Universidad Autónoma de Querétaro

Facultad de Informática

Ingeniería de Software (SOF18)

Sistemas Distribuidos

Maestro: Carlos Alberto Olmos Trejo

“Manual Técnico”

Integrantes:

- García Vargas Michell Alejandro – 259663
- León Paulin Daniel – 260541

Semestre: 5° **Grupo:** 30

Fecha: 10 de diciembre del 2021

Índice

Manual Técnico

Este manual describe el desarrollo de la API (Aplication Program Interface), las funcionalidades, ¿cómo es que funciona la API?, ¿qué es lo que realiza el código en cada caso concreto? Y ¿cómo se desarrolló la API?

Código Fuente de toda la API: https://github.com/AleGV258/ExpressChat.git

Requerimientos mínimos de Hardware y Software:

- **Procesador:** Intel Celeron (o superior), o AMD Athlon (o superior).
- **Memoria RAM:** 2 Gigabytes (GB).
- **Disco Duro o Estado Sólido:** HDD (Hard Disk Drive) de 30 Gigabytes, o SSD (Solid State Disk) de 30 Gigabytes.
- **Privilegios:** Contar con privilegio de Administrador.
- **Sistema Operativo:** Windows 7 (o superior), Linux o MacOS.
- **Navegador:** Chrome, Firefox, Edge y Opera.

Tecnologías y herramientas utilizadas para el desarrollo:

- **Lenguajes:**
  - JavaScript
  - CSS3
- **Frameworks:**
  - Node.js
- **Librerías/Dependencias:**
  - SQLite3
  - Express.js
  - EJS (Embebed JavaScript)
  - Express-Session

Inicialización del proyecto:

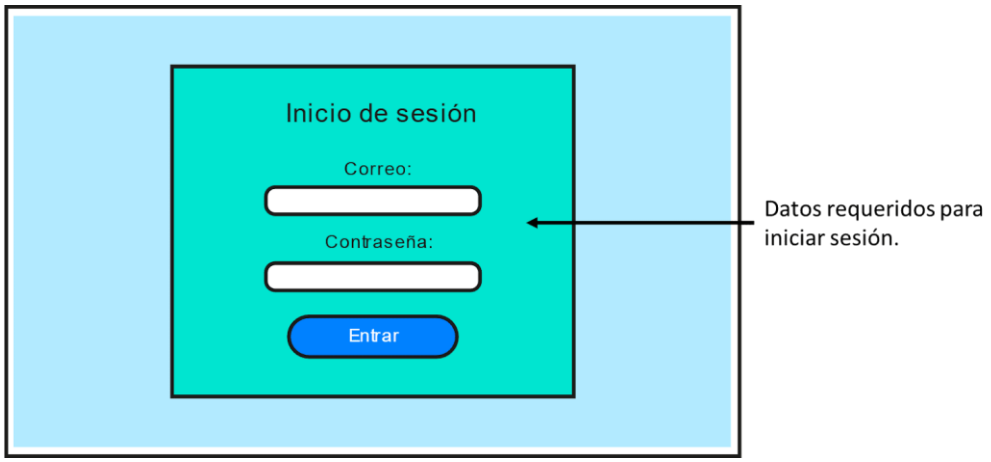
Para la creación del proyecto, primeramente, se debe tener ya instalado y en el path Node.js, una vez que se tenga instalado, se debe abrir la consola de comandos y dirigir al directorio donde se va a desarrollar la API.

Una vez realizado lo anterior, se debe inicializar el proyecto con el comando *"npm init -y"*, posteriormente el comando *"npm install"*, después el comando *"npm install express"*, después *"npm install sqlite3"*, después *"npm install ejs"*, por último, el comando *"npm install express-session"*. Una vez ejecutados estos comandos, en el directorio raíz se debieron de haber creado, una carpeta de módulos de node, y dos archivos .json con la nomenclatura "package".

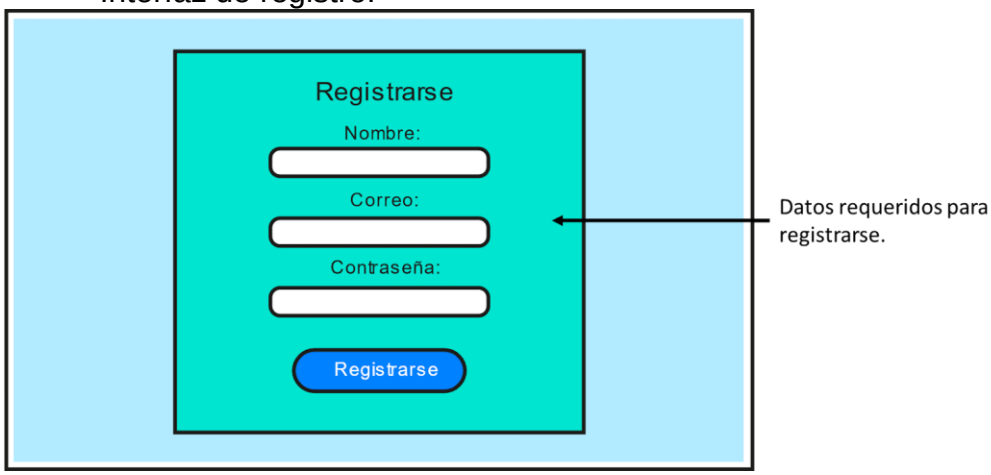
Ejecutando los comandos anteriores ya podemos comenzar con la construcción y escritura del código para generar la API, pero primero, debemos definir de manera correcta la estructura del proyecto para conocer cómo es que se debe construir, su diseño arquitectónico y de clases del proyecto, para ello debemos crear los modelos de alto nivel de las interfaces del proyecto, el modelo relacional de la base de datos y para comprender como es que funciona, un diagrama de secuencia.

Interfaces, Modelo de Alto Nivel:

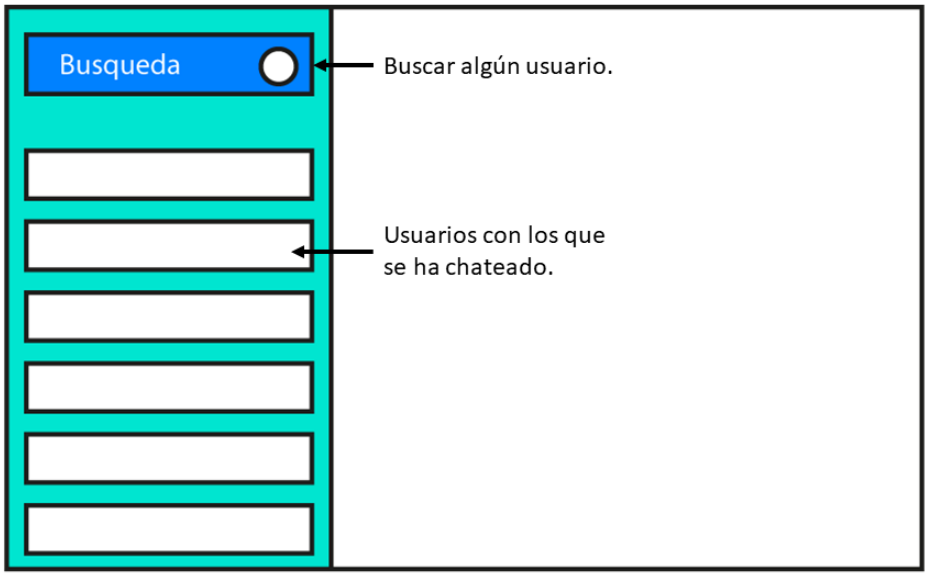
- Interfaz de inicio de sesión:



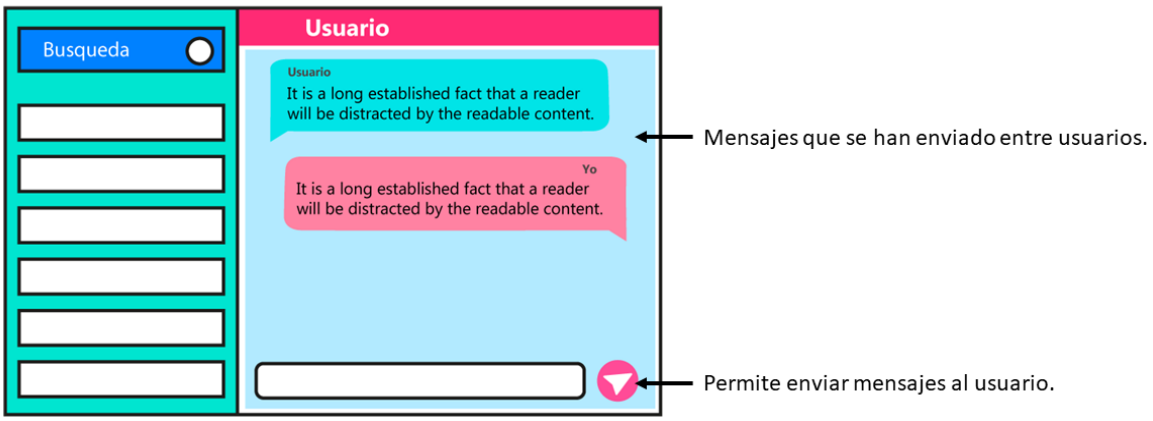
- Interfaz de registro:



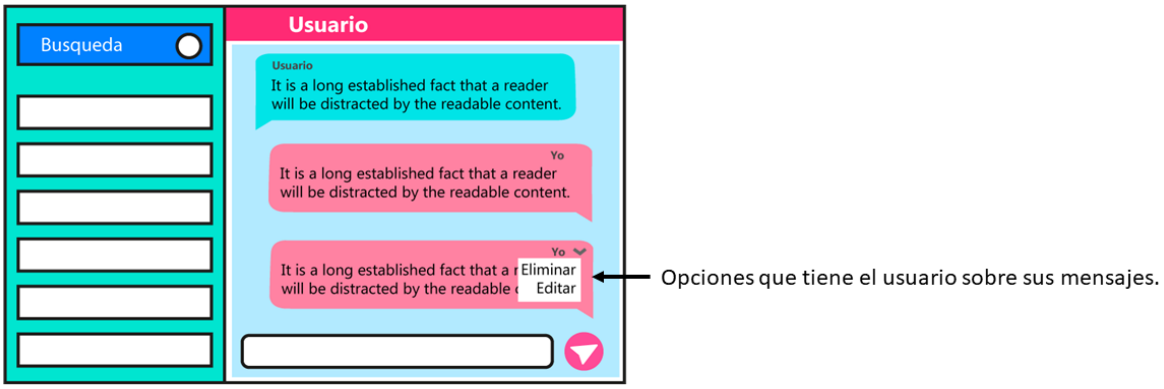
- Interfaz de chat:



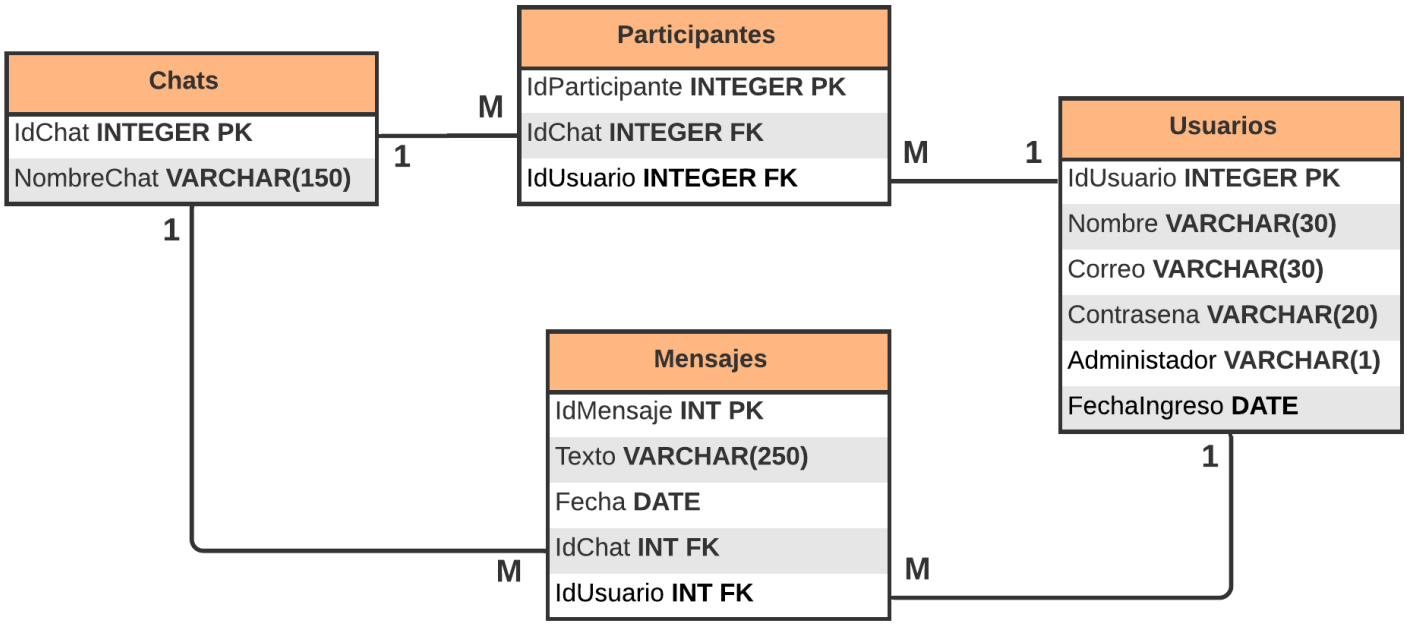
- Interfaz del chat especifico:



- Opciones de un mensaje:

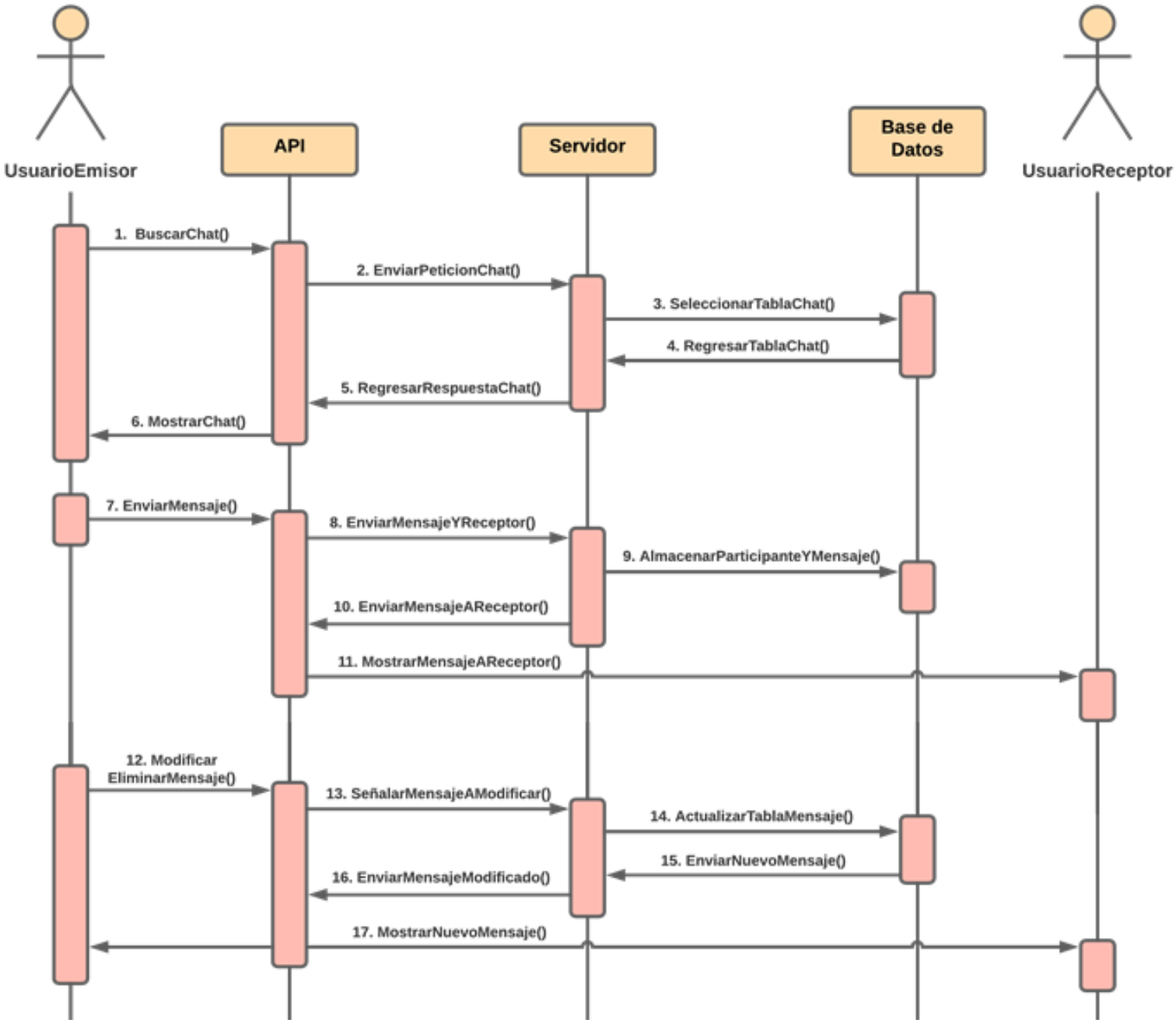


Modelo Relacional de la Base de Datos:



Dentro de este modelo, se crean cuatro tablas principales, “Chats”, que hace referencia a todos los chats donde los usuarios envían mensajes, “Usuarios”, que hace referencia a los usuarios que pueden mandar mensajes, “Mensajes”, que hace referencia al texto que se comunica entre usuarios, y por último, “Participantes”, que hace referencia a los espacios que ocupa cada usuario dentro de un chat, donde un chat puede tener muchos participantes y un usuario puede estar en muchos chats.

Diagrama de Secuencia:



Directorios del Proyecto:
Dentro del directorio raíz del proyecto, donde se desarrolla el proyecto, en el mismo nivel donde se instalaron los módulos de node y los archivos .json, se deben crear dos subdirectorios, denominados “public” y “views”, estos subdirectorios son las carpetas preestablecidas, para que la librería de archivos EJS puedan trabajar con los archivos generados para la API, donde en “views”, se almacenan todos los archivos dinámicos que tienen código embebido de javascript, y en donde en “public”, se almacenan todos los archivos estáticos del proyecto.

Dentro del subdirectorio “public”, se almacenan en subdirectorios, el archivo de la base de datos, el archivo de estilos de la API, las imágenes y las fuentes que la API ocupa, dentro del subdirectorio “database”, donde se almacena la base de datos, de igual forma se almacena la aplicación .exe de sqlite3, la cual ayuda a la librería a ejecutar de forma correcta la base de datos. Y por último, al mismo nivel que los módulos de node, los subdirectorios “views”, “public” y los archivos .json, se debe crear un archivo .js (javascript), donde se desarrolla todo el código del lado del servidor, entonces el directorio completo quedaría tal que:

- ExpressChat
  - node_modules
    - (modulos de node…)
  - public
    - database
      - (archivo de la base de datos (.db)…)
      - (aplicación de sqlite3)
    - fonts
      - (fuentes utilizadas en el diseño…)
    - images
      - (imágenes utilizadas dentro de la API…)
    - styles
      - (archivo de estilos de los archivos EJS (.css)…)
  - views
    - (archivos EJS dinámicos (.ejs)…)
  - package-lock.json
  - package.json
  - servidor.js

Generación de la Base de Datos:
Para poder comenzar con el desarrollo de la API, primero se debe generar la base de datos, para esto, se debe dirigir al subdirectorio donde se almacena la base de datos, en este caso [../ExpressChat/public/database/], dentro de la consola de comandos, y ejecutar el comando *“sqlite3 [nombre_de_la_base_datos].db”*, al ejecutar el comando, se debe abrir otra sección en la consola, donde se puede observar que es una sección específica para sqlite3, donde se pueden ejecutar comandos SQL, gracias a esto es que podemos generar las tablas del modelo relacional y rellenarlas con unos datos preestablecidos, los comandos a ejecutar son:

```
/* Como es SQLite, por default las llaves foráneas no están activadas para poder relacionarse entre sí, por ende, hay que activarlas */
PRAGMA foreign_keys = ON;

/* Se crea la tabla chatss */
CREATE TABLE Chats(
	IdChat INTEGER PRIMARY KEY AUTOINCREMENT,
	NombreChat VARCHAR(150) NOT NULL UNIQUE
);

/* Se crea la table usuarios */
CREATE TABLE Usuarios(
	IdUsuario INTEGER PRIMARY KEY AUTOINCREMENT,
	Nombre VARCHAR(30) NOT NULL UNIQUE,
	Correo VARCHAR(30) NOT NULL UNIQUE,
	Contrasena VARCHAR(20) NOT NULL,
	FechaIngreso DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
	Administrador VARCHAR(1)
);

/* Se crea la table participantes */
CREATE TABLE Participantes(
	IdParticipante INTEGER PRIMARY KEY AUTOINCREMENT,
	IdChat INT NOT NULL,
	IdUsuario INT NOT NULL,
	FOREIGN KEY (IdChat)
		REFERENCES Chats (IdChat)
	FOREIGN KEY (IdUsuario)
		REFERENCES Usuarios (IdUsuario)
);

/* Se crea la table mensajes */
CREATE TABLE Mensajes(
	IdMensaje INTEGER PRIMARY KEY AUTOINCREMENT,
	Texto VARCHAR(250) NOT NULL,
	Fecha DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
	IdChat INT NOT NULL,
	IdUsuario INT NOT NULL,
	FOREIGN KEY (IdChat)
		REFERENCES Chats (IdChat)
	FOREIGN KEY (IdUsuario)
		REFERENCES Usuarios (IdUsuario)
);

/* Se insertan unos chats dentro de la tabla chats */
INSERT INTO Chats (NombreChat) VALUES('Grupo30');
INSERT INTO Chats (NombreChat) VALUES('Fornite');
INSERT INTO Chats (NombreChat) VALUES('Tareas');
```

```
/* Se insertan unos usuarios en la tabla usuarios */
INSERT INTO Usuarios (Nombre, Correo, Contrasena, Administrador) VALUES('Michell', 'michell@gmail.com', 'michell1234', 'A');
INSERT INTO Usuarios (Nombre, Correo, Contrasena, Administrador) VALUES('Daniel', 'daniel@gmail.com', 'daniel1234', 'A');
INSERT INTO Usuarios (Nombre, Correo, Contrasena, Administrador) VALUES('Carlos', 'carlos@outlook.com', 'carlos1234', 'U');
INSERT INTO Usuarios (Nombre, Correo, Contrasena, Administrador) VALUES('Perla', 'perla@alumnos.uaq.mx', 'perla1234', 'U');
INSERT INTO Usuarios (Nombre, Correo, Contrasena, Administrador) VALUES('Ana', 'ana@gmail.com', 'ana1234', 'U');

/* Se insertan unos participantes en la tabla participantes */
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(3, 1);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(3, 2);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(2, 1);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(2, 2);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(2, 3);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(1, 1);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(1, 2);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(1, 3);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(1, 4);
INSERT INTO Participantes (IdChat, IdUsuario) VALUES(1, 5);

/* Se insertan unos mensajes en la tabla mensajes */
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('Hola, oigan jugamos unas partidas hoy?', 2, 3);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('Yo no puedo, voy a salir :p', 2, 1);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('yo si, dame 5 minutos y me conecto', 2, 2);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('Oye que falta para el proyecto de sistemas distribuidos', 3, 1);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('la interfaz', 3, 2);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('okay vale', 3, 1);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('oigan que habia de tarea para mañana', 1, 4);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('pues lo de ingles', 1, 3);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('sip, hacer un ensayo de un libro en inglés', 1, 2);
INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES('vale, muchas gracias', 1, 4);
```

Creación de tablas de la Base de Datos:

Primeramente, se creó la tabla “Chats”, donde se tiene un id y el nombre del chat en cual no puede ser nulo, ni se puede repetir. Después en la tabla “Usuarios”, se tiene un id, el nombre del usuario, su correo, los cuales no son nulos, ni se pueden repetir, contiene su contraseña la cual no puede ser nula, contiene su fecha de ingreso, la cual toma como default el tiempo actual y no puede ser nulo, por último, tiene un atributo de administrador, el cual puede ser nulo, y si es así, se toma como que el usuario no es administrador. Después en la tabla “Participantes”, contiene un

id, el id del usuario y el id del chat, ambos no pueden ser nulos, esto es para relacionar los usuarios con un chat, y cada uno de estos id de participantes, representa una unión de un usuario con un chat. Por último, en la tabla “Mensajes”, contiene un id de un mensaje, el texto del mensaje, el cual no puede ser nulo, la fecha en la que se envió el mensaje, el cual no puede ser nulo, y tomo por default la fecha actual, por último, el id del usuario que envía el mensaje y el id del chat hacia donde es dirigido.

Configuraciones del Servidor:

Ahora se dará paso a explicar las configuraciones básicas del servidor para que pueda ser utilizado y generar las funciones dentro de la API. Primero en diversas constantes denominadas de cualquier forma, se importa la librería de express, body parser, sqlite3 y express sesión. A las variables se les asigna la importación de los módulos con el siguiente código, “require(‘express’)”, “require(‘body-parser’)”, “require(‘sqlite3’)” y “require(‘express-session’)”, después una vez importado express, se debe crear una nueva aplicación de tipo express y asignarlo a una variable, para poder trabajar todas las funciones dentro de esta nueva aplicación.

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const sqlite3 = require('sqlite3');
const session = require('express-session');
```

Una vez configuradas las importaciones, se debe hacer que el servidor inicie al conectarse a localhost en un puerto e inicializar el servidor, para esto, a la aplicación creada de express se le debe colocar el método “.listen()”, donde el primer parámetro dentro del método es el puerto donde se levanta el servidor para esperar por conexiones, después, se puede agregar código extra para imprimir en consola cualquier mensaje que se desee enviar al servidor, cuando este ya está en funcionamiento.

```
app.listen(5000, () => { console.log('Servidor Web Iniciado'); });
```

Posterior a esta configuración, se deben colocar mas configuraciones para poder utilizar de forma correcta los directorios del proyecto y sus sesiones, primero se configura con el método “.use()”, el directorio local de archivos estáticos (static) del servidor, esta configuración es importante, ya que gracias a esto el servidor va a poder buscar los archivos estáticos en esta ruta.

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

Una vez realizado lo anterior, con el método “.set()”, se configura el middleware que se va a utilizar para los archivos dinámicos, en este caso los EJS, gracias a este método es que el servidor puede renderizar de forma correcta los archivos EJS al momento de mandarlos a llamar por la url, dentro de este método se especifica, donde se encuentran los archivos dinámicos, en caso de no encontrarlos en el subdirectorio público, además, se coloca la extensión de los archivos, que debe encontrar para poder utilizar y renderizar dentro de la API.

```
app.set('view engine', 'ejs');
```

Ahora, con el método ".use()" nuevamente, establecemos los parámetros utilizados para las sesiones de express, donde se establece el id secreto utilizado para firmar las sesiones que se inicialicen en el servidor, no se guardan los cambios en las sesiones de express, ya que la API planeada almacena los cambios realizados en la base de datos.

```
app.use(session({
  secret: 'ProySisDist123',
  resave: false,
  saveUninitialized: false,
}));
```

Por ultimo, se debe inicializar la base de datos, para esto, se debe en una variable crear una nueva base de datos, donde como parámetros se le especifica el directorio donde se encuentra la base de datos anteriormente creada, en caso de que se encuentre se regresa un mensaje correcto, en caso contrario se regresa un error y un mensaje de que no se pudo establecer conexión.

```
const db = new sqlite3.Database("./public/database/expressDB.db", (err) => {
  if (err) {
    console.log('No se puede conectar a la base de datos\n');
    console.log(err)
  } else {
    console.log('Conectado a la base de datos\n');
  }
});
```

Funciones generadas para peticiones en el Servidor:

Ahora bien, una vez generadas las configuraciones para que el servidor y express puedan funcionar correctamente, se deben generar las funciones que regresan una respuesta, en base a una petición hecha al servidor. Primero la petición de la página inicial, o sin hacer ninguna petición la url "/", esta función se escribe con el método ".get()", debido a que no hay ningún parámetro dentro de la url, recordando que el método get del protocolo HTTP, permite que los clientes puedan visualizar los parámetros enviados por url, y por ende cambiarlos. Esta función renderiza al cliente el archivo index.ejs, que es la página inicial o login de la API.

```
app.get('/', (req, res) => {
  res.render('index.ejs', { validacion: 'N' });
});
```

Después, tenemos la función, de la petición "/Registro.ejs", la cual esta conformada por el método ".get()" y únicamente renderiza la página Registro.ejs.

```
app.get('/Registro.ejs', (req, res) => {
  res.render('Registro.ejs', { validacion: 'N' });
});
```

Ahora, tenemos la función de la petición "/Logout", la cual esta conformada por el método ".get()" y renderiza la página principal o índex, regresa un status correcto de petición y destruye la sesión creada al ingresar a la API.

```
app.get('/Logout', function (req, res) {
  req.session.destroy();
  res.status(200);
  res.redirect('/');
});
```

Después, tenemos la función de la petición “/Login”, la cual esta construida con el método “.post()”, ya que los datos a mandar por la url son sensibles y si son robados puede representar un problema para el cliente, recordando que el método post del protocolo HTTP, no son visibles para el usuario, ni en el transporte de esta información, esta función valida que los datos proporcionados por el usuario en el formulario de index.ejs, se encuentren en la base de datos, en la tabla usuario, sean correctos y coincidan, en caso de que esto sea correcto se le renderiza al usuario la petición “/ExpressChat”, se manda un estatus correcto (200) como respuesta y se almacenan los datos del usuario en la sesión del mismo, esto con la finalidad de ocuparlos en otras funciones, de lo contrario, se le renderiza al usuario la pagina inicial con un estatus incorrecto (400), para que el usuario pueda validar nuevamente sus credenciales.

```
app.post('/Login', (req, res) => {
  let correo = req.body.correo;
  let contrasena = req.body.contrasena;
  var idusuario;
  var admin;
  var nombre
  sql = 'SELECT * FROM Usuarios WHERE Correo = ?;';
  db.get(sql, [correo], (err, row) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      if (typeof row === 'undefined') {
        res.status(400);
        res.render('index.ejs', { validacion: 'I' })
      } else {
        if (correo == row.Correo && contrasena == row.Contrasena) {
          idusuario = row.IdUsuario;
          admin = row.Administrador;
          nombre = row.Nombre;
          req.session.id_Usuario = idusuario
          req.session.admon = admin;
          req.session.nombre = nombre;
          res.status(200);
          res.redirect('/ExpressChat');
        } else {
          res.status(400);
          res.render('index.ejs', { validacion: 'I' })
        }
      }
    }
  });
});
```

Después, tenemos la función de la petición “/ExpressChat”, la cual está construida con el método “.get()”, esta función únicamente busca en la base de datos todos los chats pertenecientes al usuario que ingreso correctamente sus credenciales, en caso de que sea correcto, la función renderiza la página Chat.ejs, con un parámetro, las cuales son los chats que pertenecen al usuario, y estos los muestra en la página del Chat.ejs, con un estatus correcto (200), en caso contrario, regresa un error con el estatus incorrecto (400).

```
app.get('/ExpressChat', (req, res) => {
  var idUsuarioActual = req.session.id_Usuario;
  sql = 'SELECT * FROM Chats WHERE IdChat IN (SELECT IdChat FROM
Participantes WHERE IdUsuario = ?);';
  db.all(sql, [idUsuarioActual], (err, rows) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.render('Chat.ejs', { chats: rows });
    }
  });
});
```

Después, tenemos la función de la petición “/registro”, la cual está construida con el método “.post()”, esta función se encarga de insertar en la base de datos, en la tabla usuarios, los datos proporcionados por el usuario en el formulario de Registro.ejs, con el estatus correcto (200), en caso contrario se renderiza nuevamente Registro.ejs, con parámetro y estatus incorrecto (400).

```
app.post('/registro', (req, res) => {
  var reqBody = req.body;
  db.run('INSERT INTO Usuarios (Nombre, Correo, Contrasena,
Administrador) VALUES(?, ?, ?, ?);', [reqBody.nombre, reqBody.correo,
reqBody.contrasena, 'U'], (err, result) => {
    if (err) {
      res.status(400);
      res.render('Registro.ejs', { validacion: 'I' });
      return;
    }else{
      res.status(200);
      res.render('Registro.ejs', { validacion: 'C' });
    }
  });
});
```

Después, tenemos la función de la petición “/resultadosBusqueda”, la cual está construida con el método “.get()”, esta función se encarga de buscar en la base de datos, la información proporcionada por el usuario al buscar a otro usuario, en la tabla usuarios, si los resultados coinciden, se renderiza la página resultadosBusqueda.ejs con parámetros, los cuales son, los usuarios encontrados en la base de datos que coinciden con la búsqueda, estos se despliegan en la misma página, con el estatus correcto (200), en caso contrario se regresa un error, con el estatus incorrecto (400).

```
app.get('/resultadosBusqueda', (req, res) => {
  let usuario = req.query.usuario;
  var admin = req.session.admon
  var IdUsuarioActual = req.session.id_Usuario;
  sql = 'SELECT * FROM Usuarios WHERE Nombre LIKE ? AND IdUsuario <> ?;';
  db.all(sql, ['%' + usuario + '%', IdUsuarioActual], (err, rows) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.render('resultadosBusqueda.ejs', { users: rows, usuario:
usuario, administrador: admin });
    }
  });
});
```

Después, tenemos la función de la petición “/NuevoChat/:idUsuario/:nombre”, la cual está construida con el método “.get()”, esta función se encarga de buscar si el usuario en cuestión ya tiene un chat llamado de una especifica forma, en caso de que sea correcto, lo renderiza al usuario con la petición “/Chat/:idChat”, en caso contrario, la función se encarga de insertar en la tabla chats, el nuevo chat proporcionado por el usuario, se crea, y se inserta en la tabla participantes la relación del usuario actual con el chat nuevo creado, y después, lo renderiza al usuario con la petición “/Chat/:idChat” con el estatus correcto (200), en caso contrario, regresa un error con el estatus incorrecto (400).

```
app.get('/NuevoChat/:idUsuario/:nombre', function (req, res) {
  var idUsuario = req.params.idUsuario;
  var nombre = req.params.nombre;
  var nombreUsuarioActual = req.session.nombre;
  var idUsuarioActual = req.session.id_Usuario;
  var nombreChat = nombreUsuarioActual + ' - ' + nombre;
  var nombreChat2 = nombre + ' - ' + nombreUsuarioActual;
  db.get('SELECT * FROM Chats WHERE NombreChat = ? OR NombreChat = ?
ORDER BY IdChat ASC LIMIT 1;', [nombreChat, nombreChat2], (err, row) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      if (typeof row === 'undefined') {
        db.run('INSERT INTO Chats (IdChat, NombreChat) VALUES(?, ?);', [,
nombreChat], (err, result) => {

          if (err) {
            res.status(400).json({ "error": err.message });
            return;
          } else {
            db.get('SELECT * FROM Chats WHERE NombreChat = ? ORDER BY
IdChat ASC LIMIT 1;', [nombreChat], (err, row) => {
              if (err) {
                res.status(400).json({ "error": err.message });
                return;
              } else {
```

```
                db.run('INSERT INTO Participantes (IdChat, IdUsuario)
VALUES (?, ?);', [row.IdChat, idUsuario], (err, result2) => {

                  if (err) {
                    res.status(400).json({ "error": err.message });
                    return;
                  }
                });
                db.run('INSERT INTO Participantes (IdChat, IdUsuario)
VALUES (?, ?);', [row.IdChat, idUsuarioActual], (err, result2) => {
                  if (err) {
                    res.status(400).json({ "error": err.message });
                    return;
                  }
                })
                direccion = '/Chat/' + row.IdChat;
                res.redirect(direccion);
              }
            });
          }
        });
      } else {
        direccion = '/Chat/' + row.IdChat;
        res.redirect(direccion);
      }
    }
  });
});
```

Después, tenemos la función de la petición “/Chat/:idChat”, la cual está construida con el método “.get()”, esta función se encarga de seleccionar el chat, seleccionado por un usuario, posteriormente comprueba que este chat, exista en la relación de la tabla participantes, si no es así, no se le renderiza nada al usuario, en caso de que si exista la relación, se seleccionan los datos del chat, de los usuarios que participan en ese chat, y los mensajes que se han enviado a ese chat, para mandarlo como parámetros al renderizar la página Chat.ejs, la cual se va a encargar de acomodar todos los datos enviados por el servidor hacia el cliente, en caso de que todo esto se cumpla se envía un estatus correcto (200), en caso contrario, se envía un estatus incorrecto (400).

```
app.get('/Chat/:idChat', function (req, res) {
  var idChat = req.params.idChat;
  var usuarioActual = req.session.id_Usuario;
  sql = 'SELECT * FROM Chats c, Usuarios u, Participantes p WHERE
c.IdChat = p.IdChat AND u.IdUsuario = p.IdUsuario AND u.IdUsuario = ?
ORDER BY c.IdChat ASC;';
  db.all(sql, [usuarioActual], (err, comprobar) => {
    if(err){
      res.status(400).json({ "error": err.message });
      return;
    }else{
      comprobar.forEach((fila) => {
        if(fila.IdChat == idChat) {
          sql = 'SELECT * FROM Mensajes m, Usuarios u WHERE m.IdChat = ?
AND m.IdUsuario = u.IdUsuario ORDER BY m.IdMensaje ASC;';
          db.all(sql, [idChat], (err, rows) => {
```

```
            if (err) {
              res.status(400).json({ "error": err.message });
              return;
            } else {
              sql = 'SELECT * FROM Chats WHERE IdChat = ?;';
              db.all(sql, [idChat], (err, rows2) => {
                if (err) {
                  res.status(400).json({ "error": err.message });
                  return;
                } else {
                  sql = 'SELECT * FROM Chats c, Usuarios u, Participantes
p WHERE c.IdChat = p.IdChat AND u.IdUsuario = p.IdUsuario AND u.IdUsuario
= ? ORDER BY c.IdChat ASC;';
                  db.all(sql, [usuarioActual], (err, chatmenu) => {
                    if (err) {
                      res.status(400).json({ "error": err.message });
                      return;
                    } else {
                      res.status(200);
                      res.render('Chat.ejs', { mensajes: rows, actual:
usuarioActual, chatActual: rows2, chatMenu: chatmenu, idChatActual:
idChat, administrador: comprobar });
                    }
                  });
                }
              });
            }
          });
        }
      });
    }
  });
});
```

Después, tenemos la función de la petición “/chat/enviarMensaje/:idChat/:idUsuario”, la cual está construida con el método “.post()”, esta función se encarga de insertar en la tabla mensajes, el texto escrito en el formulario de la página Chat.ejs, en el chat y con el usuario especifico que envió el mensaje y hacía que chat lo envió, para posteriormente volver a renderizarlo con el estatus correcto (200), en caso contrario, regresa un error y el estatus incorrecto (400).

```
app.post('/chat/enviarMensaje/:idChat/:idUsuario', function (req, res) {
  var mensaje = req.body.mensajeEscrito
  var idChat = req.params.idChat;
  var idUsuario = req.params.idUsuario;
  db.run('INSERT INTO Mensajes (Texto, IdChat, IdUsuario) VALUES(?, ?,
?);', [mensaje, idChat, idUsuario], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect(req.get('referer'));
    }
  });
});
```

Después, tenemos la función de la petición “/NuevoGrupo”, la cual está construida con el método “.post()”, esta función se encarga de insertar en la tabla chats, un nuevo grupo creado por el usuario, después, selecciona y regresa todos los usuarios en una página llamada participantesGrupo.ejs, para que esta página muestre los usuarios y el usuario creador del grupo pueda escoger que usuario desea agregar al nuevo chat, si todo es correcto, se regresa un estatus correcto (200), en caso contrario, se regresa un error y un estatus incorrecto (400).

```
app.post('/NuevoGrupo', function (req, res) {
  var nombreGrupo = req.body.nombreGrupo;
  var idUsuarioActual = req.session.id_Usuario;
  db.run('INSERT INTO Chats (NombreChat) VALUES (?);', [nombreGrupo],
(err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      setTimeout(function () {
        db.get('SELECT * FROM Chats WHERE NombreChat = ? ORDER BY IdChat
ASC;', [nombreGrupo], (err, idChat) => {
          if (err) {
            res.status(400).json({ "error": err.message });
            return;
          } else {
            db.all('SELECT * FROM Chats WHERE NombreChat = ? ORDER BY
IdChat ASC;', [nombreGrupo], (err, chats) => {
              if (err) {
                res.status(400).json({ "error": err.message });
                return;
              } else {
                db.all('SELECT * FROM Usuarios WHERE IdUsuario != ?;',
[idUsuarioActual], (err, infoUsuarios) => {
                  if (err) {
                    res.status(400).json({ "error": err.message });
                    return;
                  } else {
                    db.run('INSERT INTO Participantes (IdChat, IdUsuario)
VALUES (?, ?);', [idChat.IdChat, idUsuarioActual], (err, result2) => {
                      if (err) {
                        res.status(400).json({ "error": err.message });
                        return;
                      } else {
                        res.render('participantesGrupo.ejs', { users:
infoUsuarios, idChatAgregar: chats, nombreNuevoGrupo: nombreGrupo });
                      }
                    });
                  }
                });
              }
            });
          }
        });
      }, 2000);
    }
  });
});
```

Después, tenemos la función de la petición “/agregarUsuario/:idChat/:idUsuario”, la cual está construida con el método “.post()”, esta función se encarga de insertar en la tabla participantes el usuario seleccionado por el usuario creador del chat, en el mismo chat, para que este pueda participar, además, selecciona todos los usuarios que no están en ese chat y se los regresa y renderiza la página participantesGrupo.ejs, pero esta vez con los participantes y usuarios que no están en el recién creado chat, si todo esto es correcto se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400), en ambas funciones descritas con anterioridad, se le coloca una función “setTimeout” para que a la base de datos, tenga tiempo a procesar los comandos DML de SQLite3, y los siguientes comandos no fallen.

```
app.post('/agregarUsuario/:idChat/:idUsuario', (req, res) => {
  var idChat = req.params.idChat;
  var idUsuario = req.params.idUsuario;
  var idUsuarioActual = req.session.id_Usuario;
  sql = 'INSERT INTO Participantes (IdChat, IdUsuario) VALUES (?,?);';
  db.run(sql, [idChat, idUsuario], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      setTimeout(function () {
        sql = 'SELECT DISTINCT(u.IdUsuario), u.Nombre, u.Correo FROM
Usuarios u, Participantes p WHERE u.IdUsuario != ? AND u.IdUsuario =
p.IdUsuario AND u.IdUsuario NOT IN (SELECT IdUsuario FROM Participantes
WHERE IdChat = ?);';
        db.all(sql, [idUsuarioActual, idChat], (err, infoUsuarios) => {
          if (err) {
            res.status(400).json({ "error": err.message });
            return;
          } else {
            sql = 'SELECT * FROM Chats WHERE IdChat = ?;';
            db.all(sql, [idChat], (err, nombreGrupo) => {
              if (err) {
                res.status(400).json({ "error": err.message });
                return;
              } else {
                res.status(200);
                res.render('participantesGrupo.ejs', { users:
infoUsuarios, idChatAgregar: nombreGrupo });
              }
            });
          }
        });
      }, 2000);
    }
  });
});
```

Después, tenemos la función de la petición “/Chat/EliminarMensaje/:IdMensaje”, la cual está construida con el método “.get()”, esta función se encarga de eliminar el mensaje seleccionado por el usuario, dentro de la función se borra de la tabla mensajes el id del mensaje enviado por el usuario, en caso de que sea correcto, se redirige a la misma página Chats.ejs, se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400).

```
app.get('/Chat/EliminarMensaje/:IdMensaje', function (req, res) {
  var IdMensaje = req.params.IdMensaje;
  sql = 'DELETE FROM Mensajes WHERE IdMensaje = ?;';
  db.run(sql, [IdMensaje], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect(req.get('referer'));
    }
  });
});
```

Después, tenemos la función de la petición “/Chat/ModificarMensaje/:IdMensaje”, la cual está construida con el método “.get()”, esta función se encarga de recibir un nuevo texto por parte del usuario que desea cambiar el texto de un mensaje, posteriormente la función actualiza la tabla mensajes con el nuevo mensaje, y redirige a la página Chat.ejs, y se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400).

```
app.get('/Chat/ModificarMensaje/:IdMensaje', function (req, res) {
  var IdMensaje = req.params.IdMensaje;
  var Texto = req.query.TextoMensajeModificar
  console.log('El nuevo mensaje es : ' + req.query.TextoMensajeModificar)
  sql = "UPDATE Mensajes  SET Texto = ? WHERE IdMensaje= ?";
  db.run(sql, [Texto, IdMensaje], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect(req.get('referer'));
    }
  });
});
```

Después, tenemos la función de la petición “/Chat/EliminarChat/:IdChat”, la cual está construida con el método “.get()”, esta función se encarga de eliminar un chat deseado por un usuario administrador, esta función borra de la tabla chats el chat seleccionado, en caso de que sea correcto se redirige a la página inicial donde estan todos los chats, y se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400).

```
app.get('/Chat/EliminarChat/:IdChat', function (req, res) {
  var IdChat = req.params.IdChat;
  sql = 'DELETE FROM Chats WHERE IdChat = ?;';
  db.run(sql, [IdChat], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect('/ExpressChat');
    }
  });
});
```

Después, tenemos la función de la petición “/Chat/ModificarChat/:IdChat”, la cual está construida con el método “.get()”, esta función se encarga de modificar el nombre del chat seleccionado por un usuario administrador, esta función modifica la tabla chats y cambia el nombre del chat, si todo es correcto se regresa a la misma página Chat.ejs, y se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400).

```
app.get('/Chat/ModificarChat/:IdChat', function (req, res) {
  var IdChat = req.params.IdChat;
  var Texto = req.query.TextoChatModificar
  console.log('El nuevo chat es : ' + req.query.TextoChatModificar)
  sql = "UPDATE Chats SET NombreChat = ? WHERE IdChat= ?";
  db.run(sql, [Texto, IdChat], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect(req.get('referer'));
    }
  });
});
```

Después, tenemos la función de la petición “/EliminarUsuario/:IdUsuario”, la cual está construida con el método “.get()”, esta función se encarga de eliminar un usuario seleccionado por el usuario administrador, esta función elimina el id especifico de un usuario de la tabla usuarios, si esto es correcto, se redirige a la misma página resultadoBusqueda.ejs, y se regresa un estatus correcto (200), en caso contrario, se regresa un estatus incorrecto (400).

```
app.get('/EliminarUsuario/:IdUsuario', function (req, res) {
  var IdUsuario = req.params.IdUsuario;
  sql = 'DELETE FROM Usuarios WHERE IdUsuario = ?;';
  db.run(sql, [IdUsuario], (err, result) => {
    if (err) {
      res.status(400).json({ "error": err.message });
      return;
    } else {
      res.status(200);
      res.redirect(req.get('referer'));
    }
  });
});
```

Por último, tenemos la función de la petición “*”, la cual está construida con el método “.get()”, esta función se encarga de renderizar la página Error404.ejs con un estatus incorrecto (404), a toda url que no coincida con las peticiones y funciones anteriormente descritas.

```
app.get('*', function (req, res) {
  res.status(404);
  res.render('Error404.ejs');
});
```

Páginas dinámicas con formato EJS:

Dentro del subdirectorio “views”, se crearon seis principales páginas, “Chat.ejs”, “Error404”, “index.ejs”, “participantesGrupo.ejs”, “Registro.ejs” y “resultadosBusqueda.ejs”, dentro de estas páginas se encuentran estructuras básicas HTML, con estilizado de un archivo CSS, pero con la diferencia de que dentro de estos archivos, se encuentra código de JavaScript que actúa cada que se carga la página, gracias a este código embebido, es que se puede estructurar con condicionales y ciclos, elementos dentro del HTML dependiendo del usuario que este ingresando y los datos que se estén recibiendo por parte del servidor, gracias a esto es que podemos desplegar elementos que tienen diferentes datos, dependiendo de la información que reciba como parámetros el archivo .ejs, con esto se generan gracias a los if, diferentes estructuras dentro de los archivos, y los for permiten desglosar la información que se recibe por parte del servidor en formato JSON, ya que, el servidor al renderizar un archivo .ejs, con parámetros, los parámetros se envían como un JSON, entonces gracias a los ciclos es que podemos desglosar y extraer la información (value) de cada dato (key), y después mostrarla en las etiquetas HTML, por último, recalcar que este código embebido dentro del EJS no se escribe tal cual como el JavaScript, si no que se debe de encerrar en etiquetas de EJS, donde, las etiquetas <% %>, sirven para colocar elementos del lenguaje de programación, como pueden ser ciclos, condicionales, métodos, funciones, etc, y las etiquetas <%= =%> sirven para colocar datos o variables regresadas por el servidor.

Ahora, con respecto a agregar, nuevos administradores, esto no es posible desde la API como tal, esto es debido a que se desea que el cliente no pueda hacer nada de esos temas administrativos, si no que, para agregar un administrador nuevo, se debe de hacer desde la parte del servidor en la consola de comandos de SQLite3, con el comando “UPDATE Usuarios SET Administrador = ‘A’ WHERE IdUsuario = ?”, de esta forma es que se pueden agregar nuevos privilegios de administrador a ciertos usuarios, ya que la aplicación no planea que existan muchos.

```
C:\Users\alegv\Desktop\ExpressChat\public\database>sqlite3 expressDB.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite>
sqlite> UPDATE Usuarios SET Administrador = ‘A’ WHERE IdUsuario = 1;
```