Universidad de Colima

Facultad de Telemática
Ingeniería en Tecnologías de Internet
Semestre febrero -julio 2021

Colima Col., a 25 de junio del 2021

Asignatura: Programación distribuida de servicios de internet

Catedrático: Montaño Araujo Sergio Adrián

Documento

"Documentación de proyecto chat con WebSockets"

4° C

Paz Zamora Alfredo

Contenido

Introducción

El propósito de este proyecto es analizar la importancia de los WebSockets en las aplicaciones de mensajería online, utilizando la comunicación bidireccional entre cliente y servidor a través de un único socket TCP. Este protocolo va más allá del paradigma típico de solicitud y respuesta HTTP. Con WebSockets, el servidor puede enviar datos a un cliente sin que el cliente inicie una solicitud, lo que permite algunas aplicaciones muy interesantes. W1ebSockets puede ser muy beneficioso para crear aplicaciones de transmisión de datos o comunicación en tiempo real en la web, como aplicaciones de chat y aplicaciones que transmiten imágenes u otros tipos de medios. Además, puede configurar fácilmente una conexión a cualquier punto final de socket utilizando la API de JavaScript WebSocket de su navegador.

Para crear una aplicación web moderna en tiempo real impulsada por Socket.io se hizo uso de varias funciones de socket.io, como salas y sesiones. Además, el realizar nuestra aplicación con socket.io nos ayuda a escalar el proyecto de manera sencilla.

A continuación, se muestra la documentación de la configuración de un sistema de chat con salas con Socket.io usando Node.js / Express en el lado del servidor y JavaScript en línea en el cliente del navegador.

Desarrollo

Código

Repositorio: https://github.com/AlfredoPazZamora/chatRealtime

Pagina web: https://chat-realtime-telematica.herokuapp.com/

Dependencias instaladas

Consolidate

Consolidate.js proporciona una función para los diversos motores de plantilla que admite.

- Instalación: *npm i consolidate*
- Documentación: https://www.npmjs.com/package/consolidate

Cookie-Parser

Cookie-parser es un middleware que analiza las cookies adjuntas al objeto de solicitud del cliente.

- Instalación: *npm i cookie-parser*
- Documentación: https://www.npmjs.com/package/cookie-parser

Express

Express.js es un marco de aplicación web gratuito y de código abierto para Node.js. Se utiliza para diseñar y crear aplicaciones web de forma rápida y sencilla.

- Instalación: *npm i express*
- Documentación: https://www.npmjs.com/package/express

Express-session

Express-session almacena los datos de sesión en el servidor; sólo guarda el ID de sesión en la propia cookie, no los datos de sesión. De forma predeterminada, utiliza el almacenamiento en memoria y no está diseñado para un entorno de producción.

- Instalación: *npm i express-session*
- Documentación: https://www.npmjs.com/package/express-session

Mustache

Mustache es una sintaxis de plantilla sin lógica. Se puede usar para HTML, archivos de configuración, código fuente, cualquier cosa. Funciona expandiendo etiquetas en una plantilla utilizando valores proporcionados en un hash u objeto.

- Instalación: *npm i mustache*
- Documentación: https://www.npmjs.com/package/mustache

Mysql

MySQL es el sistema de gestión de bases de datos relacional más extendido en la actualidad al estar basada en código abierto. En node js es un controlador que este escrito en JavaScript no requiere compilación y tiene licencia 100% MIT.

- Instalación: *npm i mysql*
- Documentación: https://www.npmjs.com/package/mysql

Nodemon

Nodemon es una herramienta que ayuda a desarrollar aplicaciones basadas en node.js al reiniciar automáticamente la aplicación de nodo cuando se detectan cambios de archivo en el directorio.

- Instalación: *npm i nodemon*
- Documentación: https://www.npmjs.com/package/nodemon

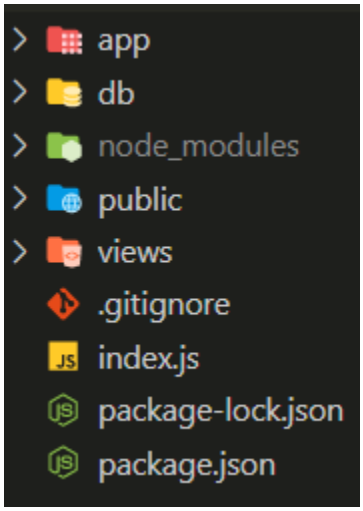
Socket.io

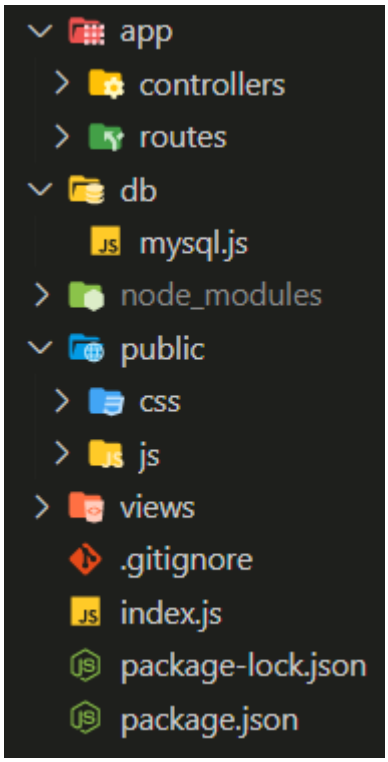
Es una librería open source con una amplia comunidad que nos ayudará a construir aplicaciones con conexión persistente entre cliente y servidor. Por lo que contaremos con librerías para cada lado.

- Instalación: *npm i socket.io*
- Documentación: https://www.npmjs.com/package/socket.io

Estructura de archivos

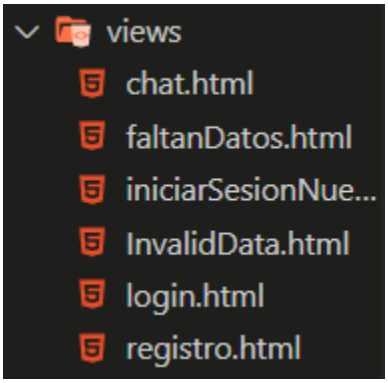
Se usa un sistema de orden de archivos, esto para tener un control mayor en los errores del código y saber dónde se encuentra, además de crear una mejor organización de archivos según sea su uso.





Directorio views

En este directorio se guardan todos los archivos HTML que después se renderizan para poder mostrarlos en el navegador. Cada archivo cuenta con un nombre relacionado a lo que se va a realizar.



Directorio DB

En el directorio de base de datos (DB), se encuentra el archivo de configuración para la conexión a nuestra base de datos.

En este archivo creamos la conexión con los parámetros necesarios para nuestra base de datos, en este caso tenemos dos diferentes parámetros, los primeros son para la base de datos de heroku, los que están comentados son para su uso de manera local.

```
let mysql = require('mysql');

let connection = mysql.createConnection({
    host: 'us-cdbr-east-04.cleardb.com',
    user: 'be6e7059cd1c8a',
    password: '94db54d6',
    database: 'heroku_08b9a5b304298d8'
    // host: 'localhost',
    // user: 'root',
    // password: '',
    // database: 'chat-sesion'
});
```

Antes de exportar la conexión debemos de saber si se conectó correctamente nuestra base, por lo que hacemos un condicional, en el cual si no tenemos algún error nos dará un mensaje de conexión exitosa.

```
connection.connect(
    (err) => {
        if(err){
            console.log(err);
            return;
        }else{
            console.log('BD esta conectada');
        }
    }
);

module.exports = connection;
```

Archivo index.js

El archivo index se divide en 5 partes:

Requerimiento de dependencias

Básicamente mandamos llamar las dependencias instaladas en nuestro archivo principal para poder usarlas a lo largo del programa; incluso el uso de variables importantes.

```
let express = require('express');
let cookieParser = require('cookie-parser');
let session = require("express-session");
let http = require('http');
let { Server } = require("socket.io");
let connection = require('./db/mysql');
let app = express();
let server = http.createServer(app);
let io = new Server(server);
```

Uso de variables y llamadas de paquetes

En esta parte usamos algunos de los paquetes de express para su uso, como lo son las sesiones de express, las cuales las usamos en socket.io para su manejo. Estas sesiones las manejaremos de una forma más fácil con el uso de la librería de cookie-parser.

```
let sessionMiddleware = session({
    secret: 'keyUltraSecret',
    resave: true,
    saveUninitialized: true
});

io.use((socket, next) => {
    sessionMiddleware(socket.request, socket.request.res, next)
})

app.use(sessionMiddleware);
app.use(cookieParser());
```

En esta sección tomamos las solicitudes con cargas útiles codificadas en urlencoded. En la segunda parte transformamos estas solicitudes en formato Json.

```
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
```

Renderizado para HTML

Apartado en donde requerimos las dependencias para el renderizado de HTML, y el archivo de rutas de nuestro proyecto

```
let engines = require('consolidate');
let routes = require('./app/routes');
```

Configuramos como archivos estáticos lo que tenemos en nuestra carpeta public, y la carpeta de views como archivos de vistas HTML que se mostraran en el navegador. Además de configuraciones de la dependencia consolidate y mustache.

```
app.use('/', routes)
app.use(express.static(__dirname + '/public'));

//renderizar html con las vistas
app.set('views', __dirname + '/views');
app.engine('html', engines.mustache);
app.set('view engine', 'html');
```

Funcionamiento de Socket.io

En este apartado se hacen todas las operaciones de socket.io.

En la primera parte se realizan las declaraciones de variables importantes, y cada una teniendo una función, las cuales son contener valores de las sesiones que se crean dentro de la aplicación, así como variables declarativas de apoyo.

```
io.on('connection', (socket)=> {

    let req =  socket.request;

    const nameBot = 'BotChat';
    let roomId = 0;
    let { usuario, usuarioId, roomName } = req.session;

    console.log('Entre a ' + roomName);
```

Mandamos llamar a la función de socket creada posteriormente de historial, esto para retomar los mensajes anteriores. Así como el uso de join para la conexión a las diferentes salas que tenga el proyecto.

```
    socket.emit('historial');
    socket.join(roomName);
    botTxt('entroSala');
```

Validaciones de datos de usuario para reconocer si se ha iniciado sesión, así como mostrar en consola quien fue el que entro al chat.

```
    if(usuarioId != null){
        connection.query('SELECT * FROM usuarios where  id= ?', [usuarioId],
        (errors, results, fields) => {
            console.log(`Sesion iniciada con el ID_usuario: ${usuarioId} y u
suario ${usuario}`);
            socket.emit('logged_in', {usuario: usuario, sala: roomName});
        });
    }else{
        console.log('No se ha iniciado session');
    }
```

Función de socket de historial, en la cual se realizan llamadas a la base de datos para traer al usuario los datos necesarios de los mensajes anteriores, como lo es en que sala de hizo el mensaje, el usuario que mando el mensaje, así como el mismo mensaje enviado. Esta función la emite a la parte del cliente para que sea procesada y mostrada al usuario.

```
    socket.on('historial', ()=>{
        console.log(`Buscando historial de la sala ${roomName}`);
        connection.query('SELECT * FROM salas where nombre_sala = ?',
        [roomName], (err, result, fields) => {

            let id = -1;
        let resultArray = Object.values(JSON.parse(JSON.stringify(result)));
            resultArray.forEach( v => id = v.id );

            console.log(id)

            console.log(`Buscando historial de la sala ${roomName}`);

            let sql =  `SELECT salas.nombre_sala, usuarios.usuario, mensajes
.mensaje FROM mensajes
                        INNER JOIN salas ON salas.id = mensajes.sala_id
                        INNER JOIN usuarios ON usuarios.id = mensajes.user_i
d
                        WHERE salas.id = ${id}
                        ORDER BY mensajes.id ASC`;

            connection.query(sql, (err, result, fields) => {
                if(err) throw err
                socket.emit('mostrarHistorial', result);
            })
```

```
        })
    });
```

En esta función se reciben los datos mandados por el usuario, lo que se recibe es un mensaje, el identificador del usuario y el identificador de la sala por la cual se envío el mensaje. Esta información se guarda en nuestra base de datos para poder crear un historial simple. Al final emitimos los datos (nombre de usuario, identificador de la sala, y el mensaje) al apartado del cliente, para que este procese estos datos y los muestre de una manera mas clara en el navegador del usuario. No solo se emite a el usuario que envío el mensaje, si no también al resto de usuarios que están conectados en la misma sala.

```
    socket.on('mjsNuevo', (data) => {
        connection.query('SELECT * FROM salas where nombre_sala = ?', [roomN
ame],(err, result, field) => {
            if(!err){
                let sala = result[0].id;
                connection.query('INSERT INTO mensajes( mensaje , user_id, s
ala_id, fecha ) VALUES (?,?,?,CURDATE()) ', [data, usuarioId, sala],
                (error, results, fields) => {
                    if(!error){
                        console.log('Mensaje agregado correctamente');

                        socket.broadcast.to(roomName).emit('mensaje', {
                            usuario: usuario,
                            mensaje: data,
                            roomId: roomId
                        });

                        socket.emit('mensaje', {
                            usuario: usuario,
                            mensaje: data,
                            roomId: roomId
                        });
                    }
                });
            }else{
                console.log(err);
            }

        })

    });
```

En esta función solamente traemos desde la base de datos todas las salas que tenemos almacenadas, para posteriormente enviárselas al cliente para mostrarlas en el navegador.

```
    socket.on('getSalas', (data) => {
        connection.query('SELECT * FROM salas', (err, result, fields) => {
            if(err) throw err
            socket.emit('salas', result);
        });
    })
```

Apartado para recibir qué información del cliente, la cual representa que un usuario realizo un cambio de sala. La información que se recibe es el identificador de la sala y el nombre de esta. Posteriormente se procede a dejar la sala para poder conectarse a la seleccionada desde el navegador. Además, mostramos al usuario un mensaje que se realizo un cambio de sala

```
    socket.on('cambioSala', (data) => {
        const idSala = data.idSala;
        let nombreSala = data.nombreSala;

        socket.leave(roomName);

        roomId = idSala;
        roomName = nombreSala;

        socket.join(roomName),
        botTxt('cambioSala');
    })
```

Función que, a partir de la acción de salir del chat tomada por el cliente, se procede a dejar la sala, así como destruir la sesión que tenia el usuario en el chat. Además, mostramos a los usuarios que se quedaron en la sala un mensaje, el cual menciona que un usuario abandono la sala.

```
    socket.on('salir', (requ, res) => {
        socket.leave(roomName);
        botTxt('seFue')
        req.session.destroy();
    });
```

Acción que se realiza en diferentes ocasiones para mostrar un mensaje al usuario acerca de los estados en el chat, por ejemplo, cuando se entra a una sala, o cuando se realiza el cambio a otra, además, un mensaje cuando un usuario abandona la

sala. Estas acciones se envían al apartado del cliente para poder mostrarlos en el navegador de una forma mas detallada con HTML, CSS.

```
    function botTxt(data){
        entroSala = `Bienvenido a la sala ${roomName}`;
        cambioSala = `Cambiaste a la sala ${roomName}`;
        seFue = `El usuario ${usuario} ha salido de la sala`;

        if(data == "entroSala"){
            socket.emit('mensaje',{
                usuario: nameBot,
                mensaje: entroSala
            })
        }else if(data == 'cambioSala'){
            socket.emit('mensaje',{
                usuario: nameBot,
                mensaje: cambioSala
            })
        }else{
            socket.emit('mensaje',{
                usuario: nameBot,
                mensaje: seFue
            })
        }
    }

})
```

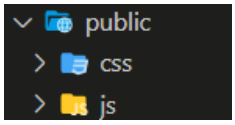
Puerto de funcionamiento de la aplicación

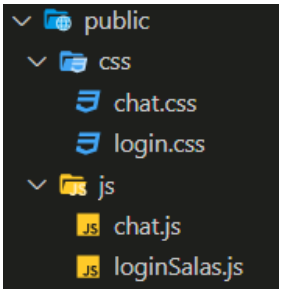
Simple acción para la conexión de la aplicación según sea el puerto al que se conecte o por defecto el puerto 3000. Además, esta acción nos permite arrancar el servicio web de nuestro proyecto.

```
const PORT = process.env.PORT || 3000;
server.listen(PORT, (req, res) => {
    console.log(`Escuchando por el puerto ${PORT}`);
});
```

Directorio public

Se compone de dos subdirectorios importantes para el manejo de las acciones que realiza el usuario los cuales se guardan en la carpeta JS. En cambio, en la de CSS solo se guardan los diferentes archivos de estilos que mejoran la visualización de nuestro HTML.





Directorio JS

Dentro de esta carpeta tenemos dos archivos, el archivo de loginSalas.js administra algunas de las sentencias de la vista de login.

En esta ocasión usamos jQuery para manejar las consultas de una manera mas sencilla. Lo que se hace primeramente es declarar el uso de socket.io en nuestro archivo, para después comenzar como el manejo de las funciones que tenemos en el archivo de index.js, como lo es la función de *'getSalas',* la cual nos permite traer las salas registradas en la base de datos.

```
$(document).ready(function(){
    let socket = io();

    socket.emit('getSalas');
```

En este apartado, recibimos la información que procede del archivo index.js para poder mostrar en el navegador en forma de una lista de opciones que podemos elegir, en este caso, las opciones son las distintas salas a las que nos podemos conectar.

```
    socket.on('salas', (data) => {
        $.each(data, (id, val) => {
            $('#rooms').append($('<option>', {
                value: data[id].nombre_sala,
                text: data[id].nombre_sala,
                id: data[id].id
            }));
        });
    });
})
```

En el archivo de chat.js tenemos mas funciones a realizar por parte del usuario. Primeramente, se declara el uso de socket.io en nuestro archivo para poder usar las distintas funciones creadas en el archivo index.js. Algunas de las principales funciones que usamos son *'getSalas'* para tomar todas las salas registradas en nuestra base de datos y mostrarlas en el navegador, así como *'historial'*, que trae los últimos mensajes que se tenían en la sala.

```
$(document).ready(function(){
    let socket = io();
    let room = 0;

    socket.emit('getSalas');

    socket.emit('historial');
```

La función *'salas'* recibe la información de las salas de nuestra base de datos, para posteriormente mostrarlas como una lista de opciones en nuestro navegador.

```
    socket.on('salas', (data) => {

        $.each(data, (id, val) => {
            $('#roomsCambio').append($('<option>', {
                value: data[id].nombre_sala,
                text: data[id].nombre_sala,
                id: data[id].id
            }));
        });
    });
```

Función que toma los valores de logue, estos son el nombre de usuario y el nombre de la sala a la que se ingresó, y posteriormente se muestra en el navegador.

```
    socket.on('logged_in', (data) => {
        $('#usernameTag').append(data.usuario);
        $('#roomTag').append(data.sala);
        //console.log(data);
    })
```

Esta sección de código sirve como un evento que se dispara cuando se hace un clic dentro del botón enviar mensaje, que como su nombre lo dice envía el mensaje de la caja de texto. Pero antes de enviarlo, se realiza una condición para saber si contiene algo nuestra caja de texto, si la contiene enviamos los datos del mensaje a nuestra función *'mjsNuevo'.*

```
    $('#enviarMensaje').click(() => {
        if($('#mensaje').val().length <= 0){
            alert('Escriba un mensaje');
        }else{
            let mensaje = $('#mensaje').val();
            // console.log(mensaje);
            socket.emit('mjsNuevo', mensaje);
        }
    });
```

La función de ‘mensaje’, recibe los datos de nombre de usuario, el mensaje que se envío y el identificador de sala, para posteriormente poner el mensaje en el navegador, ya sea si es por parte del Bot o de otro usuario.

```
    socket.on('mensaje', ({usuario, mensaje, roomId}) => {
        let nuevoMensaje = ''
        room = roomId;
        if(usuario == 'BotChat'){
            nuevoMensaje = `<li><small><b>${usuario}</b> ${mensaje}</small><
/li>`
        }else{
            nuevoMensaje = `<li> <strong>${usuario}:</strong> ${mensaje} </l
i>`;


        }

        $('#messages').append(nuevoMensaje);
        // window.scrollTo(0, document.body.scrollHeight);
        $('#mensaje').val("");
    });
```

En esta función se realiza la acción cuando se cambia de sala estando dentro de otra, la cual envía los datos de la nueva sala a la que se va a entrar, junto con su identificador. Después de enviar los datos, se cambian los datos de la sala, así como la recuperación del historial de mensajes de esta.

```
    $('#roomsCambio').change(() => {
        let roomId = room;
        roomName = $(this).find('option:selected').text();
        $('#messages').empty();

        socket.emit('cambioSala', {
            idSala: roomId,
            nombreSala: roomName
        })
```

```
        console.log('Cambio Select ID: ' + roomId + ' con nombre: ' + roomNa
me);
        $('#roomTag').empty();
        $('#roomTag').append(roomName);
        socket.emit('historial');
    });
```

Este código recibe los datos de todos los mensajes que tiene guardada nuestra base de datos según su sala, y los muestra en el navegador, esto junto con un mensaje del chat Bot para decir que se recuperaron los datos de los mensajes anteriores.

```
    socket.on('mostrarHistorial', (data) => {
        let mensajes = '';
        $.each(data, (id, val) => {
            mensajes += `<li><small><b>${data[id]['usuario']}</b> ${data[id]
['mensaje']}</small></li>`
        })

        mensajes += `<li style="background: #fffff"><small><b>BotChat</b> Ul
timos mensajes del historial de la sala</small></li>`
        $('#messages').append(mensajes + '</br>');
    })
```

Esta sección se activa cuando el usuario quiere salir de la sala y del chat, por lo cual se manda a llamar la función salir de nuestro index.js.

```
    $('.logout').click(() => {
        socket.emit('salir')
    });
})
```

Directorio app

Directorio controllers

Los archivos en este directorio se encargan de controlar todas las acciones que realizan los usuarios dependiendo el endpoint que se coloca en la URL.

chatController.js – En este archivo únicamente se hace una exportación del endpoint de chat, el cual únicamente renderiza la vista de chat.

```
module.exports = {
    chat: (req, res) => {
        res.render('chat');
    }
};
```

loginController.js – En este archivo se exportan las acciones a realizar en login, además de requerir la conexión a la base de datos. En la primera acción llamada index, podemos observar que únicamente renderiza la vista login.

```
let connection = require("../../db/mysql");

module.exports = {
    index: (req, res) => {
        res.render('login');
    },
```

En la acción de *auth*, se hace una validación inicial para saber si se mandó algún valor desde los campos de texto de login, después procede a realizar una llamada a la base de datos para traer a todos los usuarios que tengan el mismo nombre y contraseña que el agregamos, si existe el usuario que quiere entrar procedemos a llenar nuestra sesión del chat con los datos de este usuario, para después redirigirlo al enpoint de home. Si no existe el usuario se renderiza la vista de datos inválidos, y si es que no coloco todos los datos se muestra la pagina de falta de datos.

```
    auth: (req, res) => {
        let usuario  = req.body.usuario;
        let pw       = req.body.pw;
        let room     = req.body.rooms;

        if((usuario && pw) && room != 'null' ){
            connection.query('SELECT * FROM usuarios WHERE usuario = ? AND p
w = ?', [usuario, pw],
            (error, results, fields) => {
                if(results.length > 0){
```

```
                    req.session.loggedin = true;
                    req.session.usuario = usuario;
                    req.session.usuarioId = results[0].id;
                    req.session.roomName = room;
                    res.redirect('/home');
                }else{
                    res.render('invalidData');
                }
            });
        }else{
            res.render('faltanDatos');
        }
    },
```

Este endpoint renderiza la vista de chat, si es que el usuario se ha colocado sus datos correctamente, si no pide que se vuelva a iniciar sesión con la una vista nueva.

```
    home: (req, res) => {
        if(req.session.loggedin){
            res.render('chat');
        }else{
            res.render('iniciarSesionNuevamente');
        }
    },
```

Registrar es un enpoint que como su nombre lo dice se encarga de registrar al usuario. Las funciones que realiza son parecidas al de login, la única diferencia después de validar que el usuario coloco todos los datos, se valida que no exista un usuario con los mismos datos que se acaban de ingresar, después se procede a guardar en la base de datos el usuario registrado, mostrando enseguida un mensaje para ir a iniciar sesión con el usuario agregado.

```
    registrar: (req, res) => {
        let email = req.body.email;
        let usuario = req.body.usuario;
        let pw = req.body.pw;
        if(usuario && pw && email){
            connection.query('SELECT * FROM usuarios WHERE usuario = ? OR em
ail = ?', [usuario, email],
            (error, results, fields) => {
                if(results.length != 0){
                    let usuariosDB = results.map( data => {
                        return data.usuario;
                    });
```

```
                    let emailsDB = results.map(data => {
                        return data.email;
                    });

                    let usuarioRepetido = '';
                    let emailRepetido = '';

                    for(let i = 0; i < results.length; i++){ //0
                        if(usuariosDB[i] == usuario && emailsDB[i] == email)
{
                            res.send(`
                                Ya existe un usuario con email <strong>${ema
il}</strong> y con usuario <strong>${usuario}</strong>
                                <br><br>
                                <a href='/registro'>Volver a registro</a>
                                <br><br>
                                <a href='/'>Volver a inicar sesion</a>
                            `);

                        }else{

                            if(emailsDB[i] == email){
                                console.log('email -> ' +emailsDB[i]);
                                emailRepetido = emailsDB[i];
                                break;
                            }else{
                                console.log('user -> ' +usuariosDB[i]);
                                usuarioRepetido = usuariosDB[i];
                                break;
                            }
                        }
                    }

                    if(usuarioRepetido != ''){

                        res.send(`
                            Ya existe un usuario con el usuario <strong>${us
uario}</strong>
                            <br><br>
                            <a href='/registro'>Volver a registro</a>
                            <br><br>
                            <a href='/'>Volver a inicar sesion</a>
                        `);
                    }
```

```
                    if(emailRepetido != ''){
                        res.send(`
                Ya existe un usuario con el email <strong>${email}</strong>
                            <br><br>
                            <a href='/registro'>Volver a registro</a>
                            <br><br>
                            <a href='/'>Volver a inicar sesion</a>
                        `);
                    }
                }else{
                    connection.query('INSERT INTO usuarios (usuario, pw, ema
il) VALUES (?,?,?)',[usuario, pw, email],
                    (err, rows, fiel) => {
                        if(err){
                            res.send(
                                `Error al registrar
                                <br><br>
                                <a href='/registro'>Volver a registro</a>
                                <br><br>
                                <a href='/'>Volver a inicar sesion</a>
                                `);
                        }else{
                            res.send(
                                `Usuario registrado
                                <a href='/'>Inicar sesion</a>
                            `);
                        }
                    });
                }
            });
        }else{
            res.send('Favor de ingresar usuario y contraseña');
            res.end();
        }
    }
}
```

registroController.js – En este archivo únicamente se renderiza el archivo de HTML de registro.

```
module.exports = {
    index: (req, res) => {
        res.render('registro');
    },
};
```

Directorio routes

En cada uno de los archivos de muestran los métodos usados para cada enpoint, así como el requerimiento de la instrucción router de express, además de llamar al controlador de cada uno, ya sea para el chat, login, o registro. También cada uno de estos archivos se debe exportar para su uso en otras partes del código.

Chat.js - Solo tenemos un método GET, que se usa para mandar llamar al endpoint de chat.

```
//Controlador y administrador de rutas
let router = require('express').Router();
let chatControlador = require('../controllers/chatController');

//GET
router.get('/', (req,res) =>{
    chatControlador.chat(req, res);
});

module.exports = router;
```

Index.js – En este archivo requerimos el uso de express-session para poder manejar correctamente las sesiones de cada usuario. Este es el código raíz de las rutas, pues en este se requieren los archivos de las rutas de chat, login, registro para poder redireccionar dependiendo el enpoint.

```
let router = require('express').Router();

//Requerimos las rutas de los archivos
let chat = require('./chat');
let login = require('./login');
let registro = require('./registro');

let session = require("express-session");

router.use(
    session({
      secret: "secret",
      resave: true,
      saveUninitialized: true,
    })
);

//usamos las rutas
```

```
router.use('/', login);
router.use('/registro',registro);
router.use('/chat', chat);

module.exports = router;
```

Login.js – Se hace uso de las sesiones de express para su mejor manejo en los controladores. Uso de dos métodos GET (index = vista login, home = vista chat), y dos POST (autenticación y registro).

```
let router = require('express').Router();
let loginControlador = require('../controllers/loginController');
let session = require("express-session");

router.use(
    session({
      secret: "secret",
      resave: true,
      saveUninitialized: true,
    })
);

//GET
router.get('/', (req, res) => {
    loginControlador.index(req, res);
});

router.get('/home', (req, res) => {
    loginControlador.home(req, res);
})

//POST
router.post('/auth', (req, res) => {
    loginControlador.auth(req, res);
})

router.post('/register', (req, res) => {
    loginControlador.registrar(req, res);
})

module.exports = router;
```

Registro.js – Uso de un método GET, para el renderizado de la vista de registro.

```
let router = require('express').Router();
```

```
let registroControlador = require('../controllers/registroController');

router.get("/", (req, res) => {
    registroControlador.index(req, res);
});

module.exports = router;
```

Conclusión

El proyecto resultó muy interesante para nuestro crecimiento educativo en el campo tecnológico, ya que se aplicaron importantes tecnologías a lo largo de su desarrollo. También nos ayudó a comprender lo sencillo que es utilizar este enfoque de WebSockets en distintos proyectos, pues esta tecnología no solo aplica para realizar chat en tiempo real, sino que también se pueden crear aplicaciones más complejas. Además, en la parte personal, me enseñó a estructurar de mejor manera mis programas, pues ayuda a saber en donde se encuentran errores en el código, así como ubicar las distintas funcionalidades de cada archivo.

Las diferentes cosas interesantes que se puedes hacer con JavaScript y el conocimiento básico de Node.js son muy extensas, solo basta paciencia, lógica y la idea de lo que se esta haciendo para completar correctamente cualquier proyecto que uno se pueda plantear.

Glosario

WebSockets: es una tecnología avanzada que hace posible abrir una sesión de comunicación interactiva entre el navegador del usuario y un servidor.

TCP: TCP (Protocolo de Control de Transmisión, por sus siglas en inglés Transmission Control Protocol) es protocolo de red importante que permite que dos anfitriones (hosts) se conecten e intercambien flujos de datos.

HTTP: Hypertext Transfer Protocol (HTTP) (o Protocolo de Transferencia de Hipertexto en español) es un protocolo de la capa de aplicación para la transmisión de documentos hipermedia, como HTML.

API: es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. API significa interfaz de programación de aplicaciones.

Middleware: se refiere a un sistema de software que ofrece servicios y funciones comunes para las aplicaciones. En general, el middleware se encarga de las tareas de gestión de datos, servicios de aplicaciones, mensajería, autenticación y gestión de API.

Cookies: es un archivo creado por un sitio web que contiene pequeñas cantidades de datos y que se envían entre un emisor y un receptor. Su propósito principal es identificar al usuario almacenando su historial de actividad en un sitio web específico, de manera que se le pueda ofrecer el contenido más apropiado según sus hábitos.

Hash: es un algoritmo matemático que transforma cualquier bloque arbitrario de datos en una nueva serie de caracteres con una longitud fija. Independientemente

Open source: es un código diseñado de manera que sea accesible al público: todos pueden ver, modificar y distribuir el código de la forma que consideren conveniente.

Json: es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web

JQuery: es una JavaScript Library que se enfoca en simplificar la manipulación del DOM, llamadas AJAX y manejo de Event. Es utilizado por desarrolladores JavaScript de manera frecuente.

Bot: es un programa de software que opera en Internet y lleva a cabo tareas repetitivas.

Chat Bot: asistente que se comunica con los usuarios a través de mensajes de texto. Se trata de una tecnología que permite al usuario mantener una conversación a través de un software que se integra en un determinado sistema de mensajería, como, por ejemplo: Facebook, Twitter, Telegram, WhatsApp, etc.

URL: URL significa Uniform Resource Locator (Localizador de Recursos Uniforme). Una URL no es más que una dirección que es dada a un recurso único en la Web.

EndPoint: es un extremo de un canal de comunicación. Cuando una API interactúa con otro sistema, los puntos de contacto de esta comunicación se consideran endpoints.

Renderizado: En una página web el renderizado de la página ocurre cuando se visita y su contenido se pinta en la pantalla.