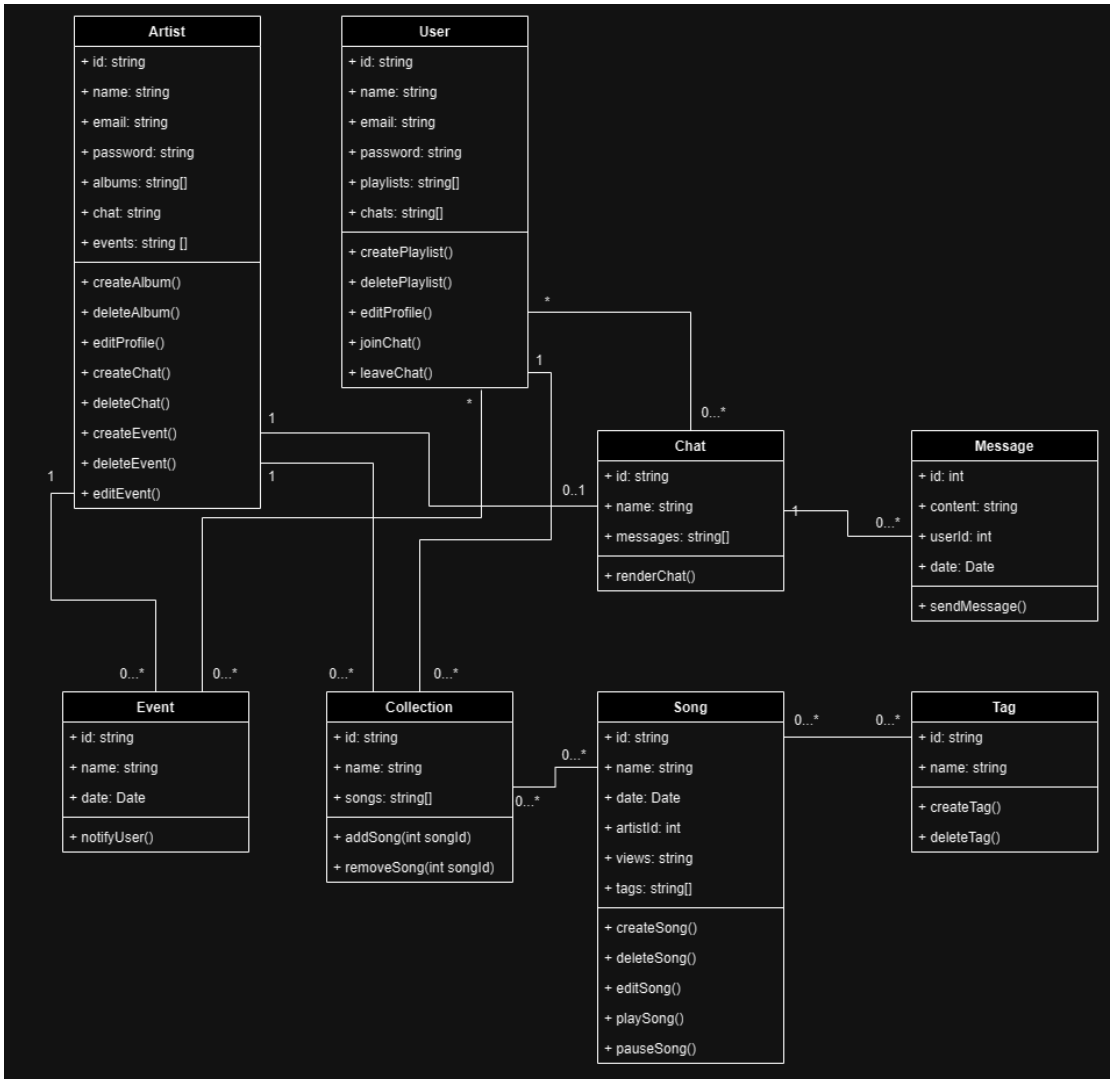
- Subir, administrar y eliminar su propia música.
- Crear, editar y eliminar eventos musicales.
- Iniciar y responder a chats con fans.
- Ver y editar su propio perfil de artista.

Planeación (mostrar un diagrama de gantt con los rangos de fechas destinados para el desarrollo de las diversas etapas del proyecto incluyendo:

Tarea	Plazo	Semanas

		1	2	3	4	5	6	7	8	9	10
Setup	1 semana										
Documentación	1 semana										
Autenticación	1 semana										
Registro de usuarios	1 semana										
Diversos CRUDs -Playlist -Tags -Canciones	2 semanas										
-Chats -Eventos -Relación eventos y chats con artistas	2 semanas										
Pruebas	1 semana										
Despliegue	1 semana										

Diagrama UML Base de datos



Acciones y permisos

Colecciones:

Crear colecciones- artistas y usuarios

Agregar canciones- artistas y usuarios

Eliminar colecciones – artistas y usuarios

Ver colecciones- artistas y usuarios

Canciones:

Subir una canción- artistas

Modificar datos de la canción- artistas

Eliminar canciones– artistas

Ver canciones- artistas y usuarios

Chats y mensajes:

Crear chats- artistas

Entrar a chats- usuarios

Mandar y ver mensajes – artistas y usuarios

Borrar un chat- artista

Entrar y salir de chats - usuarios

Eventos:

Crear un evento- artista

Borrar un evento – artista

Editar un evento – artista

Ver eventos – artistas y usuarios

Agregar Tags

Agregar tags- artistas

Borrar tags- artistas

Artistas

Crear artista – artista

Editar perfil de artista – artista

Ver artistas – artistas y usuarios

Borrar cuenta - artista

Usuarios

Crear usuario– usuario

Editar perfil de usuario – usuario

Ver usuarios – artistas y usuarios

Borrar cuenta - usuario

Diagramas de autenticación

Diagrama de secuencia para registro

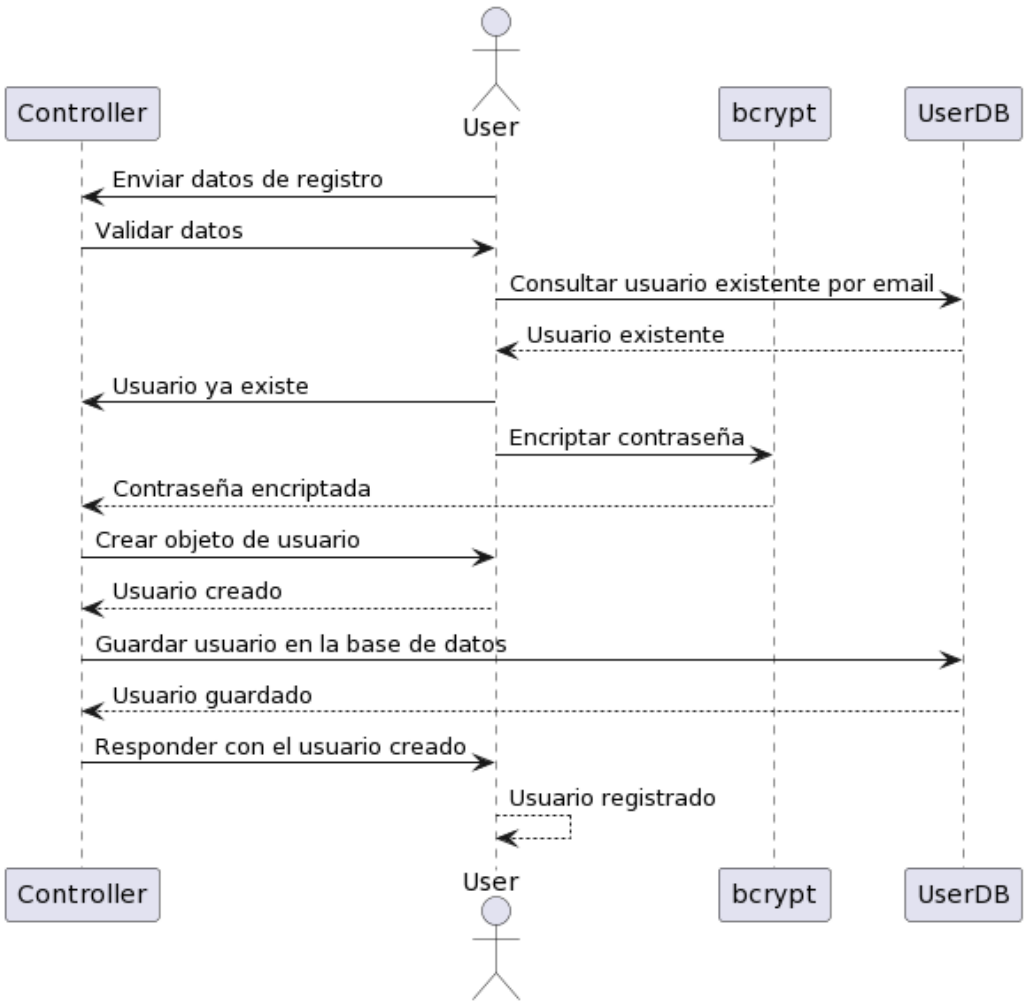


Diagrama de secuencia de inicio de sesión

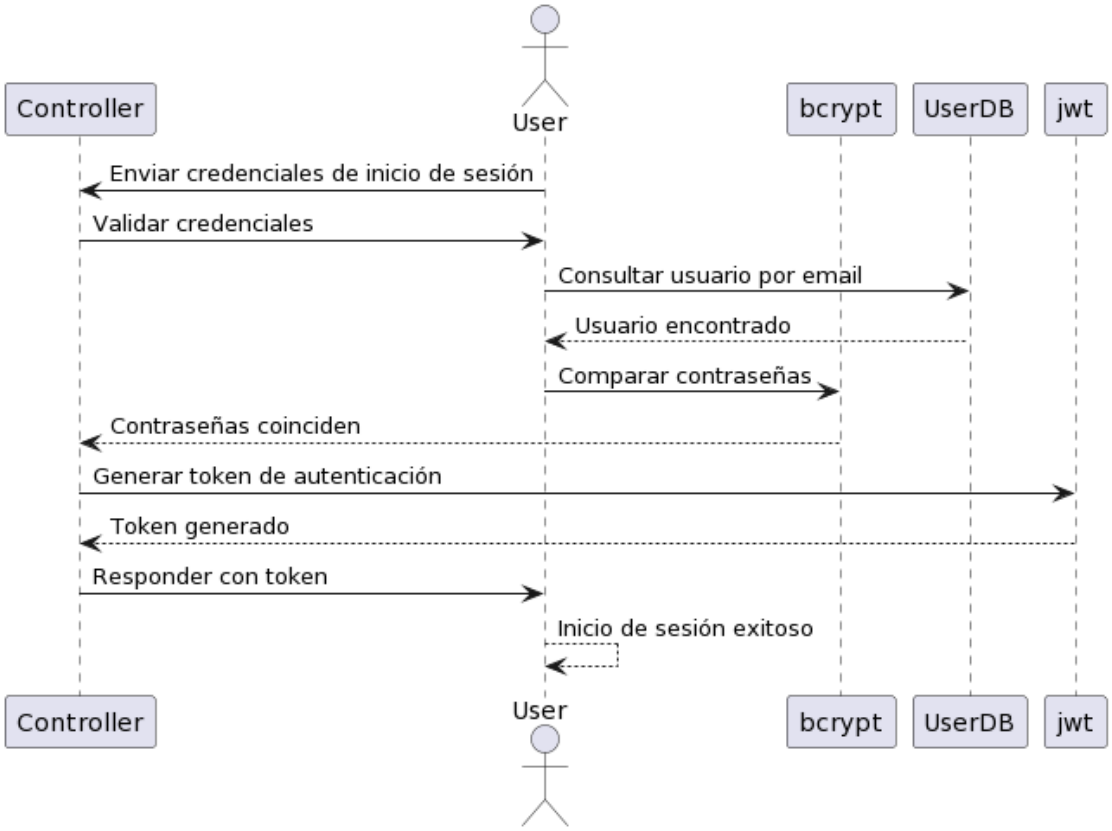


Diagrama de secuencia middleware de autenticación

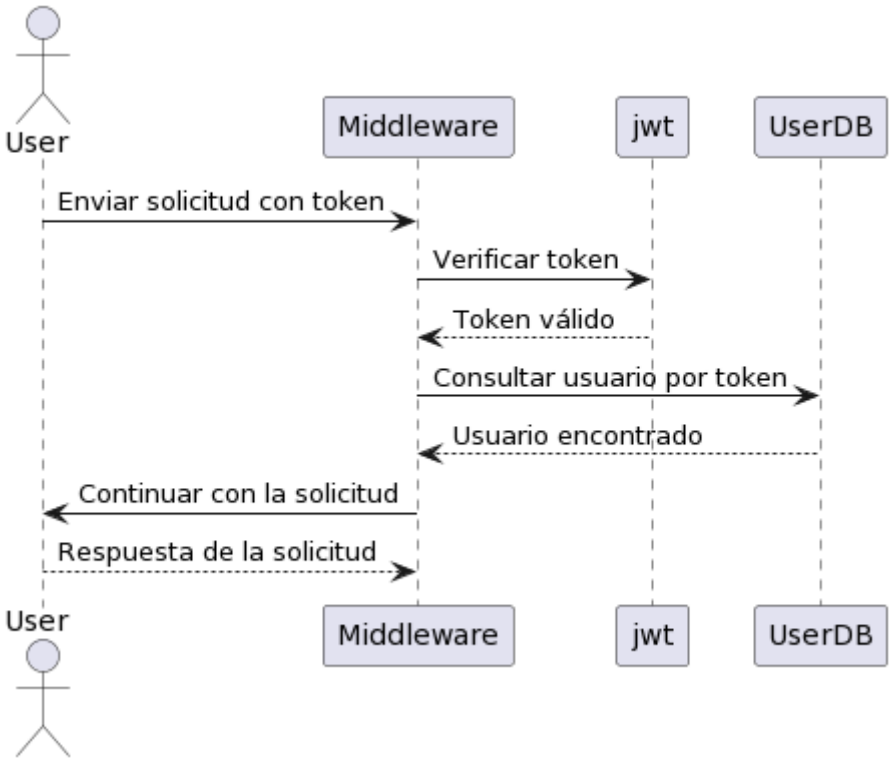
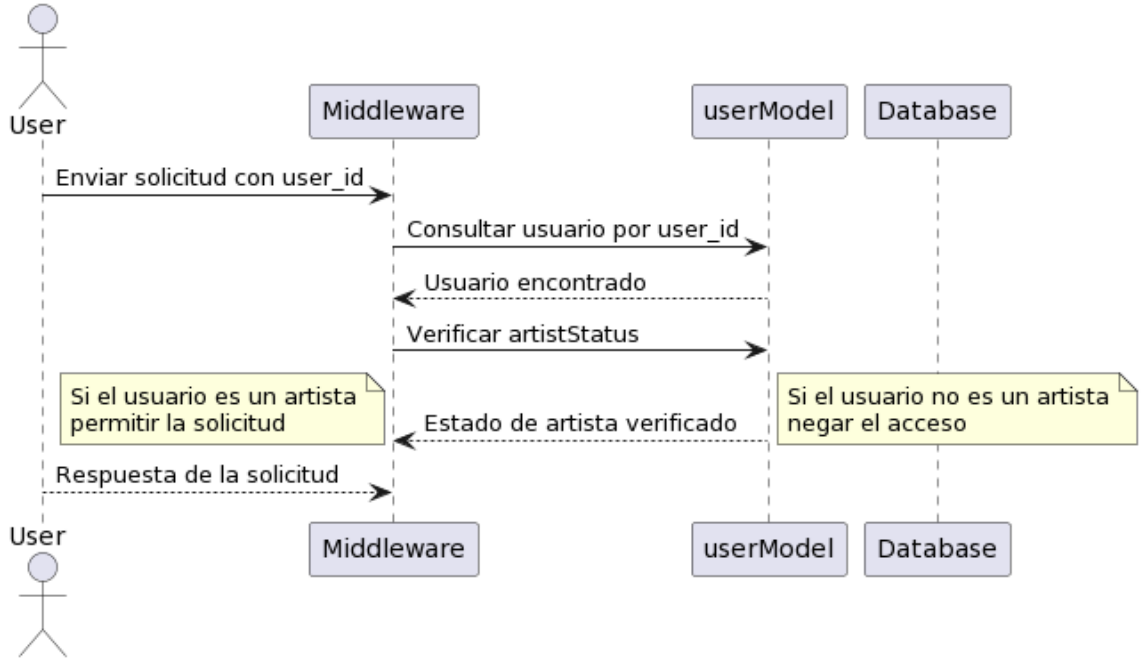


Diagrama de secuencia middleware de roles



Bloques de codigo

Controlador para login

```
    login: (req, res) => {
        const { email, password } = req.body;

        User.findOne({ email })
            .then(user => {
                if (!user) {
                    return res.status(401).json({ message: 'Usuario o contraseña incorrectos' });
                }

                bcrypt.compare(password, user.password)
                    .then(passwordMatch => {
                        if (!passwordMatch) {
                            return res.status(401).json({ message: 'Usuario o contraseña incorrectos' });
                        }
                        const payload = {
                            username: user.email,
                            role: user.artistStatus  // toma en cuenta si el usuario es un artista o no para despues determinar que permisos tiene
                        };


                        const token = jwt.sign(payload, process.env.SECRET, { expiresIn: '1h' });
                        res.json({ message: 'Inicio de sesión exitoso', token });
                    })
                    .catch(error => {
                        res.status(500).json({ message: 'Error en el servidor' });
                    });
            })
            .catch(error => {
                res.status(500).json({ message: 'Error en el servidor' });
            });
    },
};
```

Controlador para crear artista

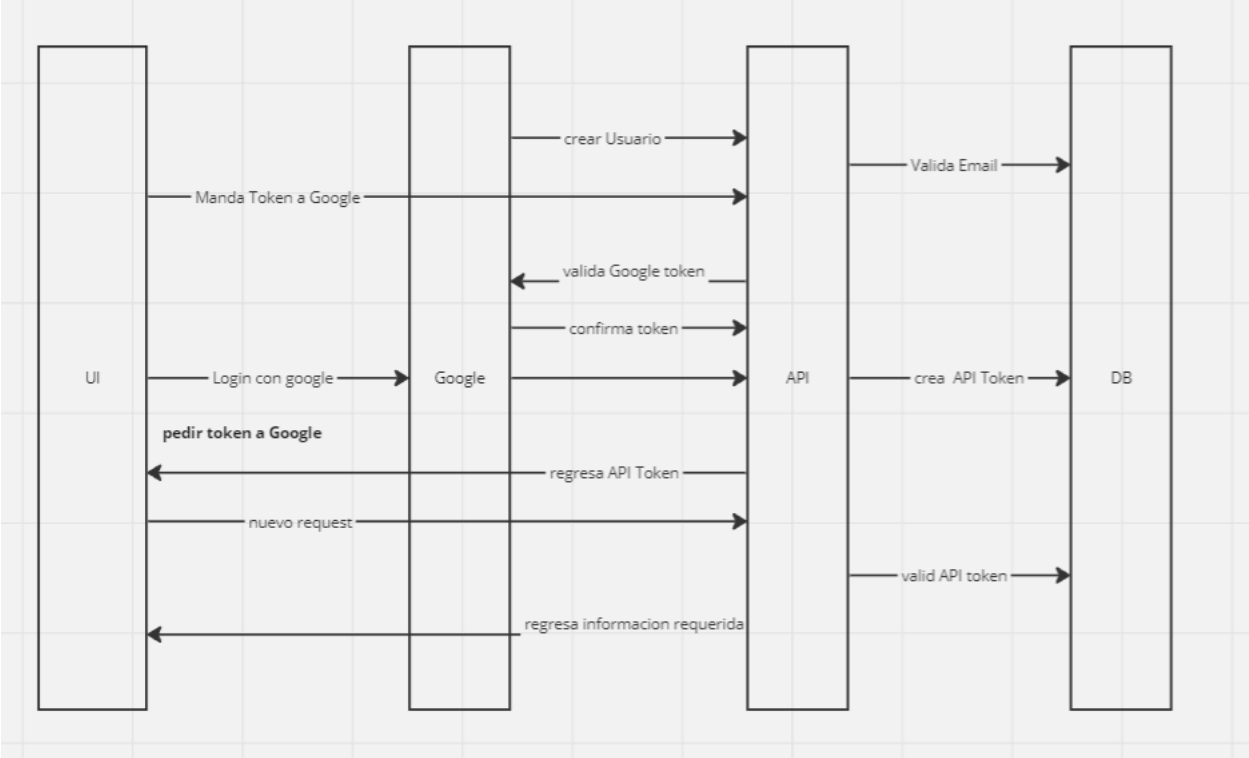
```
create: async(req, res) => {

  //Poner en el modelo de Users que este ususuario tambien es artista
  const user_id = req.params.id
  try {
    const options = { new: true };
    const user = await userModel.find({ id: user_id });
    const userId = user[0].id//
    const updatedData = {
      artistStatus: 1,
    };

    userModel.findOneAndUpdate(
      {id:user_id },updatedData, options
      )
      .then(result => {
          console.log('user updated:', result);
        })
        .catch(error => {
          console.error('User couldnt de updated:', error);
        });

  } catch (error) {
    console.error('Error al agregar como artista:', error);
    res.status(500).json({ error: 'Error al agregar como artista:' });
  }


  const artistData = {
      id: user_id,
      //username: req.body.username, // Nuevo nombre de artista
      //email: req.body.email,
      //password: req.body.password,
      chat: -1,
      albums: [],
      events: []
    };
    const newArtist = new artistModel(artistData);

  if (!newArtist) {
    return res.status(404).json({ error: 'Artista no creado' });
  }

  newArtist.save();
  res.json(newArtist)
},
```

Controlador para borrar album y todas las canciones adentro del album

```
remove: async(req, res) => {
    const collectionId= req.params.id;

    let collection = await model.findById(collectionId)
    if (!collection || !collection.songs) {
        return res.status(404).send("La colección no se encontró.");
    }


    let songArray = collection.songs

    for (let song of songArray) {
        await modelSong.findOneAndDelete(song);
    }

    model.deleteOne({ _id: collectionId })
        .then(result => {
            if (result.deletedCount === 1) {
                res.send("Colección eliminada correctamente");
            } else {
                res.status(404).send("La colección no se encontró o no pudo ser eliminada.");
            }
        })
        .catch(err => {
            console.error(err);
            res.status(500).send("Error al eliminar la colección.");
        });
}
```

Google Passport

Diagrama de inicio de sesión con google



Cuando un usuario inicia sesión por primera vez a través de Google en nuestra aplicación, nuestro objetivo es proporcionar una experiencia fluida y sin problemas. En el backend, se manejaría este escenario mediante un proceso de registro automático. Al recibir la información del usuario desde Google, verificamos si ya existe en nuestra base de datos.

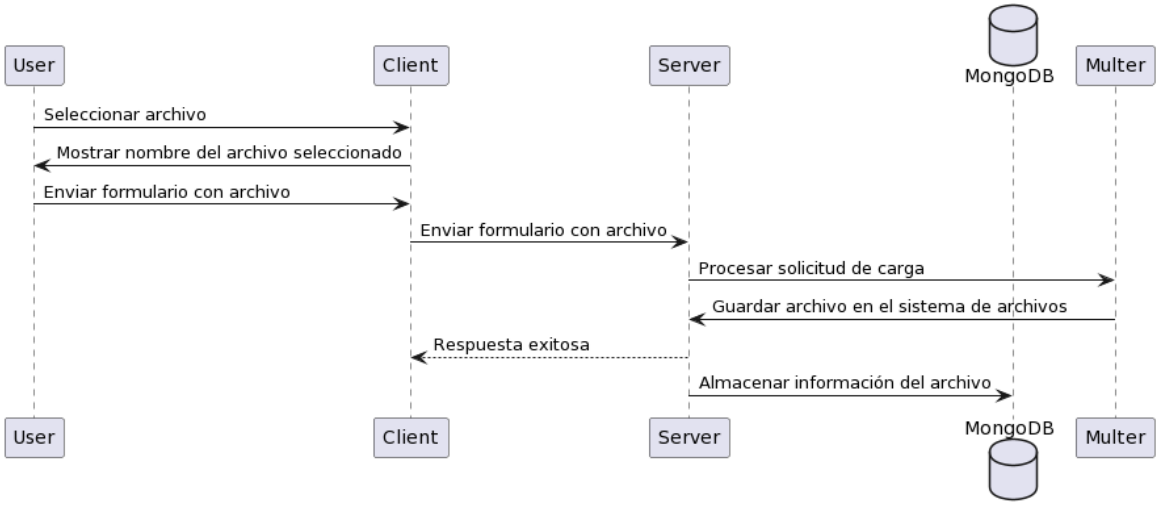
En el frontend, el flujo comienza cuando el usuario hace clic en el botón de inicio de sesión con Google. Utilizamos Angular para gestionar este evento, se encarga de comunicarse con nuestro backend, enviando el token de identificación de Google para validar la autenticidad del usuario.

Una vez que el backend recibe el token, verifica si el usuario está registrado. Si está registrado, devuelve un token de acceso que se almacena en el frontend para las futuras solicitudes. Sin embargo, si el usuario no está registrado, el backend devuelve un código de error específico (por ejemplo, 404) indicando la necesidad de registro.

Al detectar el código de error, el frontend llamaría a una función para el auto registro, que se encarga de enviar la información del usuario, recibida de Google, al backend para el proceso de registro automático. En el backend, implementamos la lógica necesaria para crear un nuevo usuario basado en esta información, asignarle un identificador único y, opcionalmente, generar un token de acceso para sesiones futuras.

De vuelta en el frontend, gestionamos la respuesta del backend después del registro automático. Si el registro es exitoso, almacenamos el nuevo token y redirigimos al usuario a la página deseada, por ejemplo, su perfil. Este enfoque permite que los usuarios se integren fácilmente en nuestra aplicación sin la necesidad de pasos manuales de registro.

Carga de archivos



En el contexto de nuestro proyecto de música, la funcionalidad de carga de archivos se implementará utilizando la biblioteca Multer en conjunto con Express para gestionar las solicitudes HTTP. La carga de archivos es esencial para permitir a los usuarios subir archivos relacionados con su experiencia musical, como imágenes de perfil y canciones.

- **Seleccionar Archivo:**
  - El usuario selecciona un archivo a través de la interfaz de usuario.
- **Mostrar Nombre del Archivo Seleccionado:**
  - El cliente muestra el nombre del archivo seleccionado para confirmación visual al usuario.
- **Enviar Formulario con Archivo:**
  - El usuario envía el formulario que contiene el archivo al servidor.
- **Procesar Solicitud de Carga:**
  - El servidor recibe la solicitud y utiliza Multer para procesar la carga del archivo.
  - Multer guarda información del archivo en MongoDB para su referencia futura.
  - Multer guarda el archivo en el sistema.
- **Respuesta Exitosa:**
  - El servidor responde al cliente indicando que la carga del archivo fue exitosa.

Middleware para la carga de archivos

```
const multer = require('multer');
const { v4: uuidv4 } = require('uuid');

const validExtensions = ['jpg', 'jpeg', 'PNG', 'mp3'];

const storage = multer.diskStorage({
    destination: (req, file, cb) =>{
        cb(null, 'uploads');
    },
    filename: (req, file, cb) =>{
        const id = uuidv4();
        const ext = file.originalname.split('.').pop();
        // const ts = new Date().getTime();
        const name = id+`.${ext}`;
        cb(null, name);
    },

});

const fileFilter = (req, file, cb) =>{
    const ext = file.originalname.split('.').pop();
    const valid = validExtensions.includes(ext);
    cb(null, valid);
}

const upload = multer({fileFilter, storage});

module.exports = upload;
```

Carga de los archivos de audio

```typescript
export class CreateSongComponent {
  constructor(private dialogRef: MatDialogRef<CreateSongComponent>, private formBuilder: FormBuilder, private songService: SongService) {
    this.signupForm = this.formBuilder.group({
      // your existing form controls
      name: ['', [Validators.required, Validators.minLength(2)]],
      artistID: ['']
    });
  }

  selectedFile: string = '';
  selectedSong: string = '';
  signupForm: FormGroup;
  file: File | null = null;

  ngOnInit() {
    // Retrieve the token from localStorage
    const token = localStorage.getItem('token');

    // Check if the token is not null or undefined before setting the value
    if (token) {
      // Set the 'artistId' form control value
      this.signupForm.get('artistID')!.setValue(token);
    }
  }
```

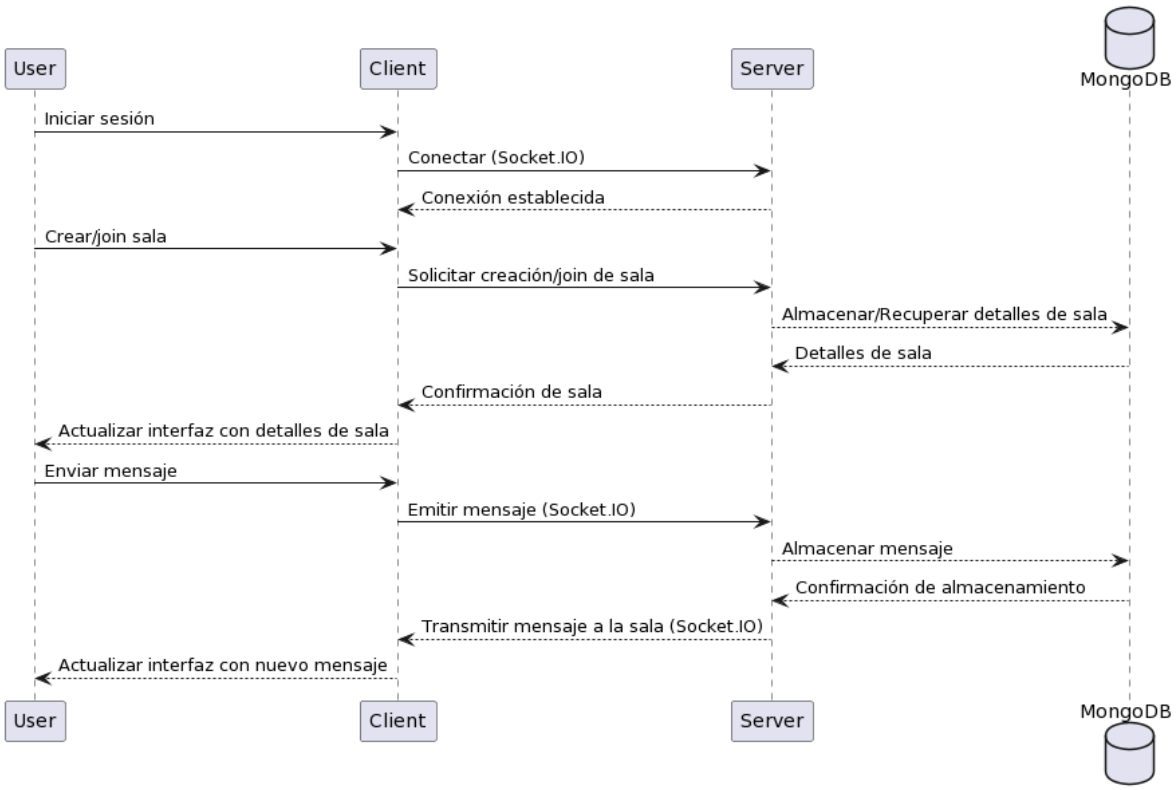
```typescript
onSubmit(): void {


  if (this.signupForm.valid && this.file) {
    const formData = this.signupForm.value;
    console.log("Submit");
    this.songService.uploadSong(this.file, formData).subscribe(
      (data) => {
        console.log('Song uploaded successfully:', data);
        // Handle the response from the server if needed
      },
      (error) => {
        console.error('Error uploading song:', error);
      }
    );
  }
  this.closeDialog()
}
```

```typescript
  uploadSong(file: File, songData: any): Observable<any> {
    const formData: FormData = new FormData();
    formData.append('file', file, file.name);

    formData.append('name', songData.name);
    formData.append('artistID', songData.artistID);
    const headers = new HttpHeaders();
    // headers = headers.append('Authorization', 'Bearer YOUR_ACCESS_TOKEN');

    return this.http.post<any>(`${this.apiUrl}songs/create`, formData, { headers });
  }

  getSongs(){
    return this.http.get<Song[]>(`${this.apiUrl}songs`);
  }
  getSongsById(id:string){
    return this.http.get<Song[]>(`${this.apiUrl}songs/${id}`);
  }
}
```

Reproducción de los archivos de audio

```typescript
ngOnInit(): void {
  // this.playSong(0);
  this.songService.getSongs().subscribe((data)=>{
    data.forEach(song => {
      this.playlist.push([song.song, song.name])
    });
    console.log(this.playlist);
  })

}

playSong(index: number) {
  this.currentSongIndex = index;
  const audioPlayer = document.getElementById('audioPlayer') as HTMLAudioElement;
  audioPlayer.src = environment.apiUrl+'assets/'+this.playlist[this.currentSongIndex][0]+'.mp3';
  audioPlayer.load();
  audioPlayer.addEventListener('canplaythrough', () => {
    this.duration = audioPlayer.duration;
    this.songName = this.playlist[this.currentSongIndex][1];
    // Set the initial volume
    audioPlayer.volume = this.currentVolume;
  });
}
```

Uso de sockets



- **Iniciar Sesión:**
  - El usuario inicia sesión en la aplicación.
- **Conectar con Socket.IO:**
  - El cliente establece una conexión con el servidor utilizando Socket.IO después de iniciar sesión.
- **Creación o Unión a Sala de Chat:**

    - Cuando un usuario crea o se une a una sala de chat, el cliente emite un evento al servidor a través de Socket.IO.
    - El servidor, al recibir la solicitud, realiza las siguientes acciones:
        - **Crear Nueva Sala (si es el caso):**
            - Genera un ID único para la nueva sala.
            - Almacena la información de la nueva sala en la base de datos MongoDB en la colección **chats**.
            - Emite un evento a todos los clientes conectados para informarles sobre la nueva sala.
        - **Unirse a Sala Existente (si es el caso):**
            - Verifica la existencia de la sala en MongoDB.
            - Actualiza la información de la sala (por ejemplo, la lista de miembros) en la base de datos.
            - Emite un evento a todos los clientes conectados para informarles sobre la actualización de la sala.
- **Enviar Mensaje:**
    - Cuando un usuario envía un mensaje, el cliente emite un evento al servidor a través de Socket.IO, junto con el contenido del mensaje y el ID de la sala.
    - El servidor, al recibir el evento, realiza las siguientes acciones:
        - Almacena el mensaje en la base de datos MongoDB en la colección **messages**.
        - Emite el mensaje a todos los clientes conectados a la misma sala para que lo muestren en tiempo real.
- **Almacenamiento y Recuperación en MongoDB:**
    - Tanto al crear o unirse a una sala como al enviar un mensaje, se realizan operaciones de lectura/escritura en la base de datos MongoDB para mantener un registro persistente de salas y mensajes.
- **Actualización de la Interfaz de Usuario:**
    - Después de cada operación exitosa (crear/join sala, enviar mensaje), el servidor emite eventos a los clientes afectados para que actualicen sus interfaces de usuario en tiempo real con la información más reciente.

Lógica para db de guardar mensajes:

```
const messegeModel = require('./src/models/messege');
const chatModel = require('./src/models/chat');
const { v4: uuidv4 } = require('uuid');


async function saveMessage(data) {
    console.log("Received data for saving message:", data);

    try {
        const chat_id = data.chatId;

        const chat = await chatModel.findOne({ id: chat_id });
        if (!chat) {
            return null;
        }

        const messageId = uuidv4();
        const newMessage = new messegeModel({
            id: messageId,
            chat_id: chat_id,
            user_id: data.user_id,
            content: data.content,
            date: data.date
        });

        await newMessage.save();

        chat.messege_ids.push(messageId);
        await chat.save();

        return newMessage;
    } catch (error) {
        throw error;
    }
}

module.exports = { saveMessage };
```

Conexiones

```
const io = socketIo(server,{
    cors:{origin: "*",
    methods: ["GET", "POST"]}
});

io.on('connection', (socket) => {
    socket.on('messegeSent', async (data) => {
        try {
            const savedMessage = await saveMessage(data);
            io.emit('messegeReceived', savedMessage);
        } catch (error) {
            console.error('Error saving message:', error);
        }
    });
});
```

```
ngOnInit(): void {
  console.log('here in ng');
  this.socket.on('userConnected', ()=>{
    console.log("un usuario se conecto");
  })
  this.socket.on('messegeReceived', (messege:Messege) => {
    this.allMesseges.push(messege)
  })
  this.setUserData()
}
```

Mandar mensajes desde frontend

```
  }
  sendMessege() {
    const newMessage: Messege = {
      id: '',
      user_id: this.user.id,
      chatId: this.currentChat,
      content: this.content,
      date: new Date()
    };

    this.socket.emit('messegeSent', newMessage);


    this.content = '';
  }
```

```
export class ChatService {

  constructor(private httpClient: HttpClient) { }

  joinChat(chat_id:string, user_id:string) {
    if(chat_id && user_id){
      const body = {
        'userId' : user_id,
        'chatId' : chat_id
      }
      const url: string = `${environment.apiUrl}chats/join`;
      return this.httpClient.put(url, body);
    }
    return 'not found'
  }



  getUserChatsbyChatId(chat_id:string):Observable<Chat> {
    const url: string = `${environment.apiUrl}chats/${chat_id}`
    return this.httpClient.get<Chat>(url)
  }
  getMessegesbyChatId(chat_id:string):Observable<Messege[]>{
    const url: string = `${environment.apiUrl}chats/messege/`+chat_id;
    return this.httpClient.get<Messege[]>(url)
  }

}
```