vecinos. service:

🔎4️⃣`export async function validarVecino(...)`

📌 **¿Qué hace?**

- Declara una función **asíncrona** (`async`) que valida si ya existe un vecino con ciertos datos.
- Se exporta para ser usada en otros archivos.

📌 **Ejemplo de uso:** En `createVecinoService()`, se llama a esta función para evitar duplicados:

javascript
const vecinoExistente = await validarVecino(nombre, rut, correo, telefono);

🔎5️⃣`const vecinoRepository = AppDataSource.getRepository(Vecino);`

📌 **¿Qué hace?**

- Obtiene un **repositorio** de la entidad `Vecino`, permitiendo hacer consultas en la tabla `vecinos`.

📌 **Ejemplo de consulta:**

javascript
const vecinos = await vecinoRepository.find();

Esto devuelve **todos los registros** de la tabla `vecinos`.

🔎6️⃣`findOne({ where: [{ nombre }, { rut }, { correo }, { telefono }] });`

📌 **¿Qué hace?**

- Busca un vecino que coincida con `nombre`, `rut`, `correo` o `telefono`.

📌 **Ejemplo:**

javascript
const existeVecino = await vecinoRepository.findOne({ where: [{ nombre }, { rut }] });

Esto verifica si ya hay un vecino con el mismo nombre o RUT en la base de datos.

🔎7️⃣ `if (excludeId) {...}` (Exclusión de registros por ID)

📌 ¿Qué hace?

- Si `excludeId` tiene un valor, se excluye ese ID de la búsqueda para evitar conflictos en actualizaciones.

📌 Ejemplo:

javascript
```
if (excludeId) {
  whereClause = whereClause.map(condition => ({ ...condition, id: { $ne: excludeId } }));
}
```

Si estás actualizando un vecino, **evitas que su propio ID sea considerado un duplicado**.

🔎8️⃣ `create()` (Crear un nuevo objeto antes de guardarlo)

📌 ¿Qué hace?

- Crea un objeto en memoria con los datos del vecino antes de guardarlo en la base de datos.

📌 Ejemplo:

javascript
```
const newVecino = vecinoRepository.create({
  nombre,
  rut,
  correo,
  telefono,
  comprobanteDomicilio,
});
```

Esto **NO guarda el vecino todavía**, solo crea el objeto.

🔎9️⃣ `save()` (Guardar un registro en la base de datos)

📌 ¿Qué hace?

- Guarda un nuevo vecino en la base de datos o actualiza uno existente.

📌 Ejemplo:

javascript
```
await vecinoRepository.save(newVecino);
```

Ahora el vecino se almacena en la tabla.

🔎 🔟 `update()` (Actualizar un registro en la base de datos)

📌 ¿Qué hace?

- Actualiza un vecino existente con nuevos datos.

📌 Ejemplo:

javascript
```
await vecinoRepository.update({ id: vecinoFound.id }, dataVecinoUpdate);
```

Esto modifica los datos del vecino **sin eliminarlo**.

🔎 🔟 `remove()` (Eliminar un registro de la base de datos)

📌 ¿Qué hace?

- Borra un vecino de la tabla.

📌 Ejemplo:

javascript
```
await vecinoRepository.remove(vecinoFound);
```

Si el vecino existe, **se elimina permanentemente**.

---

vecino.controller:

🔎3️⃣`import { vecinoBodyValidation, vecinoQueryValidation } from "../validations/vecinos.validation.js";`

📌 ¿Qué hace?

- Importa reglas de validación para los datos de los vecinos.
- Usa **Joi u otra librería de validación** para comprobar que los datos sean correctos antes de guardarlos.

📌 Ejemplo de validación:

javascript
```
const { error } = vecinoBodyValidation.validate({ nombre, rut, correo, telefono });
if (error) {
   return handleErrorClient(res, 400, error.message);
```

}

Si `rut` o `correo` son inválidos, la función **devuelve un error antes de guardar los datos**.

🔎4️⃣`export async function createVecino(req, res) {...}` (Definición de función de controlador)

📌 ¿Qué hace?

- Define una **función asíncrona** que maneja una solicitud `POST` para registrar vecinos.
- Usa `req.body` para obtener datos del usuario y `req.file` para manejar archivos adjuntos.

📌 Ejemplo clave:

javascript
const { nombre, rut, correo, telefono } = req.body;
const comprobantePath = req.file ? req.file.path : null;

Aquí, `req.file.path` **almacena la ruta del comprobante de domicilio**, si el usuario subió uno.

🔎5️⃣`const { nombre, rut, correo, telefono } = req.body;` (Desestructuración de objetos)

📌 ¿Qué hace?

- Extrae datos del cuerpo de la solicitud (`req.body`) y los guarda en variables individuales.

📌 **Ejemplo:** Si la solicitud `POST /vecinos` contiene:

json
{
  "nombre": "Juan",
  "rut": "12.345.678-9",
  "correo": "juan@example.com",
  "telefono": "987654321"
}

La línea:

javascript
const { nombre, rut, correo, telefono } = req.body;

Guardará los valores en **variables separadas**:

- `nombre = "Juan"`
- `rut = "12.345.678-9"`
- `correo = "juan@example.com"`
- `telefono = "987654321"`

🔎6️⃣`const [newVecino, errorCreateVecino] = await createVecinoService({...});`

📌 **¿Qué hace?**

- Llama a `createVecinoService()`, que interactúa con la base de datos para **crear un vecino**.
- Usa **desestructuración de arrays** para obtener **el vecino creado (`newVecino`) y un posible error (`errorCreateVecino`)**.

📌 **Ejemplo:** Si la operación es exitosa, `newVecino` contendrá el vecino registrado. Si hay un error (por ejemplo, **el RUT ya está en uso**), `errorCreateVecino` tendrá un mensaje de error.

🔎7️⃣`if (errorCreateVecino) return handleErrorClient(res, 400, errorCreateVecino);`

📌 **¿Qué hace?**

- Si `errorCreateVecino` tiene un valor (es decir, hubo un problema), **devuelve un error 400 al cliente**.

📌 **Ejemplo:** Si intentas registrar un vecino con un RUT duplicado, la respuesta sería:

json
{
  "error": "⚠️ Ya existe un vecino registrado con este RUT."
}

Esto evita que se guarde información incorrecta en la base de datos.

🔎8️⃣`handleSuccess(res, 201, "Vecino creado exitosamente", newVecino);`

📌 **¿Qué hace?**

- Usa una función externa (`handleSuccess`) para enviar una **respuesta exitosa al cliente**.
- Código `201` indica que un recurso fue **creado correctamente**.

📌 **Ejemplo:** Si el vecino se registra con éxito, la API responderá con:

```
json
{
  "message": "Vecino creado exitosamente",
  "data": {
    "nombre": "Juan",
    "rut": "12.345.678-9",
    "correo": "juan@example.com",
    "telefono": "987654321"
  }
}
```

🔎9️⃣`catch (error) { handleErrorServer(res, 500, error.message); }`

📌 **¿Qué hace?**

- Captura errores inesperados y devuelve un **error 500 (interno del servidor)**.

📌 **Ejemplo:** Si la base de datos se desconecta, la API enviará:

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

Esto evita que **el servidor se caiga** por errores inesperados.

🔎 🔟 `getVecinoService(), getVecinosService()` (Obtener vecinos de la base de datos)

📌 **¿Qué hacen?**

- `getVecinoService(query)`: Busca **un vecino** según RUT, ID o correo.
- `getVecinosService()`: Obtiene **todos los vecinos registrados**.

📌 **Ejemplo de uso:** Si llamas a `GET /vecinos?rut=12.345.678-9`, la función devolverá el vecino correspondiente.

🔎 🔟 `updateVecinoService(), deleteVecinoService()` (Actualizar y eliminar vecinos)

📌 ¿Qué hacen?

- `updateVecinoService(query, body)`: **Modifica** datos de un vecino.
- `deleteVecinoService(query)`: **Elimina** un vecino de la base de datos.

📌 Ejemplo en `updateVecinoService()`:

javascript
const [vecino, vecinoError] = await updateVecinoService({ rut, id, correo }, body);

Si `vecinoError` tiene un valor, significa que **hubo un problema al actualizar**.

_____-

reunion.controller:

🔎1️⃣`"use strict";` (Modo estricto en JavaScript)

📌 ¿Qué hace?

- Activa el **modo estricto**, lo que previene errores como el uso de variables sin declarar o asignaciones problemáticas.
- Es una **buena práctica** en proyectos de Node.js para evitar errores silenciosos.

🔎2️⃣`import { createReunionService, getReunionesService, ... } from "../services/reuniones.service.js";`

📌 ¿Qué hace?

- **Importa funciones del servicio** `reuniones.service.js`, que interactúa con la base de datos.
- Se usa en los controladores para realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar).

📌 Ejemplo de uso:

javascript
const [nuevaReunion, errorCreateReunion] = await createReunionService({ nombre, fecha });

Aquí, `createReunionService()` guarda la reunión en la base de datos.

🔎3️⃣`import { handleErrorClient, handleErrorServer, handleSuccess } from "../handlers/responseHandlers.js";`

📌 **¿Qué hace?**

- **Importa funciones para manejar respuestas HTTP**, estandarizando el formato de los mensajes.
- Mejora la organización del código evitando respuestas HTTP directas dentro de los controladores.

📌 **Ejemplo:**

javascript
handleSuccess(res, 201, "Reunión creada exitosamente", nuevaReunion);

Esta función envía una respuesta con estado `201` (creación exitosa).

🔎4️⃣`export async function createReunion(req, res) {...}` (Función asíncrona que maneja una solicitud `POST`)

📌 **¿Qué hace?**

- **Recibe una solicitud** `POST`, extrae datos del cuerpo (`req.body`) y crea una reunión.
- **Valida los datos**, evita duplicados y guarda el registro en la base de datos.

📌 **Ejemplo clave:**

javascript
const { nombre, fecha } = req.body;
if (!nombre || !fecha) {
  return handleErrorClient(res, 400, "Nombre y fecha son obligatorios.");
}

Si `nombre` o `fecha` faltan, **devuelve un error antes de continuar**.

🔎5️⃣`const esDuplicada = await validarReunion(nombre, fecha);`

📌 **¿Qué hace?**

- **Verifica si la reunión ya existe en la base de datos** antes de crearla.

📌 **Ejemplo:**

javascript

```
if (esDuplicada) {
   return handleErrorClient(res, 400, "❌ Ya existe una reunión con este nombre y fecha.");
}
```

Si la reunión **ya está registrada**, **devuelve un error y evita la duplicación**.

🔎6️⃣`const { id } = req.params;` (Extraer parámetros de la URL)

📌 **¿Qué hace?**

- Obtiene el `id` desde `req.params`, útil en rutas como `DELETE /reuniones/:id`.

📌 **Ejemplo:** Si llamas a `DELETE /reuniones/123`, entonces `req.params` contendrá:

json
```
{ "id": "123" }
```

Y el código:

javascript
```
const { id } = req.params;
```

Extraerá `id = "123"`.

🔎7️⃣`const actaPath = req.file ? req.file.path : null;`

📌 **¿Qué hace?**

- Usa un **operador ternario** (`? :`) para definir `actaPath`.
- Si `req.file` existe, obtiene la ruta del archivo (`req.file.path`).
- Si `req.file` es `undefined`, asigna `null`.

📌 **Ejemplo:** Si el usuario sube `reunion.pdf`, `req.file` tendrá:

json
```
{
  "fieldname": "acta",
  "originalname": "reunion.pdf",
  "path": "uploads/actas/reunion.pdf"
}
```

Entonces:

javascript

const actaPath = req.file ? req.file.path : null;

Guardará `"uploads/actas/reunion.pdf"`.

Si el usuario **no sube un archivo**, `actaPath = null`.

🔎8️⃣`await deleteReunionService(id);`

📌 **¿Qué hace?**

- Llama a `deleteReunionService(id)`, que busca la reunión por `id` y la elimina de la base de datos.

📌 **Ejemplo:** Si la reunión existe, **se eliminará permanentemente** con:

javascript
await reunionRepository.remove(reunion);

Si no existe, devolverá `"Reunión no encontrada"`.

🔎9️⃣`reunion.nombre = updatedData.nombre || reunion.nombre;`

📌 **¿Qué hace?**

- Usa `||` (operador OR) para **mantener el valor existente si** `updatedData.nombre` **está vacío**.
- Si `updatedData.nombre` tiene datos, **los asigna** a `reunion.nombre`.

📌 **Ejemplo:** Si `updatedData.nombre = "Nueva reunión"`, entonces:

javascript
reunion.nombre = updatedData.nombre || reunion.nombre;

`reunion.nombre` será `"Nueva reunión"`.

Si `updatedData.nombre` es `undefined`, **mantendrá el nombre anterior**.

---

reunion.service;

🔎2️⃣`import { AppDataSource } from "../config/configDb.js";` (Importación de módulos)

📌 **¿Qué hace?**

- Importa la instancia de conexión con la base de datos (`AppDataSource`).
- Se usa para obtener un repositorio y ejecutar consultas en la tabla `reuniones`.

📌 **Ejemplo de uso:** Cada función obtiene el repositorio con:

javascript
const reunionRepository = AppDataSource.getRepository(Reunion);

Esto permite interactuar con la base de datos.

🔎3️⃣`import Reunion from "../entity/reunion.entity.js";`

📌 **¿Qué hace?**

- Importa la **entidad** `Reunion`, que representa la estructura de la tabla `reuniones` en TypeORM.
- Define campos como `id`, `nombre`, `fecha`, `acta`, etc.

📌 **Ejemplo:** Si `Reunion` tiene la siguiente estructura:

javascript
```
const Reunion = new EntitySchema({
    name: "Reunion",
    tableName: "reuniones",
    columns: {
        id: { primary: true, type: "uuid", generated: "uuid" },
        nombre: { type: "varchar", length: 255, nullable: false },
        fecha: { type: "timestamp", nullable: false },
    }
});
```

El código puede interactuar con la tabla `reuniones` usando `getRepository(Reunion)`.

🔎4️⃣`export async function validarReunion(nombre, fecha) {...}`

📌 **¿Qué hace?**

- Define una función **asíncrona** (`async`) que verifica si ya existe una reunión con el mismo `nombre` y `fecha`.

📌 **Ejemplo clave:**

javascript

```
const existeReunion = await reunionRepository.findOneBy({ nombre, fecha });
return !!existeReunion;
```

Si `existeReunion` es `null`, devuelve `false`. Si encuentra una reunión, devuelve `true`.

🔎5️⃣`export async function createReunionService({ nombre, fecha }) {...}`

📌 ¿Qué hace?

- Registra una nueva reunión en la base de datos si no hay duplicados.

📌 Ejemplo de validación:

```
javascript
const esDuplicada = await validarReunion(nombre, fecha);
if (esDuplicada) {
   return [null, "❌ Ya existe una reunión con este nombre y fecha."];
}
```

Si una reunión ya existe con el mismo `nombre` y `fecha`, **devuelve un error antes de guardarla**.

📌 Ejemplo de creación:

```
javascript
const nuevaReunion = reunionRepository.create({ nombre, fecha });
await reunionRepository.save(nuevaReunion);
```

Esto **crea y guarda la reunión** en la base de datos.

🔎6️⃣`getReunionesService()` (Obtener todas las reuniones)

📌 ¿Qué hace?

- Consulta la base de datos y devuelve una lista de todas las reuniones registradas.

📌 Ejemplo:

```
javascript
const reuniones = await reunionRepository.find();
return [reuniones, null];
```

Si no hay reuniones, devuelve un array vacío.

🔎7️⃣`uploadActaService(id, actaPath)` (Subir acta a una reunión)

📌 **¿Qué hace?**

- Encuentra una reunión por `id` y guarda la ruta del acta (`actaPath`) en la base de datos.

📌 **Ejemplo:**

javascript
const reunion = await reunionRepository.findOneBy({ id });
if (!reunion) return [null, "Reunión no encontrada"];

reunion.acta = actaPath;
await reunionRepository.save(reunion);

Si `reunion.acta = actaPath`, **el acta quedará registrada en la reunión**.

🔎8️⃣`deleteReunionService(id)` (Eliminar una reunión)

📌 **¿Qué hace?**

- Busca una reunión por `id` y la elimina si existe.

📌 **Ejemplo:**

javascript
const reunion = await reunionRepository.findOneBy({ id });
if (!reunion) return [null, "Reunión no encontrada"];

await reunionRepository.remove(reunion);
return ["Reunión eliminada correctamente", null];

Si `reunion` no existe, devuelve `"Reunión no encontrada"`.

🔎9️⃣`updateReunionService(id, updatedData)` (Actualizar una reunión)

📌 **¿Qué hace?**

- Modifica los datos de una reunión existente, sin borrar la información anterior.

📌 **Ejemplo:**

javascript
reunion.nombre = updatedData.nombre || reunion.nombre;

reunion.fecha = updatedData.fecha || reunion.fecha;
await reunionRepository.save(reunion);

Si `updatedData.nombre` tiene un valor, **lo actualiza**, si no, mantiene el antiguo.

_____-

upload:

🔎1️⃣Importaciones

📌 ¿Qué hace esto?

javascript
import multer from "multer";
import fs from "fs";
import path from "path";

✅ Explicación:

- `multer`: Middleware de Node.js para manejar la subida de archivos.
- `fs` **(File System)**: Permite manipular archivos y directorios del sistema.
- `path`: Facilita la construcción de rutas de archivos en cualquier sistema operativo.

🔎2️⃣Definir la carpeta donde se guardarán las actas

📌 ¿Qué hace esto?

javascript
const actasDir = path.join("uploads", "actas");

✅ Explicación:

- `path.join("uploads", "actas")` **crea una ruta válida**, asegurando compatibilidad entre Windows y Linux/Mac.
- La variable `actasDir` almacena la carpeta donde se guardarán los archivos.

📌 ¿Para qué se usa `fs.existsSync()` y `fs.mkdirSync()`?

javascript
if (!fs.existsSync(actasDir)) {
  fs.mkdirSync(actasDir, { recursive: true });
  console.log("✅ [Multer] Carpeta creada:", actasDir);
}

✅ **Explicación:**

- `fs.existsSync(actasDir)` verifica si la carpeta ya existe.
- **Si no existe**, `fs.mkdirSync(actasDir, { recursive: true })` la crea.
- `recursive: true` permite crear `uploads` si aún no existe.

Ejemplo: Si `uploads/actas` no está en el sistema, **se crea automáticamente al ejecutar el código**.

🔎3️⃣Configurar `multer.diskStorage()`

📌 ¿Qué hace esto?

```
javascript
const storageActas = multer.diskStorage({
    destination: function (req, file, cb) {
        console.log("📂 [Multer] Guardando archivo en:", actasDir);
        cb(null, actasDir);
    },
    filename: function (req, file, cb) {
        const filename = `${Date.now()}-${file.originalname}`;
        console.log("📂 [Multer] Nombre del archivo asignado:", filename);
        cb(null, filename);
    }
});
```

✅ **Explicación:**

- `destination(req, file, cb)` → Define en qué carpeta se guardarán los archivos (`actasDir`).
- `filename(req, file, cb)` → Renombra el archivo usando `Date.now()` para evitar nombres duplicados.

Ejemplo: Si subes `reunion.pdf`, se guardará como:

uploads/actas/1685748394356-reunion.pdf

🔎4️⃣Filtrar archivos permitidos (`fileFilter`)

📌 ¿Qué hace esto?

```
javascript
const fileFilterActas = (req, file, cb) => {
    console.log("📂 [Multer] Archivo recibido antes de filtrado:", file);
    if (file.mimetype === "application/pdf") {
        cb(null, true);
```

```
    } else {
        console.error("❌ [Multer] Archivo rechazado, solo se permiten PDFs:", file.mimetype);
        cb(new Error("Solo se permiten archivos PDF"), false);
    }
};
```

✅ **Explicación:**

- **Revisa el tipo de archivo con** `file.mimetype`.
- **Si es un PDF (**`application/pdf`**), permite la subida (**`cb(null, true)`**).**
- **Si no es un PDF, lo rechaza (**`cb(new Error("Solo se permiten archivos PDF"), false)`**).**

Ejemplo de comportamiento:

- Si subes `document.pdf` ✅ **permitido**.
- Si subes `imagen.png` ❌ **rechazado** con error `"Solo se permiten archivos PDF"`.

🔎5️⃣ Crear la instancia de `multer` con la configuración

📌 ¿Qué hace esto?

javascript
```
const uploadActas = multer({ storage: storageActas, fileFilter: fileFilterActas });
```

✅ **Explicación:**

- **Crea una instancia de** `multer` con las opciones definidas (`storageActas`, `fileFilterActas`).
- **Esta instancia se usará como middleware en las rutas de subida de archivos**.

Ejemplo de uso en `reuniones.routes.js`:

javascript
```
router.post("/:id/acta", uploadActas.single("acta"), uploadActa);
```

Aquí, **el middleware de** `multer` **se ejecutará antes de** `uploadActa` para interceptar y almacenar el archivo antes de guardarlo en la base de datos.

🔎6️⃣ Exportar `uploadActas` para usarlo en otros archivos

📌 ¿Qué hace esto?

javascript

export default uploadActas;

✅ **Explicación:**

- **Exporta la configuración de subida** para ser usada en `reuniones.routes.js`.

Ejemplo de importación en otro archivo:

javascript
import uploadActas from "../middlewares/upload.js";

Esto permite utilizar `uploadActas` como middleware de subida de archivos en distintas rutas de Express.

---

uploadcomprobante:

🔎1️⃣Importaciones

📌 **¿Qué hace esto?**

javascript
import multer from "multer";
import fs from "fs";
import path from "path";

✅ **Explicación:**

- `multer`: Middleware de Node.js para manejar la subida de archivos.
- `fs` **(File System)**: Permite manipular archivos y directorios del sistema.
- `path`: Facilita el manejo de rutas en el sistema de archivos.

🔎2️⃣Definir la carpeta donde se guardarán los comprobantes

📌 **¿Qué hace esto?**

javascript
const comprobantesDir = path.join("uploads", "comprobantes");

✅ **Explicación:**

- `path.join("uploads", "comprobantes")` une `"uploads"` y `"comprobantes"` en una ruta válida.

- `comprobantesDir` almacena la ruta de la carpeta donde se guardarán los archivos.

📌 ¿Para qué se usa `fs.existsSync()` y `fs.mkdirSync()`?

javascript
```
if (!fs.existsSync(comprobantesDir)) {
    console.log("📂 [Multer] Creando directorio de comprobantes...");
    fs.mkdirSync(comprobantesDir, { recursive: true });
}
```

✅ Explicación:

- `fs.existsSync(comprobantesDir)` verifica si la carpeta ya existe.
- **Si no existe**, `fs.mkdirSync(comprobantesDir, { recursive: true })` la crea.
- `recursive: true` asegura que si `"uploads"` no existe, también se crea.

Ejemplo: Si la carpeta `uploads/comprobantes` no está en el sistema, **se crea automáticamente** al ejecutar el código.

🔎3️⃣ Configurar `multer.diskStorage()`

📌 ¿Qué hace esto?

javascript
```
const storageComprobantes = multer.diskStorage({
    destination: function (req, file, cb) {
        console.log("📂 [Multer] Guardando archivo en:", comprobantesDir);
        cb(null, comprobantesDir);
    },
    filename: function (req, file, cb) {
        const filename = `${Date.now()}-${file.originalname}`;
        console.log("📝 [Multer] Archivo guardado con nombre:", filename);
        cb(null, filename);
    }
});
```

✅ Explicación:

- `destination(req, file, cb)` → Define en qué carpeta se guardarán los archivos (`comprobantesDir`).
- `filename(req, file, cb)` → Renombra el archivo con su nombre original y una marca de tiempo (`Date.now()`) para evitar nombres duplicados.

Ejemplo: Si subes `documento.png`, el archivo guardado será:

uploads/comprobantes/1685748394356-documento.png

🔎4️⃣ Filtrar archivos permitidos (`fileFilter`)

📌 ¿Qué hace esto?

```
javascript
const fileFilterComprobantes = (req, file, cb) => {
    console.log("🔍 [Multer] Validando tipo de archivo:", file.mimetype);
    if (file.mimetype === "image/png") {
        console.log("✅ [Multer] Archivo permitido.");
        cb(null, true);
    } else {
        console.error("❌ [Multer] ERROR: Tipo de archivo no permitido:", file.mimetype);
        cb(new Error("Solo se permiten archivos PNG"), false);
    }
};
```

✅ Explicación:

- **Verifica el tipo de archivo con** `file.mimetype`.
- **Si el archivo es un PNG (**`image/png`**), permite la subida (**`cb(null, true)`**).**
- **Si no es PNG, lo rechaza (**`cb(new Error("Solo se permiten archivos PNG"), false)`**).**

Ejemplo de comportamiento:

- Si subes `documento.png` ✅ **permitido**.
- Si subes `documento.jpg` ❌ **rechazado** con error `"Solo se permiten archivos PNG"`.

🔎5️⃣ Crear la instancia de `multer` con la configuración

📌 ¿Qué hace esto?

```
javascript
const uploadComprobantes = multer({ storage: storageComprobantes, fileFilter:
fileFilterComprobantes });
```

✅ Explicación:

- **Crea una instancia de** `multer` con las opciones definidas (`storageComprobantes`, `fileFilterComprobantes`).
- **Esta instancia se usará como middleware en las rutas de subida de archivos**.

Ejemplo de uso en `vecinos.routes.js`:

javascript
router.post("/", uploadComprobantes.single("comprobanteDomicilio"), createVecino);

Aquí, **el middleware de** `multer` **se ejecutará antes de** `createVecino` para interceptar y almacenar el archivo antes de guardarlo en la base de datos.

🔎6️⃣Exportar `uploadComprobantes` para usarlo en otros archivos

📌 ¿Qué hace esto?

javascript
export default uploadComprobantes;

✅ Explicación:

- **Exporta la configuración de subida** para ser usada en `vecinos.routes.js`.

Ejemplo de importación en otro archivo:

javascript
import uploadComprobantes from "../middlewares/uploadComprobantes.js";

Esto permite utilizar `uploadComprobantes` como middleware de subida de archivos en distintas rutas de Express.