



EventHorizon

Plataforma de Eventos y Networking Online.

Un espacio virtual para organizar eventos, conferencias y facilitar el networking entre los participantes.

Jose Fernando Chavez Castañeda.
Sofia López Osuna
Froylan Cisneros Alvizo

Competencia:

<https://doodle.com/es/>

<https://whova.com/>

<https://propartyplanner.com/>

<https://eventwo.com/es/>

<https://www.eventtia.com/es/inicio>

<https://nuboxmkt.mx/>

Stack de Tecnologías a utilizar.

Frontend

React.js.

Backend

Node.js con Express.js: Node.js es eficiente para manejar conexiones concurrentes, lo que es crucial para las funcionalidades en tiempo real.

Socket.IO: Para la comunicación en tiempo real, permitiendo chats, videoconferencias, y actualizaciones en vivo de los eventos.

Base de Datos

MongoDB.

Autenticación y Seguridad

Auth0 o Firebase Authentication: Proporcionan autenticación segura y fácil de implementar con servicios de terceros como Google, Facebook, y LinkedIn.

Herramientas Adicionales

WebRTC: Para videoconferencias peer-to-peer en las salas de networking virtual.

Roles de Usuarios:

1. Organizador de Eventos:

- Descripción: Los organizadores de eventos son responsables de planificar, crear y gestionar eventos en la plataforma.

- Tareas Principales:

- Crear nuevo evento.
- Configurar detalles del evento (fecha, hora, descripción, etc.).
- Administrar inscripciones de participantes.
- Interactuar con participantes durante el evento.

2. Participante del Evento:

- Descripción: Los participantes del evento son usuarios que se inscriben y participan en los eventos organizados en la plataforma.

- Tareas Principales:

- Buscar eventos de interés.
- Registrarse para participar en eventos.
- Acceder a salas de conferencias virtuales.
- Interactuar con otros participantes.

3. Administrador del Sistema:

- Descripción: Los administradores del sistema tienen privilegios para gestionar y mantener la plataforma.

- Tareas Principales:

- Gestionar usuarios y roles.
- Monitorizar la actividad del sistema.

Diagramas:

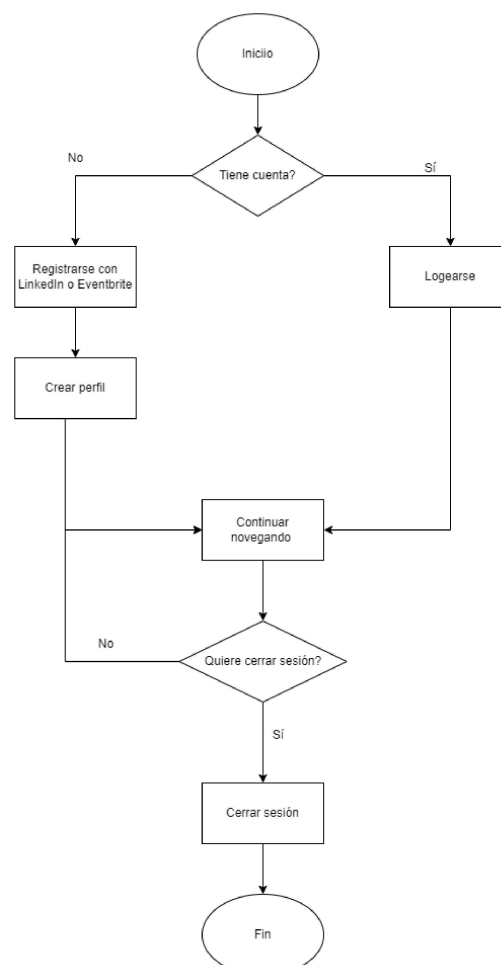


Diagrama para organizador de eventos

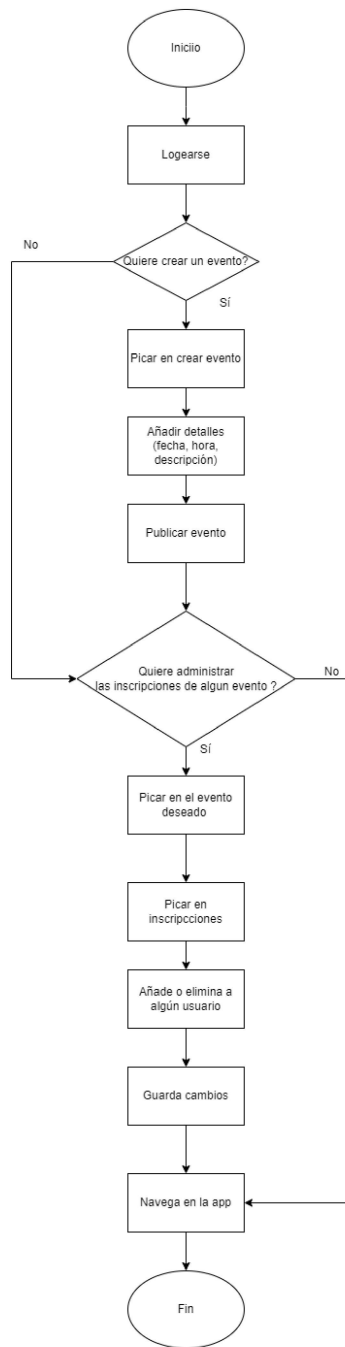
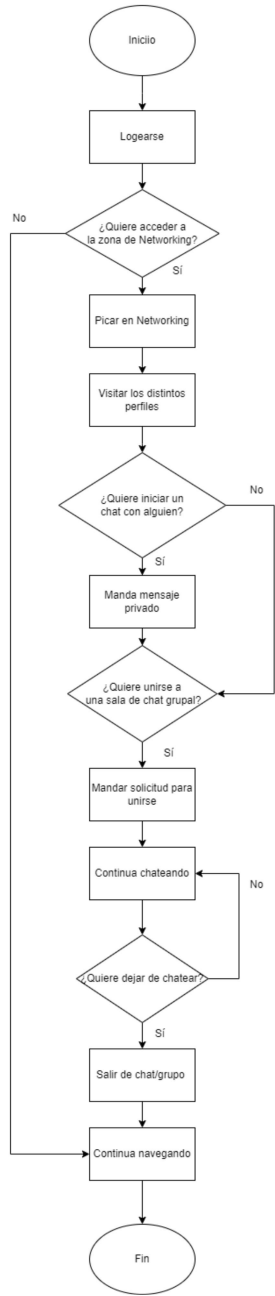
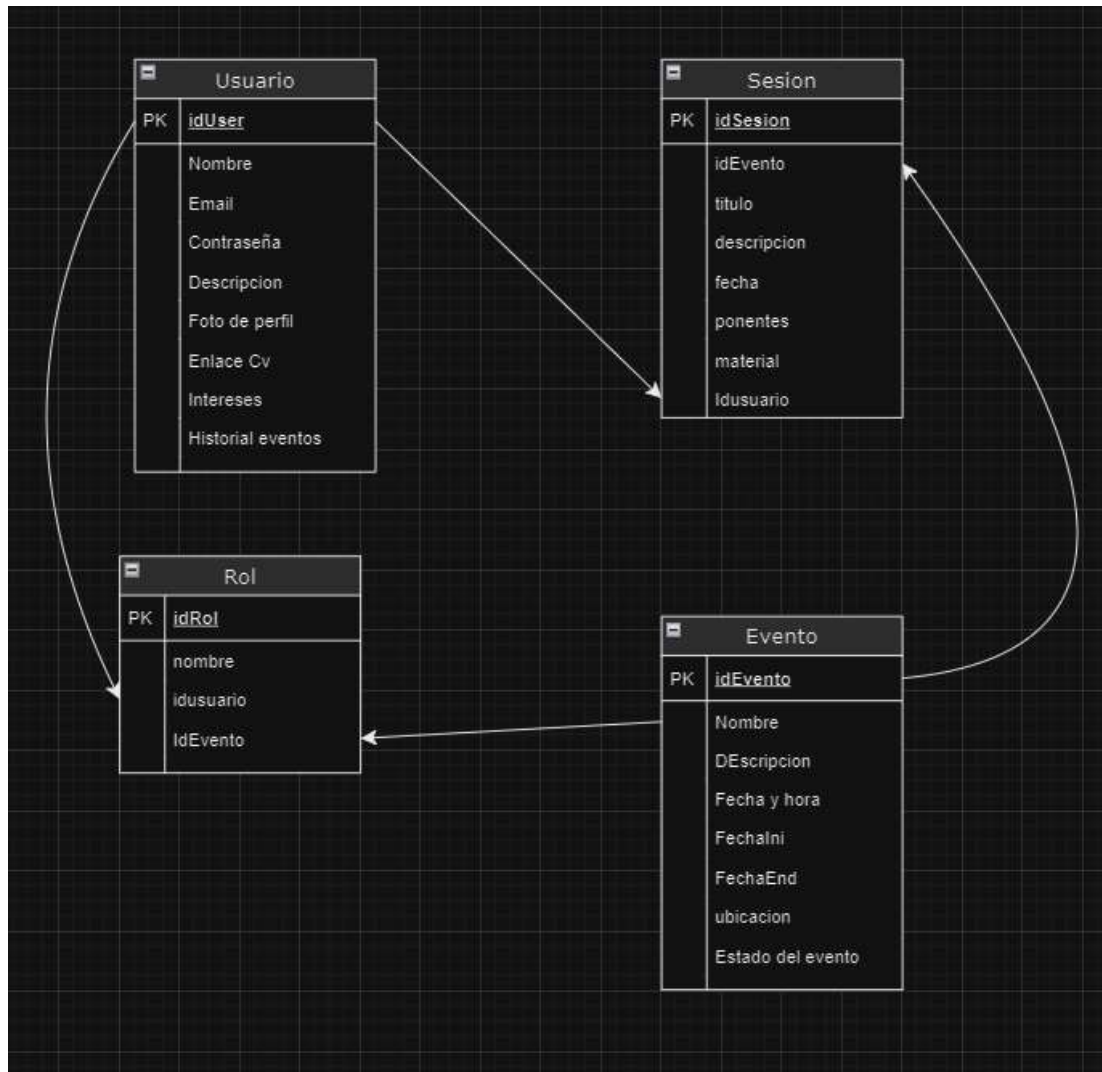


Diagrama para zona de Networking



Entidad-Relacion:

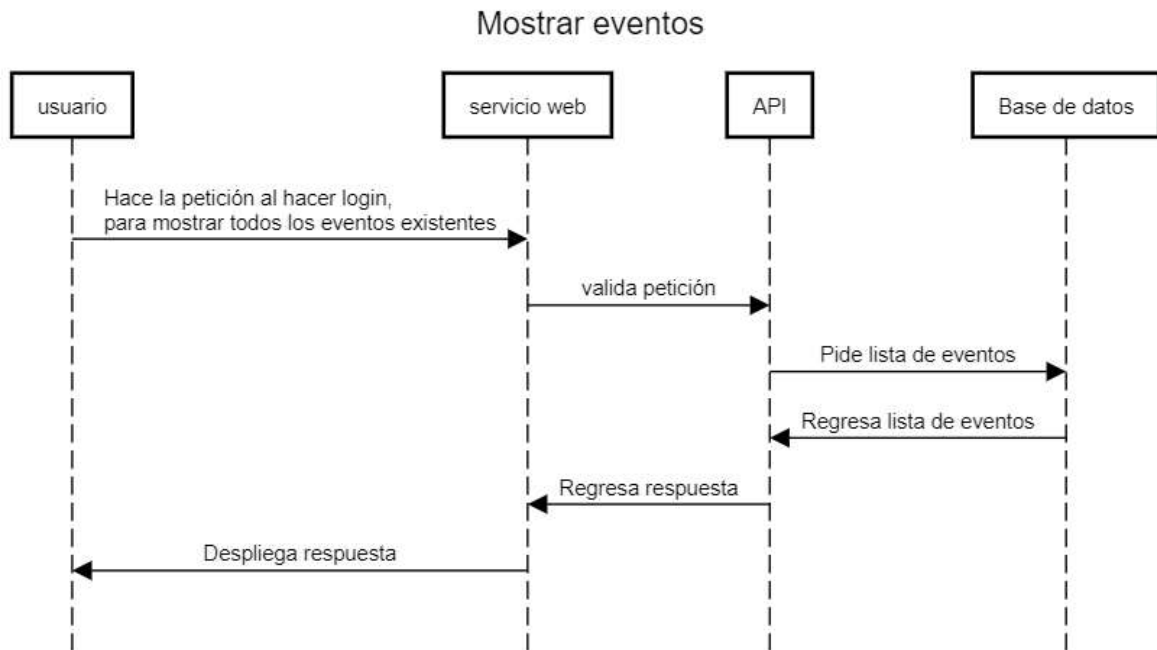


Alcance (temas a cubrir de acuerdo a la guía de aprendizaje) y breve descripción de éste

- MVC y Estructuras de código enfocadas al backend: Modelo Vista Controlador y patrones que nos servirán para organizar código de backend
- Frameworks para el Backend: Entornos de desarrollo que facilitarán la creación de el proyecto
- Webpack: Herramienta para empaquetar y minificar archivos JavaScript y otros recursos.
- TypeScript: Lenguaje de programación que añade tipos a JavaScript.
- Handlebars: Plantillas para generar HTML dinámico.

- Paradigma Orientado a Objeto y Patrones de diseño: Se utilizará el enfoque POO para poder generar soluciones que se usan de manera común en la industria para resolver problemáticas que se puedan generar.
- Testing e integración continua: Pruebas para asegurar la calidad del código.
- Plataformas para despliegue de aplicaciones: Servicios para alojar y ejecutar aplicaciones web

Diagrama de secuencia:

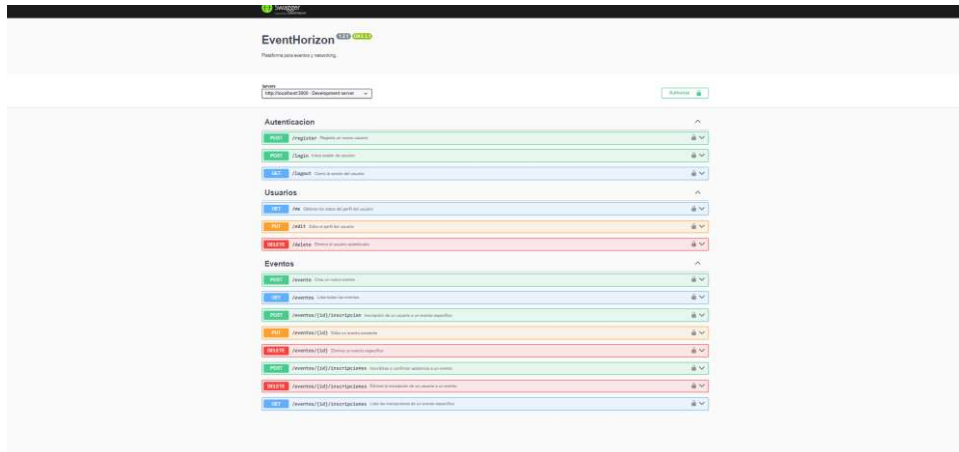


Módulos:

- evento: (lo podrán utilizar usuarios registrados como organizadores de eventos) Definir funciones para la creación de eventos, filtro de eventos, inscripción, etc.
- usuarios: (disponible para todos los usuarios) creación de usuario, log in, actualización de información, solicitar cambio de rol, etc.
- administradores: (disponible solo para administradores) dar de baja a un usuario, dar de alta a un usuario, inscribir o desinscribir a un usuario de un evento, eliminar evento, agregar evento, modificar información del evento, etc.
- autenticación: (disponible solo para administradores) funcionalidad para autenticar a los usuarios,

Swagger

<http://localhost:3000/api-docs/>



Explicación

Se definen rutas y operaciones utilizando formato YAML, los comentarios están ubicados encima de cada ruta y describen la operación, sus parámetros, respuestas y otros detalles

Se importa Swagger UI y Swagger Docs para poder configurar la interfaz de Swagger y proporcionar la documentación de la API en formato JSON

Finalmente se genera la documentación usando los comentarios

Autenticación y permisos

Se tienen dos Middlewares de autenticación, en `verifyToken` se evalúa si se proporciona un token de autorización en la solicitud, si sí fue proporcionado entonces se intenta decodificar el token usando una clave secreta. Si la verificación es exitosa entonces se agrega el usuario al `Request` y se llama a `next()` para pasar a la siguiente función. Si no se da un token o para que la verificación no sea exitosa se envía un código 403 de error, o 401 seguido de un mensaje indicando el error.

En establecerContextoAutenticación obtenemos el token de autorización del encabezado o de las cookies, Si se encuentra un token, intenta verificar y decodificar el token. Si es exitoso, establece unas variables en Response. De manera contraria, si la verificación falla, se establece use rLoggedIn en false y se llama a next()

Ejemplo:

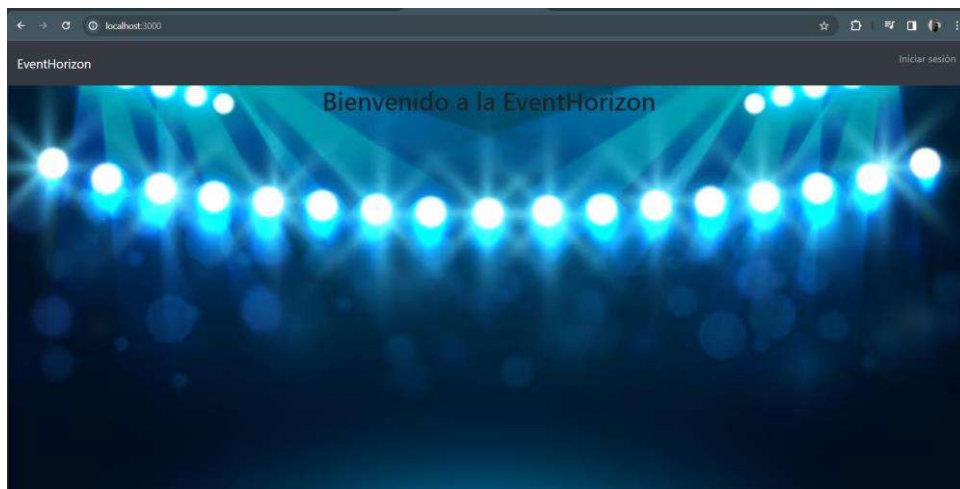
En el caso de nuestro proyecto, cada vez que se quieran ver los eventos disponibles el usuario debe estar autenticado forzosamente, se hace llamado a esta ruta: `router.get('/eventos', verificarToken, listarEventos);`

Si el usuario si está autenticado entonces sin problemas se le muestran los diferentes eventos futuros que hay en la plataforma, de otro modo si el usuario no está autenticado entonces no se le podrán

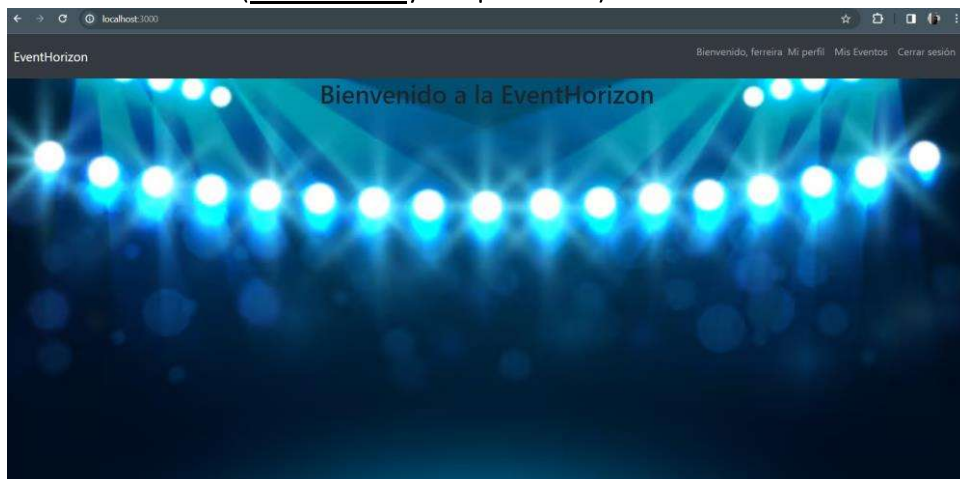
mostrar los eventos y se le muestra un mensaje de token no válido.

Capturas de pantalla de distintas respuestas:

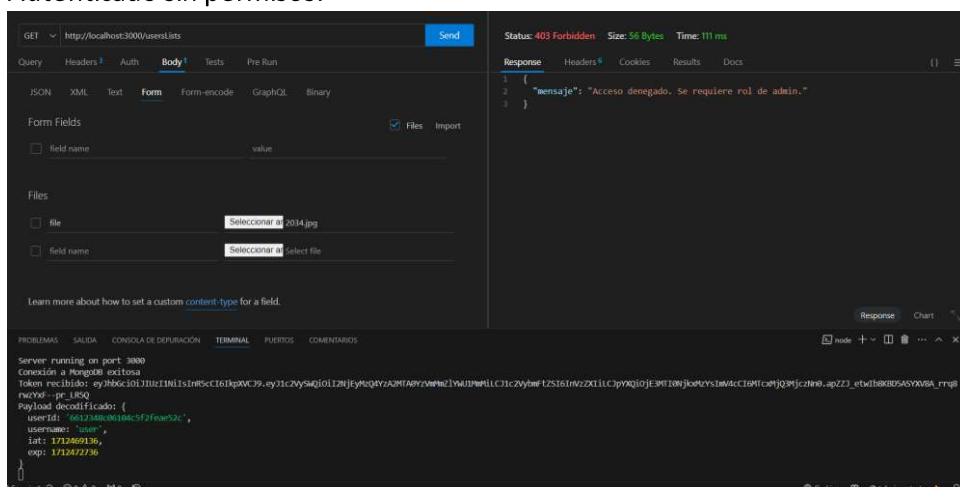
Home sin autenticar(No autenticado):



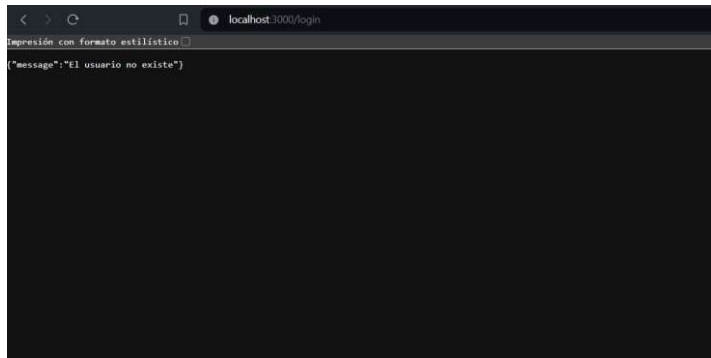
Home autenticado (Autenticado y con permisos:):



Autenticado sin permisos:



Mensaje de error al intentar autenticarse con un perfil no autorizado (front en proseso)




Para la 3ra entrega se va a mostrar todas las rutas completas automatizadas en postman y pruebas de como funciona la utenticacion por google:

Iniciar sesión

Usuario:

Contraseña:

Iniciar sesión

 **CONTINUAR CON GOOGLE**

¿No tienes una cuenta? [Regístrate aquí](#)

Registro de usuario


Nombre completo:

Usuario:

Correo electrónico:

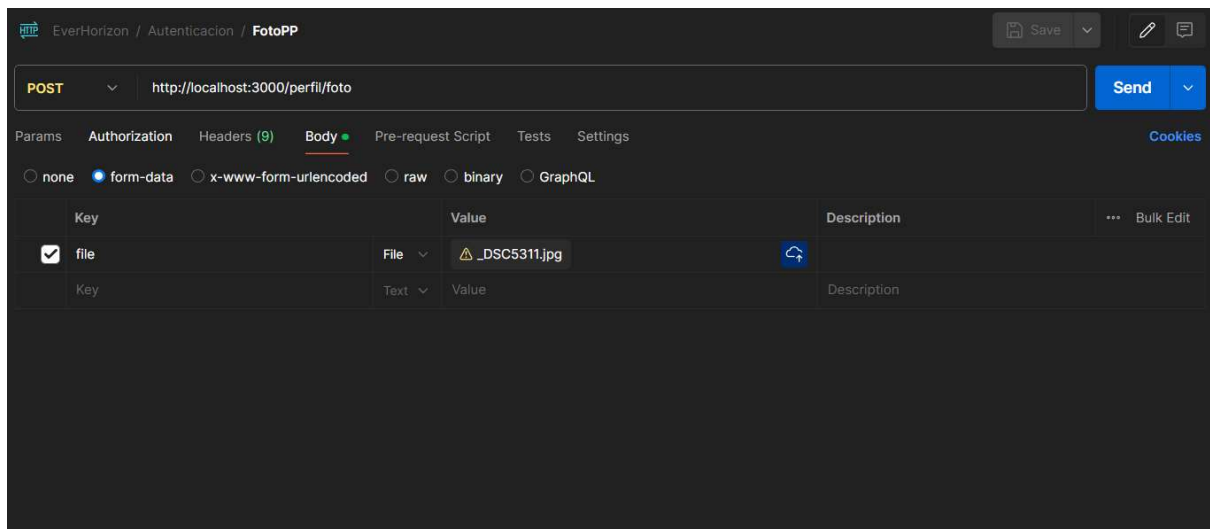
Contraseña:

Registrarse

 **REGISTRATE CON GOOGLE**

¿Ya tienes una cuenta? [Inicia sesión aquí](#)

Tenemos 2 botones de Google, que en realidad van a la misma ruta, ya que el middleware de Google revisa si el usuario está ya en la bd y si no lo crea y agrega su foto de una vez. Y la pantalla de inicio se ve igual con el nombre de usuario de la cuenta de Google, para la subida de imágenes a aws se hace ahorita solo en el backend y funciona perfectamente.



asi lo mandamos y se subira a aws la imagen y la ruta a la bd:

```
_id: ObjectId('65f52ae2909cfb1a3811153f')
fullname: "jose fernando"
username: "ferreira"
email: "elreydelapi@gmail.com"
password: "$2a$10$RBkMopR8XxJvQBNUeUs5K0aSo5NLXM/zpRWbzQsiclIZXEbe7mGSi"
role: "admin"
_v: 0
profilePicture: "https://eventhorizon.s3.us-east-2.amazonaws.com/d6511820-59ff-492c-afb..."
```

Objetos (2) Información

Los objetos son las entidades fundamentales que se almacenan en Amazon S3. Puede utilizar el [inventario de Amazon S3](#) para obtener una lista de todos los objetos de su bucket. Para que otras personas obtengan acceso a sus objetos, tendrá que concederles permisos de forma explícita. [Más información](#)

	Nombre	Tipo	Última modificación	Tamaño	Clase de almacenamiento
<input type="checkbox"/>	5edb7614-2ed0-48a6-b081-3c9c08abf414.jpg	jpg	19 Apr 2024 11:44:29 PM CST	13.2 MB	Estándar
<input type="checkbox"/>	d6511820-59ff-492c-afb7-4da3fbd4f6e6.jpg	jpg	6 Apr 2024 11:43:14 PM CST	6.1 MB	Estándar

My Workspace | New | Import | EverHorizon

EverHorizon - Run results

Run today at 00:57:31 | View all runs

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	1	1s 298ms	7	54 ms

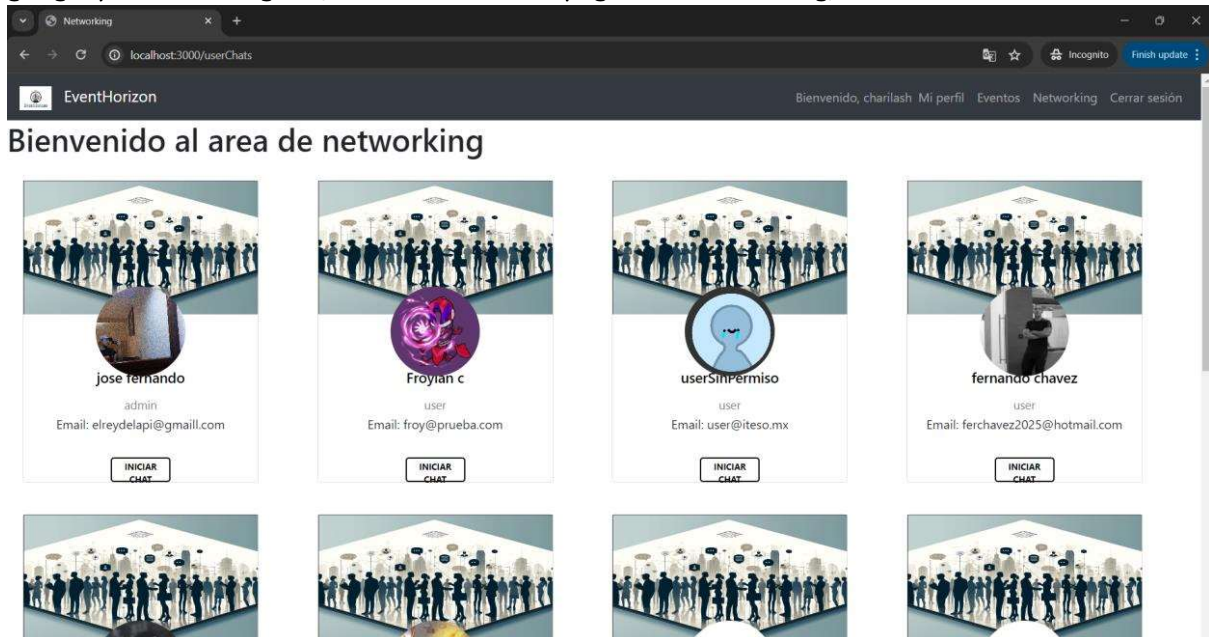
RUN SUMMARY

Test	Pass	Fail	Skipped
POST Registro	2	0	0
POST Login	1	0	0
GET logout	1	0	0
GET me	1	0	0
PUT edit	1	0	0
GET Traer eventos	1	0	0

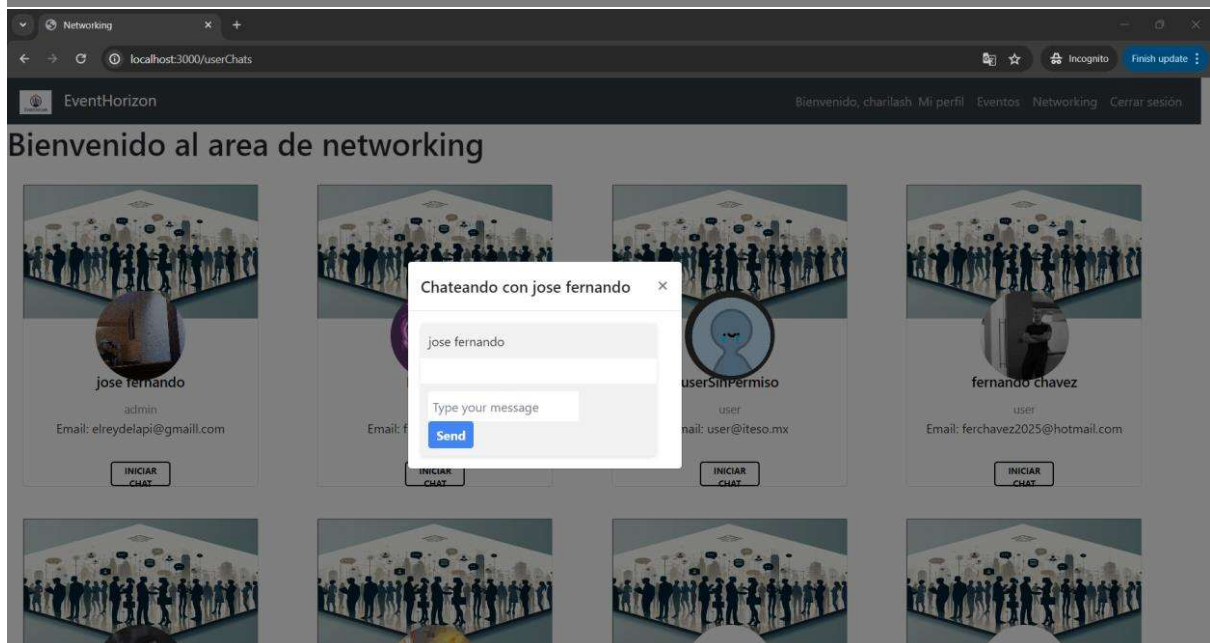
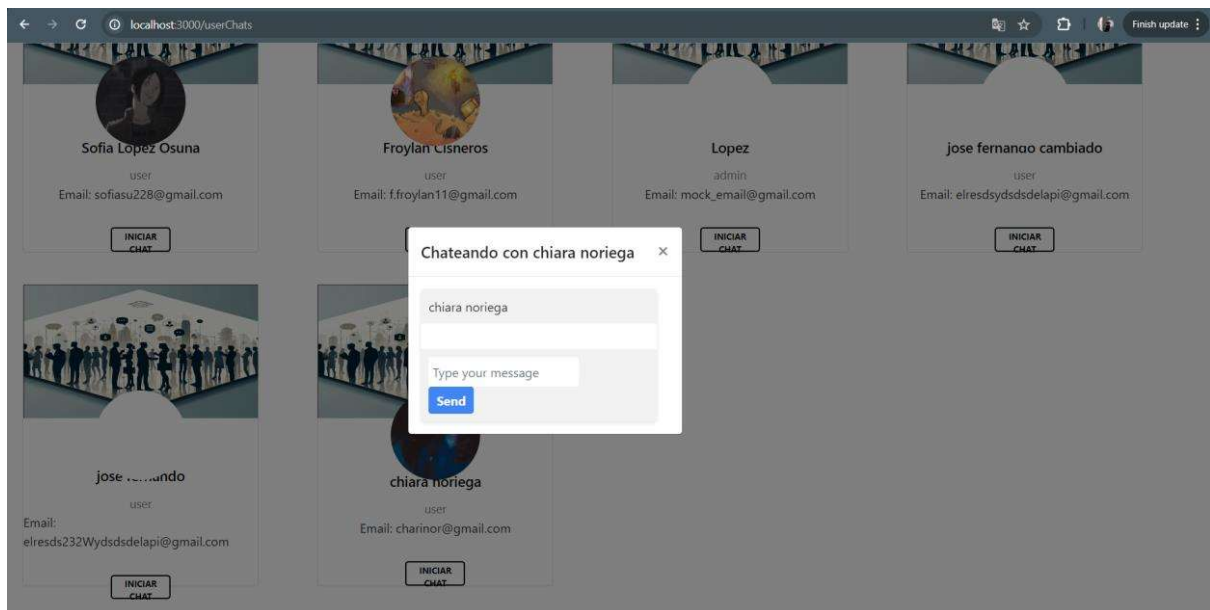
Pruebas automatizadas

Socket, chat en vivo.

Para el uso de socket se hace en un chat en vivo simulando 2 computadoras se usa una sesion en google y otra en incognito, en las 2 entras a la página de networking, con las 2 cuentas

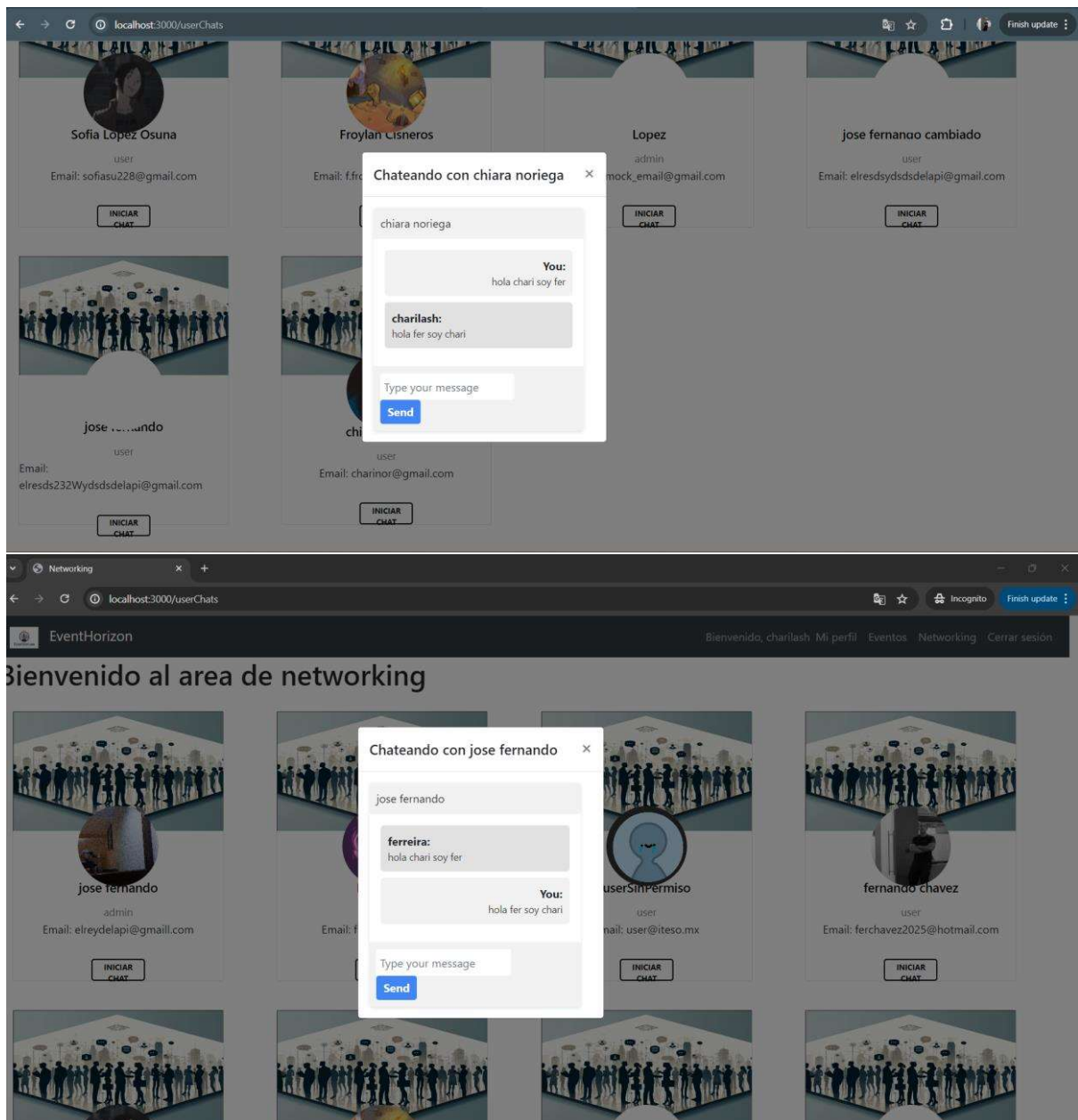


despues eliges desde cada sesion con quien quieres hablar en este caso usaremos a chari para mandarle mensaje a jose y de la sesion de jose para mandarle msj a chari.



cuando mandas un msj los tuyos aparecen en you siempre y los que recibes con el nombre de la sesion del que lo envio:

la otra persona lo recibe en menos de 1 seg y lo mismo al revés:



y cuando terminand de chatear podemos ver en el flujo como se desconectan:

A new user connected
charilash joined the chat

A new user connected
ferreira joined the chat

Message from ferreira: hola chari soy fer
Message from charilash: hola fer soy chari

ferreira disconnected
charilash disconnected

Diagrama de flujo:



En cuanto al código usamos lo siguiente, primero en el index:

```
const httpServer = createServer(app);
const io = new SocketIOServer(httpServer);

httpServer.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});

io.on('connection', (socket) => {
  console.log('A new user connected');

  socket.on('newUser', (data) => {
    socket.data.username = data.user;
    console.log(`${data.user} joined the chat`);
    socket.broadcast.emit('newUser', data);
  });

  socket.on('newMessage', (data) => {
    console.log(`Message from ${socket.data.username}: ${data.message}`);
    socket.broadcast.emit('newMessage', { user: socket.data.username, message:
data.message });
  });

  socket.on('disconnect', () => {
    console.log(`${socket.data.username} disconnected`);
    socket.broadcast.emit('userLeft', { user: socket.data.username });
  });
});
```

Aquí se configura el socket que usará nuestra conexión para la comunicación en directo, se configura para los casos donde se conecta un nuevo cliente, un nuevo mensaje, otro usuario adicional y para cuando se desconectan, después creamos un modal con la información necesaria para usarlo como chat en vivo y con un js le pasamos los datos de los usuarios conectados y como vamos modificando esas partes del modal en vivo con las funciones de newMessage:

```
document.addEventListener('DOMContentLoaded', () => {
  let socket;
  const chatButtons = document.querySelectorAll('.card__btn');

  let activeFullname;
  let myUsername = document.querySelector('.users-
container').dataset.myUsername;

  chatButtons.forEach(function (btn) {
    btn.addEventListener('click', function () {
      const fullname = btn.getAttribute('data-fullname');
```

```

        activeFullname = fullname;
        var modalTitle = document.getElementById('chatModalLabel');
        modalTitle.textContent = 'Chateando con ' + fullname;
        var chatUsername = document.querySelector('.chat-h2');
        chatUsername.textContent = fullname;

        if (!socket) {
            socket = io();

            socket.on('connect', () => {
                console.log('Connected to server');
                socket.emit('newUser', { user: myUsername });
            });

            socket.on('newMessage', (data) => {
                const sender = data.user === myUsername ? 'You' :
data.user;

                appendMessage(data.message, 'incoming', sender);
            });

            socket.on('userLeft', (data) => {
                appendMessage(`${data.user} left the chat`, 'info');
            });
        }

        document.querySelector('.chat-body').innerHTML = '';
    });
});

$('#chatModal').on('hidden.bs.modal', function () {
    if (socket) {
        socket.emit('disconnectRequest');
        socket = null;
    }
});

const messageForm = document.querySelector('.chat-footer form');
const messageInput = document.querySelector('.form-control');
const chatBody = document.querySelector('.chat-body');

if (messageForm) {
    messageForm.addEventListener('submit', function(event) {
        event.preventDefault();
        const message = messageInput.value.trim();
        if (message && socket) {
            socket.emit('newMessage', { user: myUsername, message });
            appendMessage(message, 'outgoing', 'You');
        }
    });
}

```

```

        messageInput.value = '';
    }
    });
}

function appendMessage(message, type, sender = activeFullname) {
    const messageDiv = document.createElement('div');
    messageDiv.classList.add('message', type);
    messageDiv.innerHTML = `<strong>${sender}</strong>
    <p>${message}</p>`;
    chatBody.appendChild(messageDiv);
    chatBody.scrollTop = chatBody.scrollHeight;
}
});

```

En if(!socket) no es duplicado en realidad es necesario ya que se usa para determinar de quien es el mensaje actual por la interaccion entre varias sesiones ya que el de index procesa y retransmite los mensajes a los clientes y en el js recibe los mensajes y actualiza la interfaz.