Documentación del Front-End

Introducción

El front-end de la aplicación ha sido desarrollado con Angular, y la documentación del código se ha generado utilizando **Compodoc**. Compodoc es una herramienta popular para la documentación de aplicaciones Angular que proporciona una interfaz visual clara y organizada. Esta documentación te ayudará a comprender cómo navegar por el proyecto, así como a generar y visualizar la documentación de forma efectiva.

Compodoc: Generación de la Documentación

Compodoc ya ha sido instalado en el proyecto utilizando `npm install`, por lo que ahora solo es necesario seguir algunos pasos simples para ejecutarlo y visualizar la documentación.

Pasos para ejecutar Compodoc:

1. **Generar la documentación**: Ejecuta el siguiente comando en la raíz de tu proyecto para que Compodoc genere la documentación:

   ```
   bash
   npx compodoc -p tsconfig.doc.json -s
   ```

   - `-p tsconfig.doc.json`: Especifica el archivo de configuración de TypeScript de la aplicación.
   - `-s`: Inicia un servidor HTTP para visualizar la documentación.
2. **Visualizar la documentación**: Una vez generado, Compodoc levantará un servidor local (por defecto en `http://localhost:8080`). Abre tu navegador y accede a esa dirección para explorar la documentación de tu proyecto.

Opciones adicionales de Compodoc:

- **Generar solo la documentación sin iniciar el servidor**:

  ```
  npx compodoc -p tsconfig.doc.json
  ```

- **Especificar un puerto diferente**:

  ```
  npx compodoc -p tsconfig.doc.json -s -p 8081
  ```

Estructura de la Documentación Generada

La documentación de Compodoc proporciona información detallada sobre el proyecto, incluyendo:

- **Módulos**: Una descripción de todos los módulos de Angular en el proyecto.
- **Componentes**: Información sobre cada componente, sus propiedades, métodos y diagramas de dependencia.
- **Servicios**: Detalles de los servicios, incluyendo su propósito y métodos expuestos.

- **Interfaces y Tipos**: Una lista de las interfaces y tipos utilizados en la aplicación, con explicaciones de sus propiedades.
- **Diagrama de cobertura**: Un análisis visual que muestra el nivel de cobertura de documentación del proyecto.

Ventajas de usar Compodoc:

- **Interfaz amigable**: Permite explorar el proyecto de una manera intuitiva.
- **Diagramas automáticos**: Genera gráficos de módulos y dependencias que ayudan a comprender la arquitectura del proyecto.
- **Actualización sencilla**: Con cada cambio en el código, se puede regenerar la documentación de forma rápida con el mismo comando.

Esta documentación generada con Compodoc será útil tanto para los desarrolladores que trabajan en el proyecto como para cualquier persona interesada en conocer su estructura y funcionamiento.

Documentación del Back-End

Introducción

Esta documentación está diseñada para proporcionar una visión detallada del backend de la aplicación, explicando la estructura de carpetas, los archivos principales, y las funciones generales de cada parte. El backend se ha desarrollado utilizando Node.js y Express, con una base de datos gestionada en MySQL y la configuración manejada a través de archivos de entorno. El enfoque modular y organizado de la estructura permite un mantenimiento sencillo y una fácil escalabilidad.

El propósito de este backend es ofrecer soporte a una aplicación que gestiona activos, usuarios, órdenes de trabajo y más, mediante una serie de APIs bien estructuradas. A continuación, se muestra la estructura de carpetas y archivos del proyecto, junto con una breve descripción de sus componentes.

Estructura de Carpetas

```
plaintext
Copiar código
back-end (carpeta madre)
- config
  -- db.js                          # Configuración de la conexión a la
base de datos

- src
  -- controllers                    # Controladores que manejan la
lógica de negocio
    --- activo.controller.js
    --- activostareas.controller.js
    --- autenticacion.controller.js
    --- datos.controller.js
    --- edificios.controller.js
    --- edificiospisos.controller.js
    --- login.controller.js
    --- ordenTrabajo.controller.js
    --- pisos.controller.js
    --- sector.controller.js
    --- tareas.controller.js
    --- tipoOrden.controller.js
    --- ubicacion.controller.js
    --- usuario.controller.js

  -- routes                         # Rutas que definen los endpoints
de la API
    --- activo.routes.js
    --- activostareas.routes.js
    --- autenticacion.routes.js
    --- datos.routes.js
    --- edificios.routes.js
    --- edificiospisos.routes.js
    --- login.routes.js
    --- ordenTrabajo.routes.js
    --- pisos.routes.js
    --- sector.routes.js
    --- tareas.routes.js
```

```
    --- tipoOrden.routes.js
    --- ubicacion.routes.js
    --- usuario.routes.js

  - index.js                        # Archivo principal que inicializa
el servidor y registra las rutas
```

Descripción de los Componentes

1. **`config/db.js`**: Archivo de configuración que maneja la conexión a la base de datos MySQL.
2. **`controllers/`**: Contiene la lógica de negocio de la aplicación. Cada archivo representa un controlador para gestionar un aspecto específico del sistema, como activos, usuarios, o edificios.
3. **`routes/`**: Define las rutas del servidor Express, vinculando las solicitudes HTTP con los métodos en los controladores correspondientes.
4. **`index.js`**: Punto de entrada del servidor. Configura las rutas, maneja errores y establece la configuración general del servidor y del middleware.

Esta estructura modular permite una separación clara entre la lógica de negocio y la definición de rutas, facilitando la organización del código y su mantenibilidad.

Documentación del archivo `index.js`

Descripción del archivo

El archivo `index.js` es el punto de entrada principal de la aplicación del servidor Express. Se configura el entorno, se instancian las rutas y se inicializa el servidor.

Contenido

1. **Configuración de variables de entorno:**
   - Se importa `dotenv` y se llama a `dotenv.config()` para cargar las variables de entorno desde un archivo `.env`.

```
javascript
import dotenv from 'dotenv';
dotenv.config();
```

2. **Importación de módulos:**
   - Se importan `express` y `cors` para la creación del servidor y el manejo de la seguridad.
   - Se importa `pool` desde el archivo de configuración de la base de datos.

- Se importan todas las rutas necesarias para la aplicación desde sus respectivos archivos.

```
javascript
import express from 'express';
import cors from 'cors';
import { pool } from '../config/db.js';

import pisosRoutes from './routes/pisos.routes.js';
import sectoresRoutes from './routes/sectores.routes.js';
import ubicacionRoutes from './routes/ubicacion.routes.js';
import userRoutes from './routes/usuario.routes.js';
import activoTareas from './routes/activostareas.routes.js';
import tareasRoutes from './routes/tareas.routes.js';
import loginRoutes from './routes/login.routes.js';
import edificioRoutes from './routes/edificios.routes.js';
import ordenTrabajoRoutes from
'./routes/ordenTrabajo.routes.js';
import edificioPisosRoutes from
'./routes/edificiopisos.routes.js';
import activoRoutes from './routes/activo.routes.js';
import tipoOrdenRoutes from './routes/tipoOrden.routes.js';
```

3. **Configuración de la aplicación:**
   - Se instancia la aplicación Express.
   - Se define el puerto desde una variable de entorno o un valor por defecto (3000).
   - Se configura `cors` para permitir solicitudes desde `http://localhost:4200` con credenciales.
   - Se agrega middleware para manejar datos JSON en las solicitudes.

```
javascript
const app = express();
const PORT = process.env.PORT || 3000;

app.use(cors({
    origin: 'http://localhost:4200',
    credentials: true
}));
app.use(express.json());
```

4. **Uso de rutas:**
   - Se montan todas las rutas importadas en el prefijo `/api`.

```
javascript
app.use('/api', activoRoutes);
app.use('/api', pisosRoutes);
app.use('/api', sectoresRoutes);
app.use('/api', ubicacionRoutes);
app.use('/api', userRoutes);
app.use('/api', activoTareas);
app.use('/api', tareasRoutes);
app.use('/api', loginRoutes);
app.use('/api', edificioRoutes);
app.use('/api', ordenTrabajoRoutes);
app.use('/api', edificioPisosRoutes);
app.use('/api', tipoOrdenRoutes);
```

5. **Manejo de errores:**
    - Se define un middleware de manejo de errores para capturar errores internos del servidor y responder con un mensaje de error en formato JSON.

```
javascript
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ message: 'Internal Server Error' });
});
```

6. **Inicialización del servidor:**
    - Se inicia el servidor en el puerto definido y se imprime un mensaje en la consola cuando el servidor está en ejecución.

```
javascript
app.listen(PORT, () => {
    console.log(`Server is running on
http://localhost:${PORT}`);
});
```

Resumen de las rutas y funcionalidades

Ruta Base	Descripción
`/api`	Prefijo común para todas las rutas.
`/api/pisos`	Gestión de pisos.
`/api/sector`	Gestión de sectores.
`/api/ubicaciones`	Gestión de ubicaciones.
`/api/usuarios`	Operaciones relacionadas con usuarios.
`/api/activo-tareas`	Gestión de tareas de activos.
`/api/tareas`	Operaciones sobre tareas.
`/api/login`	Autenticación de usuarios.
`/api/edificio`	Gestión de edificios.
`/api/orden_trabajo`	Gestión de órdenes de trabajo.
`/api/tiposOrden`	Gestión de tipos de orden.

Este archivo centraliza la configuración del servidor, la importación de rutas, y la inicialización del entorno de ejecución de la aplicación, proporcionando una estructura modular y organizada para el desarrollo y mantenimiento.

Documentación del archivo `db.js`

Descripción del archivo

El archivo `db.js` se encarga de configurar y establecer la conexión con la base de datos MySQL utilizando la librería `mysql2/promise`. Además, gestiona la carga de variables de entorno desde un archivo `.env` usando `dotenv`, lo que facilita el manejo de credenciales de forma segura.

Contenido

1. **Cargar variables de entorno:**
   - Utiliza la librería `dotenv` para cargar las variables de entorno desde el archivo `.env`:

     ```
     javascript
     Copiar código
     dotenv.config();
     ```

   - Esto asegura que las credenciales y parámetros de conexión (como el `host`, `user`, `password`, y `database`) estén disponibles desde las variables de entorno configuradas.
2. **Crear una piscina de conexiones a MySQL:**
   - La conexión a la base de datos se maneja a través de un `pool` creado con la librería `mysql2/promise`, lo que permite ejecutar consultas de manera eficiente utilizando promesas.

     ```
     javascript
     export const pool = createPool({
         host: process.env.DB_HOST,
         user: process.env.DB_USER,
         password: process.env.DB_PASSWORD,
         database: process.env.DB_DATABASE
     });
     ```

3. **Conexión a la base de datos:**
   - Después de crear el pool de conexiones, se establece una conexión inicial a la base de datos para verificar si la conexión es exitosa.

     ```
     javascript
     pool.getConnection()
         .then(() => console.log('Connected to the database!'))
         .catch(err => console.error('Database connection
     failed:', err));
     ```

   - Si la conexión es exitosa, se imprime el mensaje "Connected to the database!". Si ocurre un error, se muestra el mensaje "Database connection failed" junto con el error.

Uso

Este archivo exporta el objeto `pool`, que se puede importar en otros archivos de la aplicación para realizar consultas a la base de datos. Por ejemplo, en los controladores, puedes importar y usar el `pool` para ejecutar consultas SQL:

```
javascript
import { pool } from '../../config/db.js';
```

Configuración de Variables de Entorno

Las variables de entorno necesarias para la configuración de la base de datos deben definirse en un archivo `.env` en la raíz del proyecto. Ejemplo:

```
makefile
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=yourpassword
DB_DATABASE=mydatabase
```

Controladores

Documentación del Controlador `activo.controller.js`

Importaciones y Configuración

- **`import { pool } from '../../config/db.js';`**: Importa la conexión a la base de datos desde el archivo de configuración.

Funciones CRUD

1. **`getActivos`**
   - **Descripción**: Obtiene todos los registros de la tabla `activo`.
   - **Método HTTP**: `GET`
   - **Ruta**: `/api/activos`
   - **Respuesta**:
     - **Código 200**: Devuelve un array de objetos que representan los activos.
     - **Código 500**: Error interno del servidor.
2. **`getActivo`**
   - **Descripción**: Obtiene un activo específico por su `id`.
   - **Método HTTP**: `GET`
   - **Ruta**: `/api/activos/:id`
   - **Parámetros**:

        - `id`: ID del activo a obtener (en los parámetros de la ruta).
    - **Respuesta**:
        - **Código 200**: Devuelve el objeto del activo encontrado.
        - **Código 404**: Activo no encontrado.
        - **Código 500**: Error interno del servidor.
3. **`createActivo`**
    - **Descripción**: Crea un nuevo activo en la base de datos.
    - **Método HTTP**: `POST`
    - **Ruta**: `/api/activos`
    - **Cuerpo de la solicitud**:

```
json
{
  "nombre": "string",
  "tag_diminutivo": "string",
  "disponibilidad": "boolean" // opcional, por defecto 1
}
```

    - **Respuesta**:
        - **Código 200**: Devuelve un objeto con los datos del activo recién creado.
        - **Código 500**: Error interno del servidor.
4. **`updateActivo`**
    - **Descripción**: Actualiza un activo existente en la base de datos.
    - **Método HTTP**: `PUT`
    - **Ruta**: `/api/activos/:id`
    - **Parámetros**:
        - `id`: ID del activo a actualizar.
    - **Cuerpo de la solicitud**:

```
json
{
  "nombre": "string",
  "tag_diminutivo": "string",
  "disponibilidad": "boolean"
}
```

    - **Respuesta**:
        - **Código 200**: Devuelve un objeto con los datos del activo actualizado.
        - **Código 500**: Error interno del servidor.
5. **`deleteActivo`**
    - **Descripción**: Elimina un activo de la base de datos por su `id`.
    - **Método HTTP**: `DELETE`
    - **Ruta**: `/api/activos/:id`
    - **Parámetros**:
        - `id`: ID del activo a eliminar.
    - **Respuesta**:
        - **Código 200**: Mensaje de confirmación de eliminación.
        - **Código 500**: Error interno del servidor.

Notas

- **Manejo de Errores**: En cada función, se incluye un `try-catch` para capturar y manejar errores, devolviendo una respuesta con código 500 y un mensaje de error en caso de fallo.
- **Conexión a la base de datos**: Se utiliza la conexión `pool` para ejecutar las consultas SQL de forma asíncrona.

Documentación del Controlador `activostareas.controller.js`

Importaciones y Configuración

- **`import { pool } from '../../config/db.js';`**: Importa la conexión a la base de datos desde el archivo de configuración, permitiendo la ejecución de consultas SQL.

Funciones CRUD

1. **`getactivotareas`**
    - **Descripción**: Obtiene una lista de las tareas asociadas a cada activo, mostrando el nombre del activo, el ID del activo y la descripción de la tarea.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/activotareas`
    - **Respuesta**:
        - **Código 200**: Devuelve un array de objetos con las siguientes propiedades:
            - `nombre_activo`: Nombre del activo.
            - `id_activo`: ID del activo.
            - `descripcion`: Descripción de la tarea asociada.
        - **Código 500**: Error interno del servidor, acompañado de un mensaje descriptivo en formato JSON.

Consulta SQL Utilizada

- **Consulta**:

```
sql
SELECT a.nombre AS nombre_activo, at.id_activo, t.descripcion
FROM activo_tarea at
JOIN tareas t ON at.id_tarea = t.id_tarea
JOIN activo a ON at.id_activo = a.id_activo
```

    - **Explicación**:
        - Se realiza una consulta que une las tablas `activo_tarea`, `tareas`, y `activo` para obtener los detalles de las tareas asignadas a cada activo.
        - `JOIN` se utiliza para combinar las filas de las tablas basándose en las relaciones entre `id_tarea` y `id_activo`.

Notas

- **Manejo de Errores**: La función incluye un bloque `try-catch` para capturar y manejar posibles errores en la ejecución de la consulta. Si ocurre un error, se registra en la consola y se devuelve una respuesta con código 500 y un mensaje de error en formato JSON.
- **Estructura de Respuesta**: La función devuelve los datos en formato JSON con la estructura mencionada anteriormente, útil para la visualización de las tareas de activos en la aplicación.

Documentación del Controlador `autenticacion.controller.js`

Descripción General

Este archivo proporciona middleware para la autenticación de solicitudes mediante tokens JWT (JSON Web Tokens). La autenticación es un proceso esencial para proteger las rutas y garantizar que solo los usuarios autorizados puedan acceder a ciertas funcionalidades.

Importaciones y Configuración

- **`const jwt = require('jsonwebtoken');`**: Importa la biblioteca `jsonwebtoken` para manejar la creación y verificación de tokens JWT.
- **`const SECRET_KEY = process.env.SECRET_KEY;`**: Define la clave secreta utilizada para la verificación de tokens, obtenida de las variables de entorno del proyecto.

Funciones y Middleware

1. **`autenticarToken`**
    - **Descripción**: Middleware que verifica la validez de un token JWT en las solicitudes.
    - **Parámetros de la función**:
        - `req`: Objeto de solicitud.
        - `res`: Objeto de respuesta.
        - `next`: Función para pasar el control al siguiente middleware.
    - **Funcionamiento**:
        - Verifica si hay un encabezado de autorización (`Authorization`) y extrae el token de este.
        - Si el token no está presente, devuelve una respuesta con código `401` y un mensaje de error (`Token no proporcionado`).
        - Utiliza `jwt.verify()` para comprobar la validez del token:
            - Si el token no es válido, devuelve un código `403` con el mensaje `Token no válido`.

        - Si el token es válido, se adjunta el usuario decodificado al objeto `req` y se llama a `next()` para continuar con la solicitud.
    - **Respuesta**:
        - **Código 401**: Token no proporcionado.
        - **Código 403**: Token no válido.
        - **Código 200**: Si el token es válido, permite continuar con la ejecución de la ruta.

Ejemplo de Uso

Este middleware puede usarse en rutas protegidas de la siguiente manera:

```
javascript
const autenticarToken =
require('./controllers/autenticacion.controller');

app.get('/ruta-protegida', autenticarToken, (req, res) => {
    res.json({ message: 'Acceso autorizado', user: req.user });
});
```

Notas

- **Seguridad**: La clave `SECRET_KEY` es fundamental para la seguridad del token y debe mantenerse protegida. Asegúrate de definirla correctamente en tu archivo de variables de entorno (por ejemplo, `.env`).
- **Errores Manejados**:
    - Si no se proporciona un token, se devuelve un error `401`.
    - Si el token es inválido, se devuelve un error `403`.

Documentación del Controlador `datos.controller.js`

Descripción General

Este controlador gestiona las solicitudes para obtener datos de las diferentes entidades de la base de datos, como operarios, edificios, pisos y sectores. Estas funciones son útiles para rellenar selectores o listas en el frontend de la aplicación.

Funciones CRUD

1. **`getOperarios`**
    - **Descripción**: Recupera una lista de operarios.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/operarios`
    - **Respuesta**:
        - **Código 200**: Devuelve un array de objetos con los siguientes campos:

        - `id_usuario`: ID del operario.
        - `nombre`: Nombre del operario.
    - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Consulta SQL:

```
sql
Copiar código
SELECT id_usuario, nombre FROM usuarios;
```

2. **`getEdificios`**
    - **Descripción**: Obtiene una lista de edificios.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/edificios`
    - **Respuesta**:
        - **Código 200**: Devuelve un array de objetos con los siguientes campos:
            - `id_edificio`: ID del edificio.
            - `nombre`: Nombre del edificio.
        - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Consulta SQL:

```
sql
SELECT id_edificio, nombre FROM edificio;
```

3. **`getPisos`**
    - **Descripción**: Recupera una lista de pisos.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/pisos`
    - **Respuesta**:
        - **Código 200**: Devuelve un array de objetos con los siguientes campos:
            - `id_piso`: ID del piso.
            - `nombre`: Nombre del piso.
        - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Consulta SQL:

```
sql
SELECT id_piso, nombre FROM piso_nivel;
```

4. **`getSectores`**
    - **Descripción**: Obtiene una lista de sectores.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/sectores`
    - **Respuesta**:

        - **Código 200**: Devuelve un array de objetos con los siguientes campos:
            - `id_sector`: ID del sector.
            - `nombre`: Nombre del sector.
        - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

    Consulta SQL:

```
sql
SELECT id_sector, nombre FROM sector;
```

Notas

- **Manejo de Errores**: Todas las funciones incluyen un bloque `try-catch` que captura errores y devuelve un mensaje de error genérico con código `500` en caso de que ocurra algún problema en la ejecución de la consulta.
- **Respuesta de la Base de Datos**: Se devuelve la respuesta tal cual es obtenida de `pool.query`, lo que significa que los datos estarán en formato de array de objetos.

Documentación del Controlador `edificios.controller.js`

Descripción General

Este controlador está encargado de manejar las operaciones relacionadas con la entidad `edificio` en la base de datos. Proporciona funcionalidades para obtener, crear, actualizar y eliminar registros de edificios.

Funciones CRUD

1. **`getEdificios`**
    - **Descripción**: Obtiene todos los edificios registrados en la base de datos.
    - **Método HTTP**: `GET`
    - **Ruta**: `/api/edificios`
    - **Respuesta**:
        - **Código 200**: Devuelve un array de objetos con la información de todos los edificios.
        - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

    Consulta SQL:

```
sql
Copiar código
SELECT * FROM edificio;
```

2. **`createEdificio`**
   - **Descripción**: Crea un nuevo registro de edificio en la base de datos.
   - **Método HTTP**: `POST`
   - **Ruta**: `/api/edificios`
   - **Parámetros del Cuerpo (Body)**:
     - `nombre`: Nombre del edificio.
     - `direccion`: Dirección del edificio.
     - `labeltag`: Identificador de etiqueta.
     - `activo`: Estado del edificio (booleano/int).
   - **Respuesta**:
     - **Código 201**: Devuelve un objeto con el `id` del edificio recién creado y los datos proporcionados.
     - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

   Consulta SQL:

```
sql
Copiar código
INSERT INTO edificio (nombre, direccion, labeltag, activo)
VALUES (?, ?, ?, ?);
```

3. **`deleteEdificio`**
   - **Descripción**: Elimina un edificio de la base de datos según el ID proporcionado.
   - **Método HTTP**: `DELETE`
   - **Ruta**: `/api/edificios/:id`
   - **Parámetro de Ruta**:
     - `id`: ID del edificio a eliminar.
   - **Respuesta**:
     - **Código 200**: Mensaje indicando que el edificio fue eliminado con éxito.
     - **Código 404**: Si el edificio no se encuentra en la base de datos.
     - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

   Consulta SQL:

```
sql
DELETE FROM edificio WHERE id_edificio = ?;
```

4. **`updateLabelTag`**
   - **Descripción**: Actualiza el `labeltag` de un edificio.
   - **Método HTTP**: `PATCH`
   - **Ruta**: `/api/edificios/:id/labeltag`
   - **Parámetros del Cuerpo (Body)**:
     - `labeltag`: Nuevo valor para el campo `labeltag`.

- **Parámetro de Ruta**:
  - `id`: ID del edificio.
- **Respuesta**:
  - **Código 200**: Mensaje indicando que el `labeltag` fue actualizado.
  - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Consulta SQL:

```
sql
UPDATE edificio SET labeltag = ? WHERE id_edificio = ?;
```

5. **`updateEdificio`**
   - **Descripción**: Actualiza la información de un edificio según el ID proporcionado.
   - **Método HTTP**: `PUT`
   - **Ruta**: `/api/edificios/:id_edificio`
   - **Parámetros del Cuerpo (Body)**:
     - `nombre`: Nuevo nombre del edificio.
     - `direccion`: Nueva dirección del edificio.
     - `labeltag`: Nuevo valor para el `labeltag`.
     - `activo`: Estado actualizado del edificio (booleano/int).
   - **Parámetro de Ruta**:
     - `id_edificio`: ID del edificio a actualizar.
   - **Respuesta**:
     - **Código 200**: Devuelve un objeto con los datos actualizados del edificio.
     - **Código 404**: Si el edificio no se encuentra en la base de datos.
     - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Consulta SQL:

```
sql
UPDATE edificio SET nombre = ?, direccion = ?, labeltag = ?,
activo = ? WHERE id_edificio = ?;
```

Notas

- **Manejo de Errores**: Cada función incluye un bloque `try-catch` que captura errores y devuelve un mensaje con código `500` si ocurre algún problema.
- **Mensajes de Log**: Se utiliza `console.log` para registrar los datos recibidos y los resultados de la actualización para facilitar la depuración.
- **Estructura de Respuesta**: Las respuestas exitosas devuelven información clave para confirmar la operación al frontend.

Documentación del Controlador `edificiospisos.controller.js`

Descripción General

Este controlador permite obtener un edificio y los pisos específicos asociados a él, en este caso, filtrando por el nombre del edificio y pisos predefinidos. Actualmente, está diseñado para trabajar principalmente con un edificio específico (`UTN FRVT`).

Función: `obtenerPisosPorEdificio`

- **Descripción**: Recupera un edificio de la base de datos junto con sus pisos asociados, siempre y cuando cumpla con criterios específicos (nombre del edificio y pisos permitidos).
- **Método HTTP**: `GET`
- **Ruta**: `/api/edificios/:id_edificio/pisos`
- **Parámetro de Ruta**:
  - `id_edificio`: ID del edificio a consultar.
- **Respuesta**:
  - **Código 200**: Devuelve un objeto con los datos del edificio y un array de objetos con la información de los pisos asociados.
  - **Código 400**: Si el ID del edificio no es `1`, indicando que el controlador no está diseñado para manejar otros edificios.
  - **Código 404**: Si no se encuentra un edificio con el `id` y nombre especificado.
  - **Código 500**: Error interno del servidor con un mensaje de error en JSON.

Detalles de Implementación

1. **Consulta SQL para Edificio**:

   ```
   sql
   SELECT *
   FROM edificio
   WHERE id_edificio = ? AND nombre = 'UTN FRVT';
   ```

   - Filtra el edificio por `id_edificio` y asegura que el nombre sea `UTN FRVT`.
2. **Consulta SQL para Pisos**:

   ```
   sql
   SELECT p.*
   FROM piso_nivel p
   WHERE p.nombre IN ('Planta Baja', '1er. Piso', '2do. Piso',
   'Nivel 0', 'Nivel Bajo 0');
   ```

   - Recupera los pisos con nombres específicos predefinidos.
3. **Validación de ID de Edificio**:

    - La función verifica si el `id_edificio` es `1`. Si no lo es, retorna un código `400` con el mensaje: "Este edificio no está soportado actualmente".
4. **Verificación de Resultados**:
    - Si la consulta del edificio no devuelve resultados (`edificio.length === 0`), se devuelve un código `404` indicando que el edificio no fue encontrado.
5. **Respuesta Exitosa**:
    - Si se encuentra el edificio y los pisos, se responde con un objeto JSON que incluye los detalles del edificio y los pisos obtenidos.

Ejemplo de Respuesta Exitosa

```
json
{
  "edificio": {
    "id_edificio": 1,
    "nombre": "UTN FRVT",
    "direccion": "Dirección ejemplo",
    "labeltag": "Etiqueta ejemplo",
    "activo": true
  },
  "pisos": [
    {
      "id_piso": 1,
      "nombre": "Planta Baja"
    },
    {
      "id_piso": 2,
      "nombre": "1er. Piso"
    }
  ]
}
```

Manejo de Errores

- Se utilizan bloques `try-catch` para capturar errores inesperados y responder con un código `500` y un mensaje de error.
- Se registra en `console.error` para facilitar la depuración.

Documentación del Controlador `login.controller.js`

Descripción General

Este controlador gestiona las solicitudes relacionadas con el registro de nuevos usuarios, inicio de sesión, verificación de tokens y cierre de sesión. Proporciona funcionalidad para autenticar usuarios mediante JWT y manejar el flujo de autenticación en la aplicación.

Funciones CRUD

1. **registro**
   - **Descripción**: Registra un nuevo usuario en la base de datos.
   - **Método HTTP**: `POST`
   - **Ruta**: `/api/registro`
   - **Respuesta**:
     - **Código 201**: Usuario registrado exitosamente.
     - **Código 400**: Si faltan parámetros necesarios (`nombre`, `email`, `password`), o si el usuario ya existe.
     - **Código 500**: Error interno del servidor con un mensaje de error en JSON.
   - **Cuerpo de la solicitud**:

     ```
     json
     {
       "nombre": "Nombre del usuario",
       "email": "usuario@dominio.com",
       "password": "contraseña_segura"
     }
     ```

   - **Consulta SQL**:

     ```
     sql
     SELECT * FROM usuario WHERE email = ?;
     INSERT INTO usuario (nombre, email, password) VALUES (?,
     ?, ?);
     ```

2. **login**
   - **Descripción**: Inicia sesión en la aplicación, generando un token JWT para el usuario autenticado.
   - **Método HTTP**: `POST`
   - **Ruta**: `/api/login`
   - **Respuesta**:
     - **Código 200**: Inicio de sesión exitoso, devuelve el token JWT, nombre del usuario y su rol.
     - **Código 401**: Si el usuario no existe o la contraseña es incorrecta.
     - **Código 500**: Error interno del servidor con un mensaje de error en JSON.
   - **Cuerpo de la solicitud**:

     ```
     json
     {
       "email": "usuario@dominio.com",
       "password": "contraseña_segura"
     }
     ```

   - **Consulta SQL**:

     ```
     sql
     SELECT * FROM usuario WHERE email = ?;
     ```

3. **verifyToken**

- **Descripción**: Middleware para verificar la validez del token JWT en la cabecera `Authorization`.
- **Método HTTP**: No aplica (es un middleware utilizado en rutas protegidas).
- **Ruta**: No tiene ruta directa, se utiliza como middleware.
- **Respuesta**:
  - **Código 403**: Si el token no está presente o es inválido.
- **Funcionamiento**: Verifica el token enviado en la cabecera `Authorization` y decodifica su contenido. Si es válido, permite el acceso a la ruta protegida.

4. **protect**
   - **Descripción**: Middleware para proteger las rutas que requieren autenticación mediante un token JWT almacenado en cookies o en la cabecera `Authorization`.
   - **Método HTTP**: No aplica (es un middleware utilizado en rutas protegidas).
   - **Ruta**: No tiene ruta directa, se utiliza como middleware.
   - **Respuesta**:
     - **Código 401**: Si no se proporciona un token o si el token es inválido.
   - **Funcionamiento**: Verifica el token en las cookies o cabecera, y si es válido, permite el acceso a las rutas protegidas.
5. **logout**
   - **Descripción**: Cierra la sesión del usuario y elimina el token JWT de las cookies.
   - **Método HTTP**: `POST`
   - **Ruta**: `/api/logout`
   - **Respuesta**:
     - **Código 200**: Sesión cerrada exitosamente.
   - **Funcionamiento**: Elimina el token JWT de las cookies y envía un mensaje de éxito al cliente.

Notas

- **Manejo de Errores**: Cada función maneja errores mediante bloques `try-catch`, devolviendo códigos de error HTTP apropiados, como 400, 401, 500, etc. Los errores específicos (como usuario no encontrado o contraseña incorrecta) se detallan en los mensajes de error.
- **Respuesta de la Base de Datos**: Las consultas SQL devuelven un array de objetos con los resultados de la consulta, que se utilizan para verificar y gestionar el flujo de autenticación.
- **Uso de JWT**: El token JWT se genera con un tiempo de expiración de una hora y se almacena en las cookies para su uso en solicitudes posteriores.

Documentación del Controlador `ordenTrabajo.controller.js`

---

1. `getOrdenTrabajo`

Descripción:
Obtiene una lista de todas las órdenes de trabajo con los detalles relacionados, incluyendo usuario, tag, activo, edificio, piso, sector y más.

Ruta:
`GET /api/orden_trabajo`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de órdenes de trabajo con los siguientes datos:

- `id_orden_trabajo`
- `fecha_impresion`
- `hora_impresion`
- `realizada`
- `id_usuario`
- `nombre_usuario`
- `id_tag`
- `nombre_tag`
- `id_activo`
- `nombre_activo`
- `id_edificio`
- `nombre_edificio`
- `id_piso`
- `nombre_piso`
- `id_sector`
- `nombre_sector`
- `codigo`
- `observacion`

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Internal Server Error" }`.

---

2. `getDetallesOrdenTrabajo`

Descripción:
Obtiene los detalles de una orden de trabajo específica.

Ruta:
`GET /api/orden_trabajo/:id`

Parámetros:

- **`:id` (param):** ID de la orden de trabajo a obtener.

Respuesta Exitosa (200):
Retorna los detalles completos de la orden de trabajo con los siguientes datos:

- `id_orden_trabajo`
- `fecha_impresion`
- `hora_impresion`
- `realizada`
- `id_usuario`
- `nombre_usuario`
- `id_tag`
- `nombre_tag`
- `id_activo`
- `nombre_activo`
- `id_edificio`
- `nombre_edificio`
- `id_piso`
- `nombre_piso`
- `id_sector`
- `nombre_sector`
- `codigo`
- `observacion`
- `id_tipo_orden`
- `descripcion_tipo_orden`

Respuesta de Error (404):

- Si la orden de trabajo no se encuentra, devuelve el mensaje:
  `{ message: "Orden de trabajo no encontrada" }`.

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Error interno del servidor" }`.

---

3. **`nuevaODT`**

Descripción:
Crea una nueva orden de trabajo con los datos proporcionados en el cuerpo de la solicitud.

Ruta:
`POST /api/orden_trabajo`

Parámetros:

- **Cuerpo de la solicitud (JSON):**
  - `fecha_impresion`: Fecha de impresión de la orden.
  - `hora_impresion`: Hora de impresión de la orden.
  - `realizada`: Indica si la orden fue realizada (booleano).
  - `id_usuario`: ID del usuario relacionado con la orden.
  - `id_sector`: ID del sector relacionado con la orden.
  - `id_piso`: ID del piso relacionado con la orden.

- `id_edificio`: ID del edificio relacionado con la orden.
- `id_tag`: ID del tag asociado con la orden.
- `id_activo`: ID del activo asociado con la orden.
- `codigo`: Código de la orden de trabajo.
- `id_tipo_orden`: ID del tipo de orden.
- `observacion`: Observaciones relacionadas con la orden.

Respuesta Exitosa (201):
Retorna un mensaje indicando que la orden fue creada exitosamente con el `id_orden_trabajo` de la nueva orden de trabajo.
Ejemplo de respuesta:

```
json
Copiar código
{
  "message": "Orden de trabajo creada.",
  "id_orden_trabajo": 123
}
```

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Error interno del servidor" }`.

---

4. `deleteOrdenTrabajo`

Descripción:
Elimina una orden de trabajo específica por su ID.

Ruta:
`DELETE /api/orden_trabajo/:id`

Parámetros:

- **`:id` (param):** ID de la orden de trabajo a eliminar.

Respuesta Exitosa (200):
Retorna un mensaje indicando que la orden de trabajo fue eliminada exitosamente.

Respuesta de Error (404):

- Si no se encuentra la orden de trabajo con el ID proporcionado, devuelve el mensaje:
  `{ message: "Orden de trabajo no encontrada" }`.

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Error interno del servidor" }`.

---

5. `getConcatenacionIds`

Descripción:
Obtiene una concatenación de los IDs de usuario, edificio, piso, sector y activo relacionados con una orden de trabajo específica.

Ruta:
`GET /api/orden_trabajo/:id/concatenacion`

Parámetros:

- **`:id` (param):** ID de la orden de trabajo para la cual obtener la concatenación de los IDs.

Respuesta Exitosa (200):
Retorna un objeto con la concatenación de los IDs en el siguiente formato:

```
json
{
  "concatenacion": "1-2-3-4-5"
}
```

Respuesta de Error (404):

- Si no se encuentra la orden de trabajo con el ID proporcionado, devuelve el mensaje: `{ message: "Orden de trabajo no encontrada" }`.

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje: `{ message: "Error interno del servidor" }`.

---

6. `getTiposOrden`

Descripción:
Obtiene todos los tipos de orden disponibles.

Ruta:
`GET /api/tipo_orden`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de todos los tipos de orden disponibles con los siguientes campos:

- `id_tipo_orden`
- `descripcion`

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Error interno del servidor" }`.

---

7. `actualizarRealizada`

Descripción:
Actualiza el campo `realizada` de una orden de trabajo específica.

Ruta:
`PUT /api/orden_trabajo/:id/realizada`

Parámetros:

- **`:id` (param):** ID de la orden de trabajo a actualizar.
- **Cuerpo de la solicitud (JSON):**
  - `realizada`: Valor booleano (true/false) que indica si la orden fue realizada.

Respuesta Exitosa (200):
Retorna un mensaje indicando que la orden de trabajo fue actualizada correctamente.

Respuesta de Error (400):

- Si el campo `realizada` no se incluye en la solicitud, devuelve el mensaje:
  `{ message: "El campo realizada es obligatorio" }`.

Respuesta de Error (404):

- Si no se encuentra la orden de trabajo con el ID proporcionado, devuelve el mensaje:
  `{ message: "Orden de trabajo no encontrada" }`.

Respuesta de Error (500):

- Si ocurre un error interno en el servidor, devuelve el mensaje:
  `{ message: "Error interno del servidor" }`.

Documentación del Controlador `pisos.controller.js`

---

1. `getPisos`

Descripción:
Obtiene una lista de todos los pisos (niveles) registrados en la base de datos.

Ruta:
`GET /api/pisos`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de pisos con los siguientes campos:

- `id_piso`: ID único del piso.
- `nombre`: Nombre del piso.
- `labeltag`: Etiqueta asociada al piso.
- `activo`: Estado de actividad del piso (1 si está activo, 0 si no lo está).

Ejemplo de respuesta:

```
json
 [
    {
        "id_piso": 1,
        "nombre": "Piso 1",
        "labeltag": "P1",
        "activo": 1
    },
    {
        "id_piso": 2,
        "nombre": "Piso 2",
        "labeltag": "P2",
        "activo": 1
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

2. `createPiso`

Descripción:
Crea un nuevo piso en la base de datos con los datos proporcionados en el cuerpo de la solicitud.

Ruta:
`POST /api/pisos`

Parámetros:

- **Cuerpo de la solicitud (JSON):**
  - `nombre`: Nombre del piso (requerido).
  - `labeltag`: Etiqueta asociada al piso (opcional).
  - `activo`: Estado de actividad del piso (opcional, por defecto 1 si no se proporciona).

Respuesta Exitosa (201):
Retorna un mensaje indicando que el piso fue creado exitosamente junto con el `id_piso` del nuevo piso creado.
Ejemplo de respuesta:

```
json
{
  "message": "Piso creado exitosamente",
  "id": 3
}
```

Respuesta de Error (400):
Si el campo `nombre` no se proporciona en la solicitud, devuelve el mensaje:

```
json
{ "message": "El nombre del piso es obligatorio" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

3. `deletePiso`

Descripción:
Elimina un piso específico de la base de datos según su ID.

Ruta:
`DELETE /api/pisos/:id`

Parámetros:

- **`:id` (param):** ID del piso a eliminar.

Respuesta Exitosa (200):
Retorna un mensaje indicando que el piso fue eliminado exitosamente.
Ejemplo de respuesta:

```
json
{ "message": "Piso eliminado exitosamente" }
```

Respuesta de Error (404):
Si no se encuentra el piso con el ID proporcionado, devuelve el mensaje:

```
json
{ "message": "Piso no encontrado" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

Documentación del Controlador `sector.controller.js`

---

1. `getSectores`

Descripción:
Obtiene una lista de todos los sectores registrados en la base de datos.

Ruta:
`GET /api/sectores`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de sectores con los siguientes campos:

- `id_sector`: ID único del sector.
- `nombre`: Nombre del sector.
- `descripcion`: Descripción del sector.
- `activo`: Estado de actividad del sector (1 si está activo, 0 si no lo está).

Ejemplo de respuesta:

```
json
 [
    {
        "id_sector": 1,
        "nombre": "Sector A",
        "descripcion": "Descripción del Sector A",
        "activo": 1
    },
    {
        "id_sector": 2,
        "nombre": "Sector B",
        "descripcion": "Descripción del Sector B",
        "activo": 1
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

2. `createSector`

Descripción:
Crea un nuevo sector en la base de datos con los datos proporcionados en el cuerpo de la solicitud.

Ruta:
`POST /api/sectores`

Parámetros:

- **Cuerpo de la solicitud (JSON):**
  - `nombre`: Nombre del sector (requerido).
  - `descripcion`: Descripción del sector (opcional).
  - `activo`: Estado de actividad del sector (opcional, por defecto 1 si no se proporciona).

Respuesta Exitosa (201):
Retorna un mensaje indicando que el sector fue creado exitosamente junto con el `id_sector` del nuevo sector creado.
Ejemplo de respuesta:

```
json
{
  "message": "Sector creado exitosamente",
  "id": 3
}
```

Respuesta de Error (400):
Si el campo `nombre` no se proporciona en la solicitud, devuelve el mensaje:

```
json
{ "message": "El nombre del sector es obligatorio" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

3. `deleteSector`

Descripción:
Elimina un sector específico de la base de datos según su ID.

Ruta:
`DELETE /api/sectores/:id`

Parámetros:

- **`:id` (param):** ID del sector a eliminar.

Respuesta Exitosa (200):
Retorna un mensaje indicando que el sector fue eliminado exitosamente.
Ejemplo de respuesta:

```
json
{ "message": "Sector eliminado exitosamente" }
```

Respuesta de Error (404):
Si no se encuentra el sector con el ID proporcionado, devuelve el mensaje:

```
json
{ "message": "Sector no encontrado" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el mensaje:

```
json
{ "message": "Error interno del servidor" }
```

Documentación del Controlador `tareas.controller.js`

---

1. `getTareas`

Descripción:
Obtiene una lista de todas las tareas registradas en la base de datos.

Ruta:
`GET /api/tareas`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de tareas con los siguientes campos:

- `id_tarea`: ID único de la tarea.
- `descripcion`: Descripción de la tarea.

Ejemplo de respuesta:

```
json
 [
    {
        "id_tarea": 1,
        "descripcion": "Tarea de mantenimiento"
    },
    {
        "id_tarea": 2,
        "descripcion": "Reparación de equipos"
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

2. `createTarea`

Descripción:
Crea una nueva tarea en la base de datos con la descripción proporcionada en el cuerpo de la solicitud.

Ruta:
`POST /api/tareas`

Parámetros:

- **Cuerpo de la solicitud (JSON):**
  - `descripcion`: Descripción de la tarea (requerida).

Respuesta Exitosa (201):
Retorna un mensaje indicando que la tarea fue creada exitosamente junto con el `id_tarea` de la nueva tarea creada.
Ejemplo de respuesta:

```
json
{
  "message": "Tarea creada exitosamente",
  "id": 3
}
```

Respuesta de Error (400):
Si el campo `descripcion` no se proporciona en la solicitud, devuelve el siguiente mensaje:

```
json
{ "message": "Descripción es requerida" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

3. `deleteTarea`

Descripción:
Elimina una tarea específica de la base de datos según su ID.

Ruta:
`DELETE /api/tareas/:id_tarea`

Parámetros:

- **`:id_tarea` (param):** ID de la tarea a eliminar.

Respuesta Exitosa (200):
Retorna un mensaje indicando que la tarea fue eliminada exitosamente.
Ejemplo de respuesta:

```
json
{ "message": "Tarea eliminada exitosamente" }
```

Respuesta de Error (404):
Si no se encuentra la tarea con el ID proporcionado, devuelve el siguiente mensaje:

```
json
{ "message": "Tarea no encontrada" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

Documentación del Controlador `tipoOrden.controller.js`

---

1. `getTiposOrden`

Descripción:
Obtiene todos los tipos de órdenes registrados en la base de datos.

Ruta:
`GET /api/tipoOrden`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de tipos de orden con los siguientes campos:

- `id_tipo_orden`: ID único del tipo de orden.
- `nombre`: Nombre del tipo de orden.

Ejemplo de respuesta:

```
json
 [
    {
        "id_tipo_orden": 1,
        "nombre": "Mantenimiento Preventivo"
    },
    {
        "id_tipo_orden": 2,
        "nombre": "Reparación"
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

Documentación del Controlador `ubicacion.controller.js`

---

1. `getUbicaciones`

Descripción:
Obtiene todas las ubicaciones registradas en la base de datos.

Ruta:
`GET /api/ubicaciones`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de ubicaciones con los siguientes campos:

- `id_ubicacion`: ID único de la ubicación.
- `nombre`: Nombre de la ubicación.
- `labeltag`: Etiqueta asociada a la ubicación.
- `activo`: Estado de la ubicación (activo o inactivo).

Ejemplo de respuesta:

```
json
 [
    {
        "id_ubicacion": 1,
        "nombre": "Edificio A - Piso 1",
        "labeltag": "A-1",
        "activo": 1
    },
    {
        "id_ubicacion": 2,
        "nombre": "Edificio B - Piso 2",
        "labeltag": "B-2",
        "activo": 1
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

2. `createUbicacion`

Descripción:
Crea una nueva ubicación en la base de datos.

Ruta:
`POST /api/ubicaciones`

Parámetros (body):

- `nombre` (string, requerido): Nombre de la nueva ubicación.
- `labeltag` (string, opcional): Etiqueta asociada a la ubicación.
- `activo` (boolean, opcional): Estado de la ubicación (por defecto es 1, que significa activo).

Respuesta Exitosa (201):
Retorna un mensaje de éxito y el `id` de la ubicación creada.

```
json
{
    "message": "Ubicación creada exitosamente",
    "id": 5
}
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

---

3. `deleteUbicacion`

Descripción:
Elimina una ubicación de la base de datos mediante su `id`.

Ruta:
`DELETE /api/ubicaciones/:id`

Parámetros (ruta):

- `id` (integer, requerido): ID de la ubicación a eliminar.

Respuesta Exitosa (200):
Retorna un mensaje indicando que la ubicación fue eliminada exitosamente.

```
json
{ "message": "Ubicación eliminada exitosamente" }
```

Respuesta de Error (404):
Si la ubicación no existe, se retorna:

```
json
{ "message": "Ubicación no encontrada" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Error interno del servidor" }
```

Documentación del Controlador `usuario.controller.js`

---

1. `getUsers`

Descripción:
Obtiene todos los usuarios registrados en la base de datos.

Ruta:
`GET /api/usuarios`

Parámetros:

- **Ninguno.**

Respuesta Exitosa (200):
Retorna una lista de usuarios con los siguientes campos:

- `id_usuario`: ID único del usuario.
- `nombre`: Nombre del usuario.
- `email`: Correo electrónico del usuario.
- `password`: Contraseña del usuario (debe estar protegida).

Ejemplo de respuesta:

```
json
 [
    {
        "id_usuario": 1,
        "nombre": "Juan Pérez",
        "email": "juan@example.com",
        "password": "hashedpassword"
    },
    {
        "id_usuario": 2,
        "nombre": "María López",
        "email": "maria@example.com",
        "password": "hashedpassword"
    }
]
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Internal Server Error", "error": "Detalles del error" }
```

---

2. `createUser`

Descripción:
Crea un nuevo usuario en la base de datos.

Ruta:
`POST /api/usuarios`

Parámetros (body):

- `nombre` (string, requerido): Nombre del nuevo usuario.
- `email` (string, requerido): Correo electrónico del nuevo usuario.
- `password` (string, requerido): Contraseña del nuevo usuario.

Respuesta Exitosa (201):
Retorna un mensaje de éxito junto con el `id` y `nombre` del usuario creado.

```
json
{
    "id": 5,
    "nombre": "Carlos Gómez"
```

```
}
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Internal Server Error", "error": "Detalles del error" }
```

---

3. `updateUser`

Descripción:
Actualiza el nombre de un usuario existente en la base de datos.

Ruta:
`PUT /api/usuarios/:id`

Parámetros (ruta):

- `id` (integer, requerido): ID del usuario a actualizar.

Parámetros (body):

- `nombre` (string, requerido): El nuevo nombre del usuario.

Respuesta Exitosa (200):
Retorna el `id` y el nuevo `nombre` del usuario actualizado.

```
json
{
    "id": 3,
    "nombre": "Ana García"
}
```

Respuesta de Error (404):
Si no se encuentra el usuario, devuelve:

```
json
{ "message": "Usuario no encontrado" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Internal Server Error", "error": "Detalles del error" }
```

---

4. `deleteUser`

Descripción:
Elimina un usuario de la base de datos mediante su `id`.

Ruta:
`DELETE /api/usuarios/:id`

Parámetros (ruta):

- `id` (integer, requerido): ID del usuario a eliminar.

Respuesta Exitosa (204):
Retorna un estado vacío con código 204 para indicar que el usuario fue eliminado exitosamente.

Respuesta de Error (404):
Si no se encuentra el usuario, devuelve:

```
json
{ "message": "Usuario no encontrado" }
```

Respuesta de Error (500):
Si ocurre un error interno en el servidor, devuelve el siguiente mensaje:

```
json
{ "message": "Internal Server Error", "error": "Detalles del error" }
```

Rutas

Documentación del archivo `activo.routes.js`

Descripción del archivo

El archivo `activo.routes.js` define las rutas del sistema para interactuar con los recursos relacionados con "activos". Estas rutas permiten realizar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos de activos.

Contenido

1. **Importaciones:**
    - Se importa el framework `express` para definir las rutas.
    - Se importan las funciones del controlador `activo.controller.js` que gestionan las operaciones sobre los activos:

      ```
      javascript
      import { getActivo, getActivos, createActivo,
      updateActivo, deleteActivo } from
      '../controllers/activo.controller.js';
      ```

2. **Instanciación del router:**
    - Se crea una instancia de un router de Express:

```
javascript
const router = express.Router();
```

3. **Definición de rutas:**
    - **Obtener todos los activos:**
        - Ruta: `GET /activos`
        - Llama al controlador `getActivos` para obtener todos los activos.

```
javascript
router.get('/activos', getActivos);
```

    - **Crear un nuevo activo:**
        - Ruta: `POST /activos`
        - Llama al controlador `createActivo` para crear un nuevo activo.

```
javascript
router.post('/activos', createActivo);
```

    - **Actualizar un activo existente:**
        - Ruta: `PUT /activos/:id`
        - Llama al controlador `updateActivo` para actualizar un activo con un `id` específico.

```
javascript
router.put('/activos/:id', updateActivo);
```

    - **Eliminar un activo:**
        - Ruta: `DELETE /activos/:id`
        - Llama al controlador `deleteActivo` para eliminar un activo con un `id` específico.

```
javascript
router.delete('/activos/:id', deleteActivo);
```

    - **Obtener un activo por su ID:**
        - Ruta: `GET /activos/:id`
        - Llama al controlador `getActivo` para obtener un activo específico con un `id`.

```
javascript
router.get('/activos/:id', getActivo);
```

4. **Exportación del router:**
    - Se exporta el router para que pueda ser utilizado en otras partes de la aplicación, generalmente en el archivo principal de rutas (como `app.js` o `index.js`).

```
javascript
export default router;
```

Documentación del archivo `activostareas.routes.js`

Descripción del archivo

El archivo `activostareas.routes.js` define la ruta para obtener las tareas asociadas a un activo específico. En este caso, la única ruta disponible es para obtener las tareas relacionadas con los activos.

Contenido

1. **Importaciones:**
   - Se importa el framework `express` para definir las rutas.
   - Se importa la función `getactivotareas` del controlador `activostareas.controller.js` que gestiona la obtención de las tareas asociadas a los activos:

     ```
     javascript
     import { getactivotareas } from
     '../controllers/activostareas.controller.js';
     ```

2. **Instanciación del router:**
   - Se crea una instancia de un router de Express:

     ```
     javascript
     const router = express.Router();
     ```

3. **Definición de la ruta:**
   - **Obtener tareas asociadas a los activos:**
     - Ruta: `GET /activo-tareas`
     - Llama al controlador `getactivotareas` para obtener las tareas asociadas a los activos.

     ```
     javascript
     router.get('/activo-tareas', getactivotareas);
     ```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser utilizado en otras partes de la aplicación, generalmente en el archivo principal de rutas (como `app.js` o `index.js`).

```
javascript
export default router;
```

Documentación del archivo `autenticacion.routes.js`

Descripción del archivo

Este archivo define las rutas para la autenticación de usuarios, incluyendo el registro, inicio de sesión y acceso a rutas protegidas mediante JWT (JSON Web Token). También utiliza bcrypt para encriptar las contraseñas y gestionar la autenticación.

Contenido

1. **Importaciones:**
   - `express`: Framework para crear las rutas.
   - `bcrypt`: Utilizado para encriptar las contraseñas.
   - `jsonwebtoken`: Para generar y verificar los tokens JWT.
   - `base_conexion`: Conexión a la base de datos (probablemente configurada en un archivo `db.js`).
   - `authenticateToken`: Middleware que protege rutas, asegurando que el usuario esté autenticado.

```
javascript
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const base_conexion = require('../config/db');
const authenticateToken =
require('../middlewares/authMiddleware');
```

2. **Instanciación del router:** Se crea una instancia de un router de Express:

```
javascript
const router = express.Router();
```

3. **Rutas definidas:**
   - **Registro de usuario (`/register`):**
     - **Método:** `POST`
     - **Descripción:** Registra a un nuevo usuario. La contraseña se encripta antes de almacenarla en la base de datos.
     - **Datos esperados en el cuerpo de la solicitud:** `nombre`, `email`, `password`
     - **Lógica:**
       - Encripta la contraseña usando `bcrypt.hash`.
       - Inserta el nuevo usuario en la base de datos con `base_conexion.query`.
       - Responde con el `nombre` y `email` del usuario registrado.

```
javascript
router.post('/register', async (req, res) => {
    const { nombre, email, password } = req.body;
    const hashedPassword = await bcrypt.hash(password,
10);
    const query = 'INSERT INTO usuario (nombre, email,
password) VALUES (?, ?, ?)';
```

```
    base_conexion.query(query, [nombre, email,
hashedPassword], (error, result) => {
        if (error) {
            console.error('Error al registrar usuario: ' +
error);
            res.json({ error: 'Error al registrar usuario'
});
            return;
        }
        res.json({ nombre, email });
    });
});
```

- **Inicio de sesión de usuario (`/login`):**
  - **Método:** `POST`
  - **Descripción:** Inicia sesión de un usuario, comparando la contraseña ingresada con la almacenada en la base de datos. Si las credenciales son correctas, se genera un token JWT.
  - **Datos esperados en el cuerpo de la solicitud:** `email`, `password`
  - **Lógica:**
    - Busca al usuario en la base de datos por `email`.
    - Compara la contraseña ingresada con la almacenada en la base de datos usando `bcrypt.compare`.
    - Si las credenciales son correctas, se genera un JWT usando `jwt.sign`.
    - El token tiene una expiración de 1 hora.

```
javascript
router.post('/login', async (req, res) => {
    const { email, password } = req.body;
    const query = 'SELECT * FROM usuario WHERE email = ?';
    base_conexion.query(query, [email], async (error,
results) => {
        if (error) {
            console.error('Error al buscar usuario: ' +
error);
            res.status(500).json({ error: 'Error al buscar
usuario' });
            return;
        }
        if (results.length === 0) {
            res.status(401).json({ error: 'Usuario no
encontrado' });
            return;
        }
        const user = results[0];
        try {
            const isMatch = await bcrypt.compare(password,
user.password);
            if (!isMatch) {
                res.status(401).json({ error: 'Contraseña
incorrecta' });
                return;
            }
            const token = jwt.sign({ nombre: user.nombre
}, SECRET_KEY, { expiresIn: '1h' });
            res.json({ token });
        } catch (err) {
```

```
            console.error('Error al comparar contraseñas:
' + err);
            res.status(500).json({ error: 'Error al
comparar contraseñas' });
        }
    });
});
```

- **Ruta protegida (`/protected`):**
  - **Método:** `GET`
  - **Descripción:** Esta ruta está protegida por el middleware `authenticateToken`. Solo los usuarios autenticados con un token JWT válido pueden acceder.
  - **Lógica:** Si el token es válido, se devuelve un mensaje de acceso permitido con los datos del usuario.

```
javascript
router.get('/protected', authenticateToken, (req, res) =>
{
    res.json({ message: 'Acceso a ruta protegida
concedido', user: req.user });
});
```

4. **Exportación del router:**
   - Se exporta el router para ser utilizado en el archivo principal de la aplicación, como `app.js` o `index.js`.

```
javascript
module.exports = router;
```

Documentación del archivo `datous.routes.js`

Descripción del archivo

Este archivo define las rutas para obtener datos relacionados con operarios, edificios, pisos y sectores. Las rutas hacen uso de los controladores correspondientes para obtener la información desde la base de datos.

Contenido

1. **Importaciones:**
   - Se importan las funciones `getOperarios`, `getEdificios`, `getPisos`, y `getSectores` del controlador `datos.controller.js`.

```
javascript
import { getOperarios, getEdificios, getPisos, getSectores  }
from '../controllers/datos.controller.js';
```

2. **Instanciación del router:** Se crea una instancia de un router de Express:

```
javascript
const router = express.Router();
```

3. **Rutas definidas:**
   - **Obtener operarios (`/operarios`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene la lista de operarios de la base de datos.
     - **Controlador:** `getOperarios`

```
javascript
router.get('/operarios', getOperarios);
```

   - **Obtener edificios (`/edificios`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene la lista de edificios de la base de datos.
     - **Controlador:** `getEdificios`

```
javascript
router.get('/edificios', getEdificios);
```

   - **Obtener pisos (`/pisos`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene la lista de pisos de la base de datos.
     - **Controlador:** `getPisos`

```
javascript
router.get('/pisos', getPisos);
```

   - **Obtener sectores (`/sectores`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene la lista de sectores de la base de datos.
     - **Controlador:** `getSectores`

```
javascript
router.get('/sectores', getSectores);
```

4. **Exportación del router:**
   - Se exporta el router para ser utilizado en el archivo principal de la aplicación, como `app.js` o `index.js`.

```
javascript
export default router;
```

Documentación del archivo `edificiospisos.routes.js`

Descripción del archivo

Este archivo define una ruta para obtener los pisos asociados a un edificio específico. La ruta utiliza el controlador `obtenerPisosPorEdificio` para obtener los datos desde la base de datos en función del `id_edificio` proporcionado en la URL.

Contenido

1. **Importaciones:**
   - Se importa la función `obtenerPisosPorEdificio` desde el controlador `edificiospisos.controller.js`.

```
javascript
import { obtenerPisosPorEdificio } from
'../controllers/edificiospisos.controller.js';
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express:

```
javascript
const router = express.Router();
```

3. **Ruta definida:**
   - **Obtener pisos por edificio (`/edificios/:id_edificio/pisos`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene los pisos asociados a un edificio específico, identificado por el `id_edificio` en la URL.
     - **Controlador:** `obtenerPisosPorEdificio`

```
javascript
router.get('/edificios/:id_edificio/pisos',
obtenerPisosPorEdificio);
```

4. **Exportación del router:**
   - Se exporta el router para ser utilizado en el archivo principal de la aplicación, como `app.js` o `index.js`.

```
javascript
export default router;
```

Documentación del archivo `edificios.routes.js`

Descripción del archivo

Este archivo define las rutas para manejar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) relacionadas con los edificios en la aplicación. Utiliza las funciones del controlador `edificios.controller.js` para interactuar con la base de datos y realizar las acciones correspondientes.

Contenido

1. **Importaciones:**
   - Se importa el `Router` de Express para definir las rutas y las funciones del controlador `edificios.controller.js` para las operaciones en la base de datos.

```
javascript
import { Router } from 'express';
import { getEdificios, createEdificio, deleteEdificio,
updateEdificio } from '../controllers/edificios.controller.js';
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para definir las rutas.

```
javascript
const router = Router();
```

3. **Rutas definidas:**
   - **Obtener edificios (`/edificio`):**
     - **Método:** `GET`
     - **Descripción:** Obtiene la lista de todos los edificios.
     - **Controlador:** `getEdificios`

```
javascript
router.get('/edificio', getEdificios);
```

   - **Crear edificio (`/edificio`):**
     - **Método:** `POST`
     - **Descripción:** Crea un nuevo edificio en la base de datos.
     - **Controlador:** `createEdificio`

```
javascript
router.post('/edificio', createEdificio);
```

   - **Eliminar edificio (`/edificio/:id`):**
     - **Método:** `DELETE`
     - **Descripción:** Elimina un edificio de la base de datos por su `id`.
     - **Controlador:** `deleteEdificio`

```
javascript
router.delete('/edificio/:id', deleteEdificio);
```

   - **Actualizar edificio (`/edificio/:id_edificio`):**
     - **Método:** `PUT`
     - **Descripción:** Actualiza los datos de un edificio por su `id_edificio`.
     - **Controlador:** `updateEdificio`

```
javascript
router.put('/edificio/:id_edificio', updateEdificio);
```

4. **Exportación del router:**
   - Se exporta el router para ser utilizado en el archivo principal de la aplicación.

```
javascript
```

```
export default router;
```

Documentación del archivo `login.routes.js`

Descripción del archivo

Este archivo define las rutas para manejar el registro, login, logout y la verificación de autenticación de los usuarios. Utiliza las funciones del controlador `login.controller.js` para realizar las operaciones correspondientes, tales como la creación de usuarios, la validación de credenciales y la protección de rutas mediante JWT.

Contenido

1. **Importaciones:**
   - Se importa el `Router` de Express para definir las rutas y las funciones del controlador `login.controller.js` para las operaciones relacionadas con la autenticación.

```
javascript
import { Router } from "express";
import { login, registro, logout, verifyToken, protect } from
'../controllers/login.controller.js';
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para definir las rutas.

```
javascript
const router = Router();
```

3. **Rutas definidas:**
   - **Registro de usuario (`/registro`):**
     - **Método:** `POST`
     - **Descripción:** Crea un nuevo usuario en el sistema.
     - **Controlador:** `registro`

```
javascript
router.post('/registro', registro);
```

   - **Login de usuario (`/login`):**
     - **Método:** `POST`
     - **Descripción:** Verifica las credenciales de un usuario y devuelve un token JWT.
     - **Controlador:** `login`

```
javascript
router.post('/login', login);
```

   - **Logout de usuario (`/logout`):**

        - **Método:** `POST`
        - **Descripción:** Finaliza la sesión de un usuario, generalmente invalidando el token en el lado del cliente.
        - **Controlador:** `logout`

```
javascript
router.post('/logout', logout);
```

    - **Ruta protegida (`/protected`):**
        - **Método:** `GET`
        - **Descripción:** Ruta que requiere autenticación mediante JWT, verifica el token antes de permitir el acceso a la información protegida.
        - **Controlador:** `verifyToken` (para validar el token) y `protect` (para manejar el acceso a la ruta).

```
javascript
router.get('/protected', verifyToken, protect);
```

4. **Exportación del router:**
    - Se exporta el router para ser utilizado en el archivo principal de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `ordenTrabajo.routes.js`

Descripción del archivo

Este archivo define las rutas relacionadas con las órdenes de trabajo (ODT) en la aplicación. Estas rutas permiten obtener, crear, actualizar y eliminar órdenes de trabajo, así como obtener detalles y concatenaciones de las órdenes.

Contenido

1. **Importaciones:**
    - Se importa el `Router` de Express para definir las rutas y las funciones del controlador `ordenTrabajo.controller.js` para manejar las operaciones relacionadas con las órdenes de trabajo.

```
javascript
import { Router } from 'express';
import { getOrdenTrabajo, getDetallesOrdenTrabajo, nuevaODT,
deleteOrdenTrabajo, getConcatenacionIds, actualizarRealizada }
from '../controllers/ordenTrabajo.controller.js';
```

2. **Instanciación del router:**
    - Se crea una instancia del router de Express para definir las rutas.

```
javascript
```

```
const router = Router();
```

3. **Rutas definidas:**
    - **Obtener todas las órdenes de trabajo (`/orden_trabajo`):**
        - **Método:** `GET`
        - **Descripción:** Devuelve una lista de todas las órdenes de trabajo registradas en el sistema.
        - **Controlador:** `getOrdenTrabajo`

```
javascript
router.get('/orden_trabajo', getOrdenTrabajo);
```

    - **Obtener detalles de una orden de trabajo (`/detalle_orden_trabajo/:id`):**
        - **Método:** `GET`
        - **Descripción:** Devuelve los detalles de una orden de trabajo específica, identificada por su `id`.
        - **Controlador:** `getDetallesOrdenTrabajo`

```
javascript
router.get('/detalle_orden_trabajo/:id',
getDetallesOrdenTrabajo);
```

    - **Crear una nueva orden de trabajo (`/nuevaODT`):**
        - **Método:** `POST`
        - **Descripción:** Crea una nueva orden de trabajo en el sistema.
        - **Controlador:** `nuevaODT`

```
javascript
router.post('/nuevaODT', nuevaODT);
```

    - **Eliminar una orden de trabajo (`/orden_trabajo/:id`):**
        - **Método:** `DELETE`
        - **Descripción:** Elimina una orden de trabajo específica, identificada por su `id`.
        - **Controlador:** `deleteOrdenTrabajo`

```
javascript
router.delete('/orden_trabajo/:id', deleteOrdenTrabajo);
```

    - **Obtener la concatenación de IDs de una orden de trabajo (`/orden-trabajo/:id/concatenacion`):**
        - **Método:** `GET`
        - **Descripción:** Devuelve una concatenación de los IDs relacionados con una orden de trabajo.
        - **Controlador:** `getConcatenacionIds`

```
javascript
router.get('/orden-trabajo/:id/concatenacion',
getConcatenacionIds);
```

    - **Actualizar estado de la orden de trabajo a "realizada" (`/orden-trabajo/:id/realizada`):**

        - **Método:** `PUT`
        - **Descripción:** Actualiza el estado de una orden de trabajo a "realizada".
        - **Controlador:** `actualizarRealizada`

```
javascript
router.put('/orden-trabajo/:id/realizada', actualizarRealizada);
```

4. **Exportación del router:**
    - Se exporta el router para ser utilizado en el archivo principal de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `pisos.routes.js`

Descripción del archivo

Este archivo define las rutas relacionadas con la gestión de los pisos en la aplicación. Permite obtener la lista de pisos, agregar nuevos y eliminar pisos específicos.

Contenido

1. **Importaciones:**
    - Se importa `express` para usar el router de Express y se importan las funciones del controlador `pisos.controller.js` para manejar las operaciones.

```
javascript
import express from 'express';
import { getPisos, createPiso, deletePiso } from
'../controllers/pisos.controller.js'; // Asegúrate de que la
ruta sea correcta
```

2. **Instanciación del router:**
    - Se crea una instancia del router de Express.

```
javascript
const router = express.Router();
```

3. **Rutas definidas:**
    - **Obtener todos los pisos (`/pisos`):**
        - **Método:** `GET`
        - **Descripción:** Devuelve una lista de todos los pisos registrados en el sistema.
        - **Controlador:** `getPisos`

```
javascript
router.get('/pisos', getPisos);
```

    - **Crear un nuevo piso (`/pisos`):**
        - **Método:** `POST`

- **Descripción:** Agrega un nuevo piso al sistema.
- **Controlador:** `createPiso`

```
javascript
router.post('/pisos', createPiso);
```

- **Eliminar un piso específico (`/pisos/:id`):**
  - **Método:** `DELETE`
  - **Descripción:** Elimina un piso específico, identificado por su `id`.
  - **Controlador:** `deletePiso`

```
javascript
router.delete('/pisos/:id', deletePiso);
```

4. **Exportación del router:**
   - Se exporta el router para ser utilizado en el archivo principal de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `sectores.routes.js`

Descripción del archivo

Este archivo define las rutas necesarias para la gestión de sectores en la aplicación, permitiendo obtener la lista de sectores, crear nuevos sectores y eliminar sectores específicos.

Contenido

1. **Importaciones:**
   - Se importa `express` para el uso de `Router`.
   - Se importan las funciones del controlador `sector.controller.js` que manejan la lógica de negocio para cada operación.

```
javascript
import express from 'express';
import { getSectores, createSector, deleteSector } from
'../controllers/sector.controller.js'; // Importar el
controlador
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para manejar las rutas.

```
javascript
const router = express.Router();
```

3. **Rutas definidas:**
   - **Obtener todos los sectores (`/sector`):**

- **Método:** `GET`
- **Descripción:** Devuelve una lista de todos los sectores registrados en el sistema.
- **Controlador:** `getSectores`

```
javascript
router.get('/sector', getSectores);
```

- **Crear un nuevo sector (`/sector`):**
  - **Método:** `POST`
  - **Descripción:** Agrega un nuevo sector al sistema.
  - **Controlador:** `createSector`

```
javascript
router.post('/sector', createSector);
```

- **Eliminar un sector (`/sector`):**
  - **Método:** `DELETE`
  - **Descripción:** Elimina un sector identificado por la información enviada en el cuerpo de la solicitud.
  - **Controlador:** `deleteSector`

```
javascript
router.delete('/sector', deleteSector);
```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser utilizado en otros archivos de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `tareas.routes.js`

Descripción del archivo

El archivo `tareas.routes.js` define las rutas necesarias para la gestión de tareas en la aplicación, permitiendo obtener la lista de tareas, crear nuevas tareas y eliminar tareas específicas.

Contenido

1. **Importaciones:**
   - Se importa `express` para utilizar `Router`.
   - Se importan las funciones del controlador `tareas.controller.js` que manejan la lógica de negocio para las operaciones de tareas.

```
javascript
import express from "express";
```

```
import { getTareas, createTarea, deleteTarea } from
"../controllers/tareas.controller.js";
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para manejar las rutas relacionadas con las tareas.

```
javascript
const router = express.Router();
```

3. **Rutas definidas:**
   - **Obtener todas las tareas (`/tareas`):**
     - **Método:** `GET`
     - **Descripción:** Devuelve una lista de todas las tareas registradas en el sistema.
     - **Controlador:** `getTareas`

     ```
     javascript
     router.get('/tareas', getTareas);
     ```

   - **Crear una nueva tarea (`/tareas`):**
     - **Método:** `POST`
     - **Descripción:** Agrega una nueva tarea al sistema.
     - **Controlador:** `createTarea`

     ```
     javascript
     router.post('/tareas', createTarea);
     ```

   - **Eliminar una tarea (`/tareas/:id_tarea`):**
     - **Método:** `DELETE`
     - **Descripción:** Elimina una tarea específica identificada por `id_tarea`.
     - **Controlador:** `deleteTarea`

     ```
     javascript
     router.delete('/tareas/:id_tarea', deleteTarea);
     ```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser utilizado en otros archivos de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `tipoOrden.routes.js`

Descripción del archivo

El archivo `tipoOrden.routes.js` define la ruta necesaria para obtener los tipos de órdenes en la aplicación. Esta funcionalidad permite a los usuarios consultar los diferentes tipos de órdenes disponibles.

Contenido

1. **Importaciones:**
   - Se importa `Router` de `express` para gestionar las rutas.
   - Se importa la función `getTiposOrden` del controlador `tipoOrden.controller.js`, encargada de la lógica de obtención de los tipos de órdenes.

```
javascript
import { Router } from 'express';
import { getTiposOrden } from
'../controllers/tipoOrden.controller.js';  // Asegúrate de que
la ruta sea correcta
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para manejar las rutas de `tipoOrden`.

```
javascript
const router = Router();
```

3. **Definición de la ruta:**
   - **Obtener tipos de órdenes (`/tiposOrden`):**
     - **Método:** `GET`
     - **Descripción:** Devuelve una lista de todos los tipos de órdenes disponibles.
     - **Controlador:** `getTiposOrden`

```
javascript
router.get('/tiposOrden', getTiposOrden);
```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser importado y utilizado en otros archivos de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `ubicacion.routes.js`

Descripción del archivo

El archivo `ubicacion.routes.js` gestiona las rutas relacionadas con las ubicaciones en la aplicación. Permite a los usuarios obtener, crear y eliminar ubicaciones mediante las rutas definidas.

Contenido

1. **Importaciones:**
   - Se importa `express` para crear el router.
   - Se importan las funciones `getUbicaciones`, `createUbicacion` y `deleteUbicacion` desde el controlador `ubicacion.controller.js`, que contienen la lógica de las operaciones respectivas.

```
javascript
import express from 'express';
import { getUbicaciones, createUbicacion, deleteUbicacion } from
'../controllers/ubicacion.controller.js';
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para definir las rutas de `ubicacion`.

```
javascript
const router = express.Router();
```

3. **Definición de las rutas:**
   - **Obtener ubicaciones (`/ubicaciones`):**
     - **Método:** `GET`
     - **Descripción:** Devuelve una lista de todas las ubicaciones disponibles.
     - **Controlador:** `getUbicaciones`

```
javascript
router.get('/ubicaciones', getUbicaciones);
```

   - **Crear una nueva ubicación (`/ubicaciones`):**
     - **Método:** `POST`
     - **Descripción:** Crea una nueva ubicación con los datos proporcionados.
     - **Controlador:** `createUbicacion`

```
javascript
router.post('/ubicaciones', createUbicacion);
```

   - **Eliminar una ubicación específica (`/ubicaciones/:id`):**
     - **Método:** `DELETE`
     - **Descripción:** Elimina una ubicación específica según el `id` proporcionado.
     - **Controlador:** `deleteUbicacion`

```
javascript
router.delete('/ubicaciones/:id', deleteUbicacion);
```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser utilizado en otros módulos de la aplicación.

```
javascript
export default router;
```

Documentación del archivo `usuario.routes.js`

Descripción del archivo

El archivo `usuario.routes.js` define las rutas para gestionar las operaciones relacionadas con los usuarios en la aplicación, como obtener, registrar, actualizar y eliminar usuarios.

Contenido

1. **Importaciones:**
   - Se importa `express` para crear un router de Express.
   - Se importan las funciones `getUsers`, `updateUser`, y `deleteUser` desde el controlador `usuario.controller.js`.
   - Se importa la función `registro` desde el controlador `login.controller.js` para manejar la creación de usuarios.

```
javascript
import express from 'express';
import { getUsers, updateUser, deleteUser } from
'../controllers/usuario.controller.js';
import { registro } from '../controllers/login.controller.js';
```

2. **Instanciación del router:**
   - Se crea una instancia del router de Express para definir las rutas de `usuario`.

```
javascript
const router = express.Router();
```

3. **Definición de las rutas:**
   - **Obtener usuarios (`/usuarios`):**
     - **Método:** `GET`
     - **Descripción:** Devuelve una lista de todos los usuarios registrados.
     - **Controlador:** `getUsers`

```
javascript
router.get('/usuarios', getUsers);
```

   - **Registrar un nuevo usuario (`/usuarios`):**
     - **Método:** `POST`
     - **Descripción:** Crea un nuevo usuario en la base de datos.
     - **Controlador:** `registro`

```
javascript
router.post('/usuarios', registro);
```

   - **Actualizar un usuario existente (`/usuarios/:id`):**
     - **Método:** `PUT`
     - **Descripción:** Actualiza los datos de un usuario específico identificado por `id`.

- **Controlador:** `updateUser`

```
javascript
router.put('/usuarios/:id', updateUser);
```

- **Eliminar un usuario (`/usuarios/:id`):**
  - **Método:** `DELETE`
  - **Descripción:** Elimina un usuario específico de la base de datos según el `id` proporcionado.
  - **Controlador:** `deleteUser`

```
javascript
router.delete('/usuarios/:id', deleteUser);
```

4. **Exportación del router:**
   - Se exporta el router para que pueda ser utilizado en otros módulos de la aplicación.

```
javascript
export default router;
```