

Análisis y Desarrollo de Arquitecturas Backend con Node.js: Enfoques de Seguridad, Escalabilidad y Rendimiento. Una guía paso a paso.

Docente: Daniel Alfonso Martínez Payán.

Tiempo aproximado: 7 horas.

Introducción

Para desarrollar aplicaciones web modernas, según Esquivel Paula, Quisaguano Collaguazo, Caluña Guaman, & Llambo Alvarez (2024), es necesario seleccionar un framework adecuado del lado del servidor es clave para garantizar un buen rendimiento, escalabilidad y facilidad en el mantenimiento del sistema. Entre las opciones más populares y consolidadas se encuentran Node.js, Django y Laravel. Cada uno de estos entornos aporta funcionalidades y recursos particulares que los hacen apropiados según el tipo de proyecto y los requerimientos del desarrollo.

En este proyecto básico, se abordará la creación y configuración de un servidor backend utilizando Node.js. El objetivo principal es desarrollar una aplicación capaz de manejar operaciones **CRUD** (Crear, Leer, Actualizar y Eliminar) y gestionar la interacción con una base de datos **MySQL**. Según Esquivel Paula, Quisaguano Collaguazo, Caluña Guaman, & Llambo Alvarez (2024), mantener un enfoque modular en la estructura del proyecto garantiza una escalabilidad eficiente, permitiendo el crecimiento y la ampliación del sistema de manera ordenada y sostenible.

Configuración Adicional

La arquitectura modular adoptada en este proyecto facilita su escalabilidad. Siguiendo el patrón de diseño **MVC** (Modelo - Vista - Controlador), se logra mantener el código limpio, organizado y de fácil mantenimiento. Tal y como mencionan Haro, Guarda, Zambrano Peñaherrera, & Ninahualpa Quiña (2019), este patrón no solo permite una expansión ordenada del proyecto, sino que también facilita la integración de nuevas funcionalidades sin comprometer la estructura existente.

Objetivo

El propósito de este proyecto es proporcionar a los estudiantes una comprensión profunda de las herramientas y metodologías fundamentales para desarrollar aplicaciones backend con **Node.js**. A través de la correcta gestión de la comunicación entre el servidor y la base de datos, y la implementación de prácticas adecuadas, los estudiantes adquirirán los conocimientos necesarios para crear sistemas de buena escalabilidad.

Contenido

Análisis y Desarrollo de Arquitecturas Backend con Node.js: Enfoques de Seguridad, Escalabilidad y Rendimiento. Una guía paso a paso	1
Introducción	1
Crear carpeta del proyecto de NodeJs	4
Inicializar la terminal en VSC	5
Inicializar proyecto de NodeJs	6
npm init	6
Dependencias principales para instalar	7
express	9
mysql2	9
dotenv	10
cors	10
multer	11
bcrypt	11
jsonwebtoken	12
nodemon	12
Package.json Configuración	13
Archivos .json ¿Qué son?	14
Creación básica en MySQL de la base de datos	14
Estructura inicial de carpetas del proyecto con Node.Js	15
Estructura de carpetas básicas	15
Estructura completa de carpetas para el proyecto backend	15
Configuración de los archivos dentro de cada carpeta	16
Carpeta config creación del archivo db.js	16
Creación del archivo .env en la raíz del proyecto	17
Creación de los controladores: archivos crud.controller.js e imágenes.controller.js dentro de la carpeta controllers	18
Configuración del archivo crud.controller.js	20
Configuración del archivo imagenes.controller.js	23
Configuración del archivo uploads.js	25
Configuración del archivo personas.routes.js	25
Configuración del archivo imagenes.routes.js	27
Configuración del archivo app.js	29
Configuración del archivo server.js	30
Inicializar el backend con el comando npm run dev	30
Archivo .gitignore: archivos, dependencias, paquetes y librerías a ignorar	31
¿Por qué es importante?	31
¿Cómo funciona?	32
Referencias bibliográficas	33

Tabla de ilustraciones

Ilustración 1. Crear carpeta del proyecto	4
Ilustración 2. Llevar la carpeta del proyecto al IDE del VSC	4
Ilustración 3. Primera forma para abrir la terminal.	5
Ilustración 4. Segunda forma para abrir la terminal.	5
Ilustración 5. Tercera forma para abrir la terminal.	5
Ilustración 6. Comando para inicializar un proyecto Node.js	6
Ilustración 7. Configuración inicial del package.json	6
Ilustración 8. Comando y dependencias a instalar	7
Ilustración 9. Librerías, paquetes y dependencias instaladas	7
Ilustración 10. Dependencia de desarrollo Nodemon comando de instalación	8
Ilustración 11. Nodemon completamente instalado	8
Ilustración 12. archivo app.js	9
Ilustración 13. archivo db.js	9
Ilustración 14. archivo .env	10
Ilustración 15. archivo db.js	10
Ilustración 16. archivo app.js	10
Ilustración 17. archivo uploads.js	11
Ilustración 18. archivo controllers.js	11
Ilustración 19. archivo controllers.js	12
Ilustración 20. archivo package.json	12
Ilustración 21. Comando para ejecutar Nodemon	12
Ilustración 22. Configuración final del package.json	13
Ilustración 23. Archivo .json	14
Ilustración 24. Script de la base de datos crud y la tabla personas en MySQL.	14
Ilustración 25. Estructura de carpetas básicas	15
Ilustración 26. Estructura de carpetas completa	15
Ilustración 27. Archivo db.js y su configuración	16
Ilustración 28. Ejemplo de uso con pool async/await	17
Ilustración 29. Archivo .env y su configuración	17
Ilustración 30. Creación de archivos controladores	18
Ilustración 31. Configuración del archivo crud.controller.js	20
Ilustración 32. Funcionamiento del archivo crud.controller.js	21
Ilustración 33. Funcionamiento del doble interrogante ??	22
Ilustración 34. Funcionamiento y comparativa de ? ó ??	22
Ilustración 35. Error de consulta SQL	22
Ilustración 36. Inyección SQL por atacante	22
Ilustración 37. Inyección SQL por atacante	22
Ilustración 38. Protección escapando caracteres peligrosos.	23
Ilustración 39. Transformación del carácter peligroso	23
Ilustración 40. Configuración archivo imágenes.controller.js	24
Ilustración 41. Configuración archivo uploads.js	25
Ilustración 42. Configuración del archivo personas.routes.js	26
Ilustración 43. Configuración del archivo imagenes.routes.js	28
Ilustración 44. Configuración del archivo app.js	29
Ilustración 45. Configuración del server.js	30
Ilustración 46. Configuración del package.json con nodemon	30
Ilustración 47. Servidor corriendo en el puerto 3000	31
Ilustración 48. Configuración del archivo .gitignore	32

Crear carpeta del proyecto de NodeJs

Crear una carpeta vacía con el nombre del proyecto y posteriormente abrirla en Visual Studio Code (VSC), dentro del IDE (Entorno de Desarrollo Integrado).

Ilustración 1. Crear carpeta del proyecto

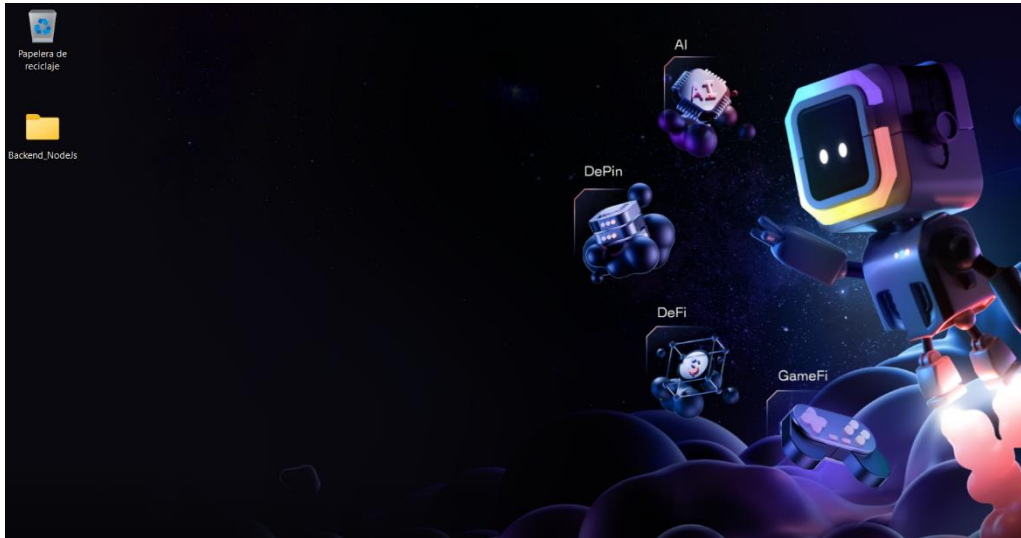
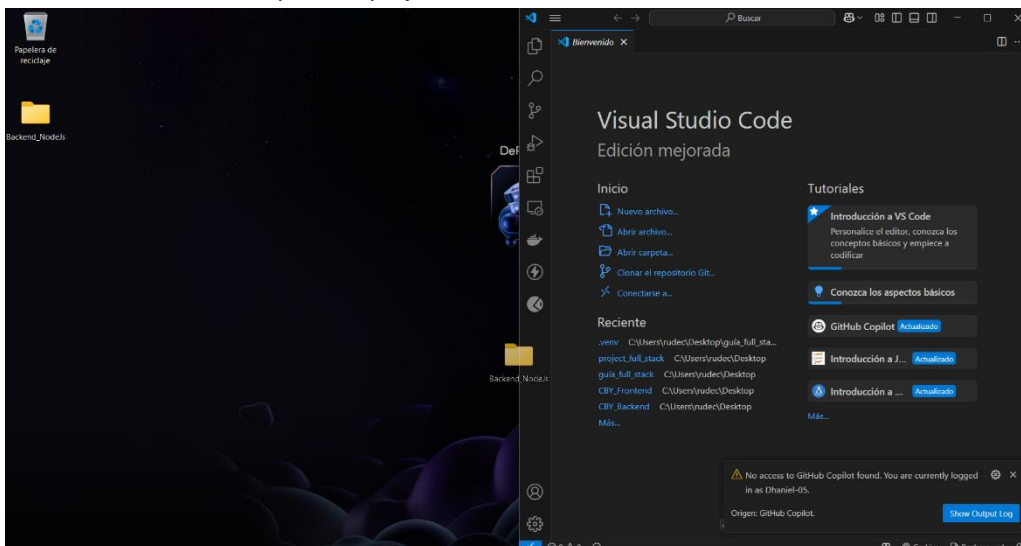


Ilustración 2. Llevar la carpeta del proyecto al IDE del VSC.



Inicializar la terminal en VSC

Ilustración 3. Primera forma para abrir la terminal.

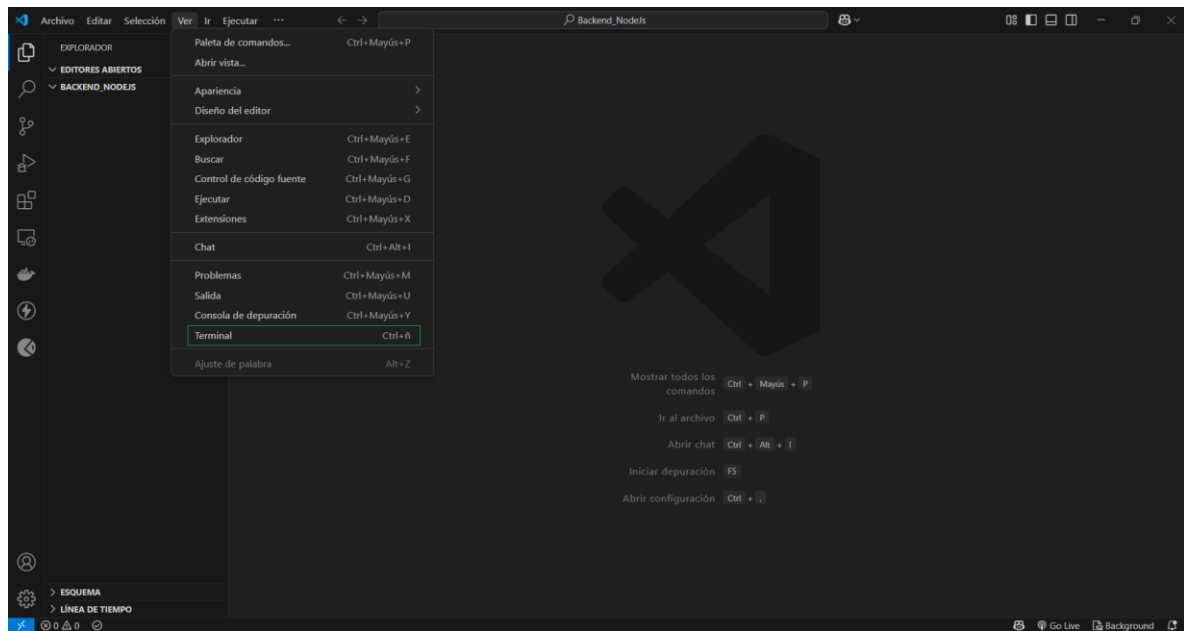


Ilustración 4. Segunda forma para abrir la terminal.

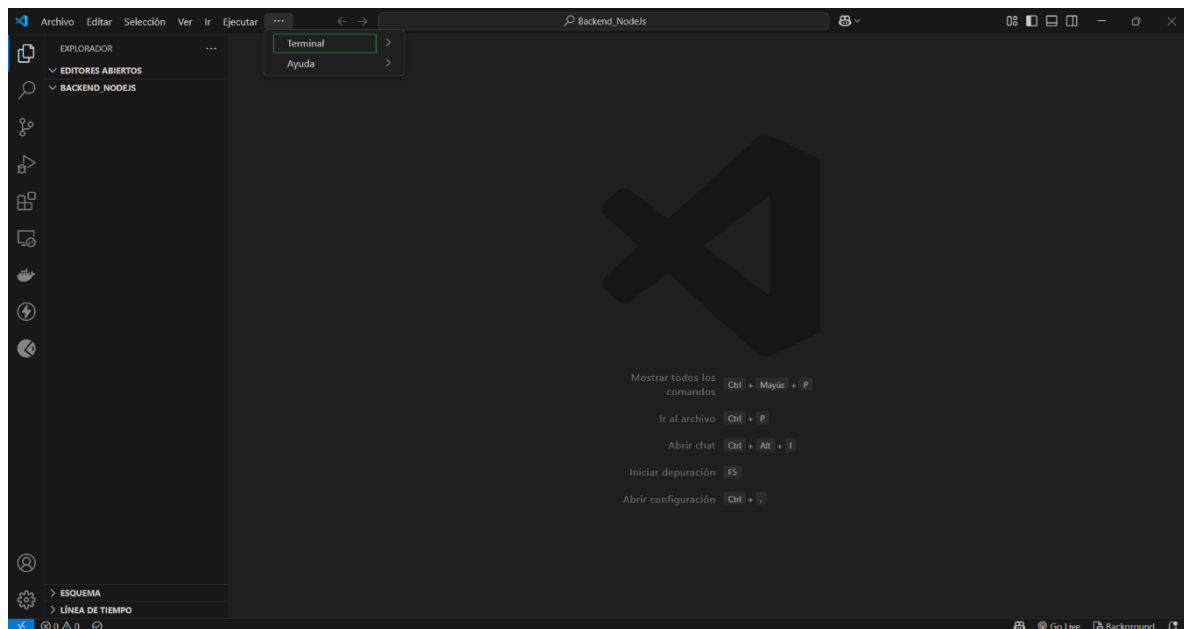
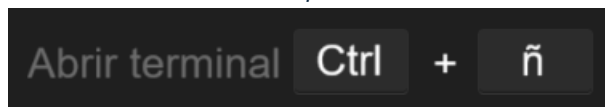


Ilustración 5. Tercera forma para abrir la terminal.



Inicializar proyecto de NodeJs

En la terminal de VSC escribir el comando **npm init** o sino **npm init -y** cualquiera de estos dos comandos sirve para crear el proyecto con Node.Js.

npm init

- ¿Qué hace?

Inicializa un nuevo proyecto de Node.Js. Este comando guía paso a paso al desarrollador a crear el archivo package.json, el cual es el corazón de cualquier proyecto Node.

Ilustración 6. Comando para inicializar un proyecto

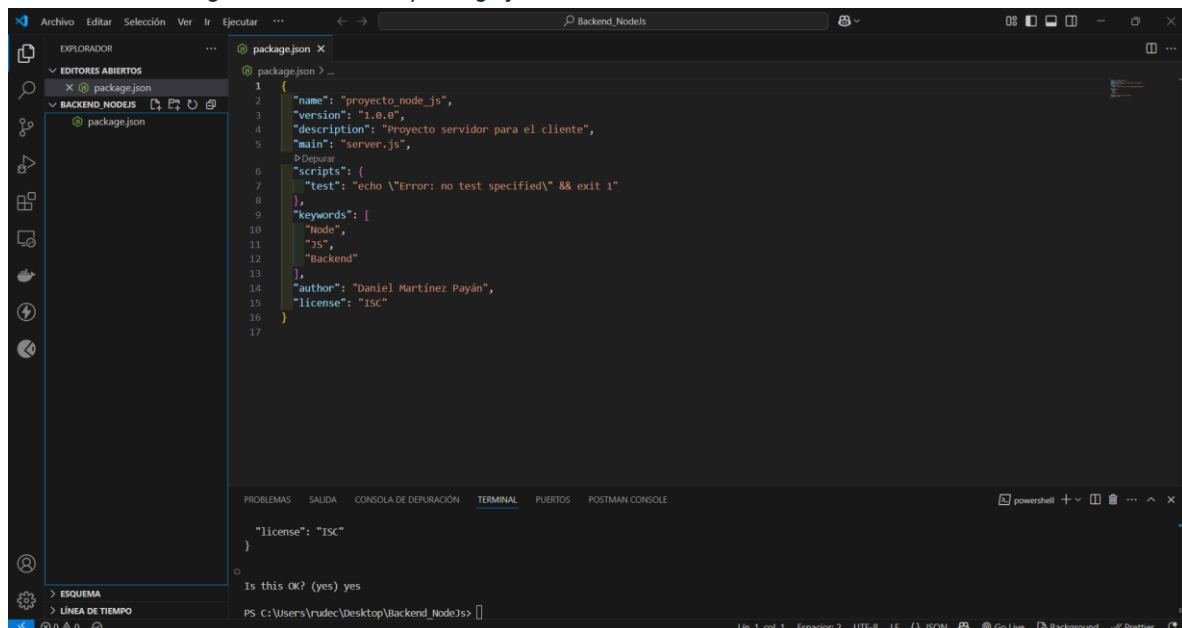
```
npm init
```

Pedirá que se complete la información del proyecto como:

- ✓ **name:** Nombre del proyecto.
- ✓ **version:** Versión inicial del proyecto (por defecto 1.0.0).
- ✓ **description:** Breve descripción.
- ✓ **entry point:** Archivo principal (por defecto index.js o server.js).
- ✓ **test command:** Comando para ejecutar pruebas.
- ✓ **git repository:** URL del repositorio si se va a utilizar.
- ✓ **keywords:** Palabras clave para describir el proyecto.
- ✓ **author:** Nombre del autor.
- ✓ **license:** Tipo de licencia (por defecto ISC).

Este proceso es interactivo. El autor (desarrollador) decide qué valores escribir o dejar en blanco, y puede confirmar todo al final.

Ilustración 7. Configuración inicial del package.json



Dependencias principales para instalar

Ilustración 8. Comando y dependencias a instalar

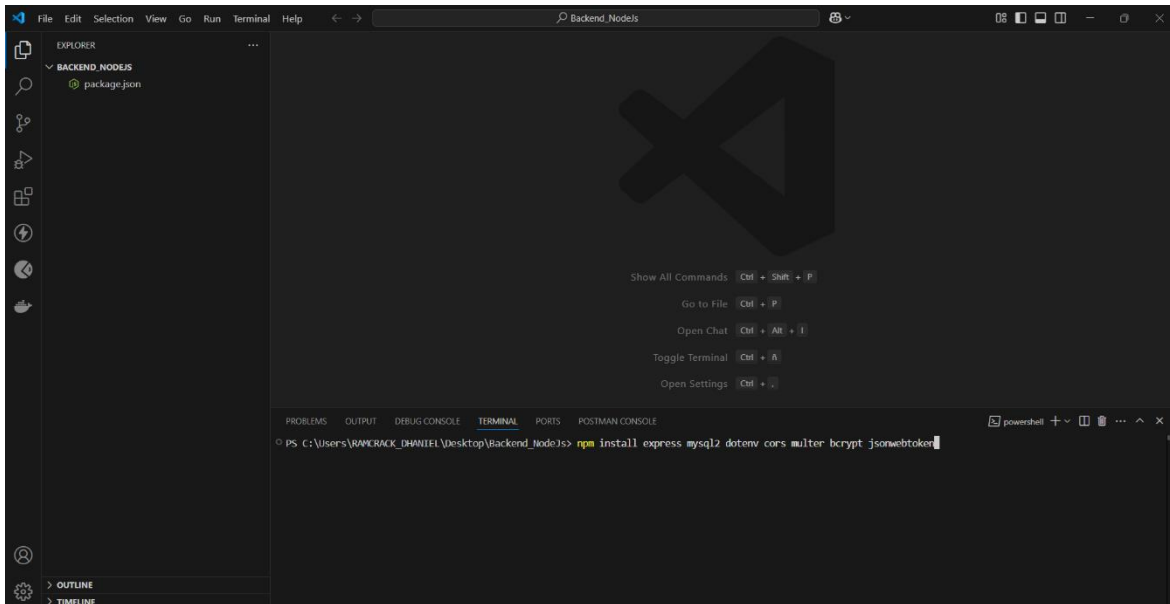


Ilustración 9. Librerías, paquetes y dependencias instaladas

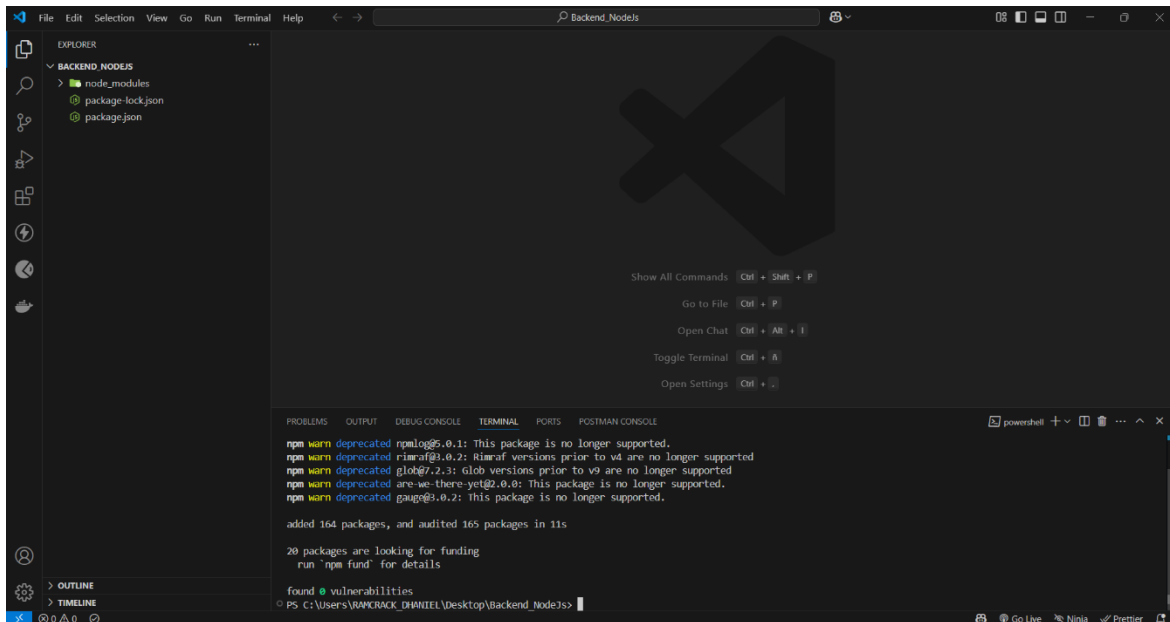


Ilustración 10. Dependencia de desarrollo Nodemon comando de instalación

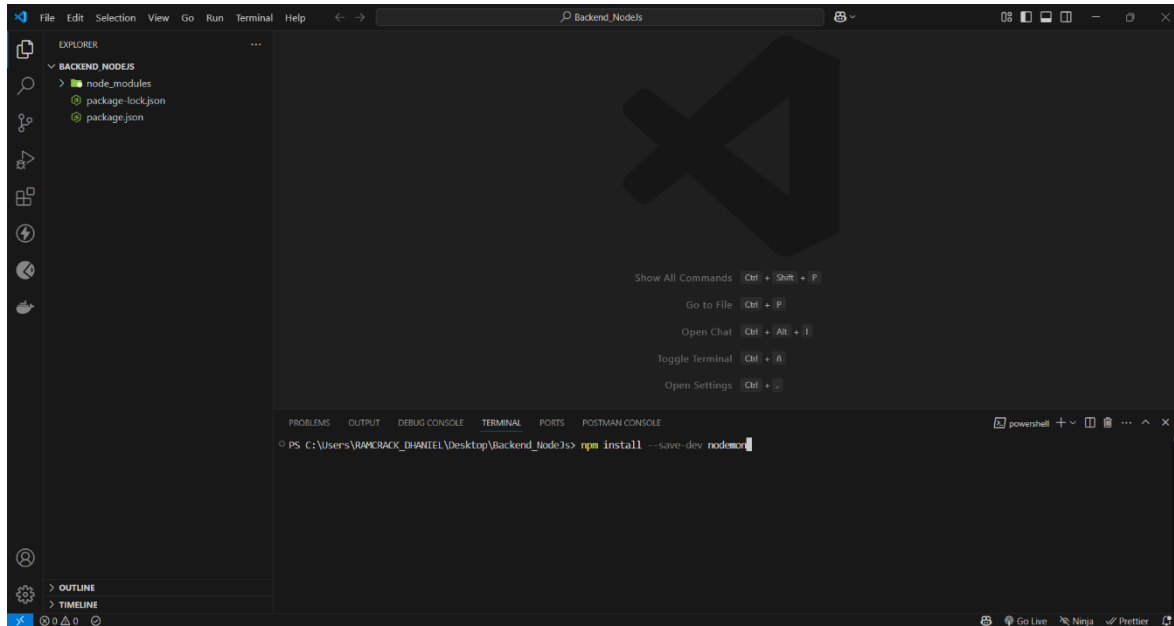
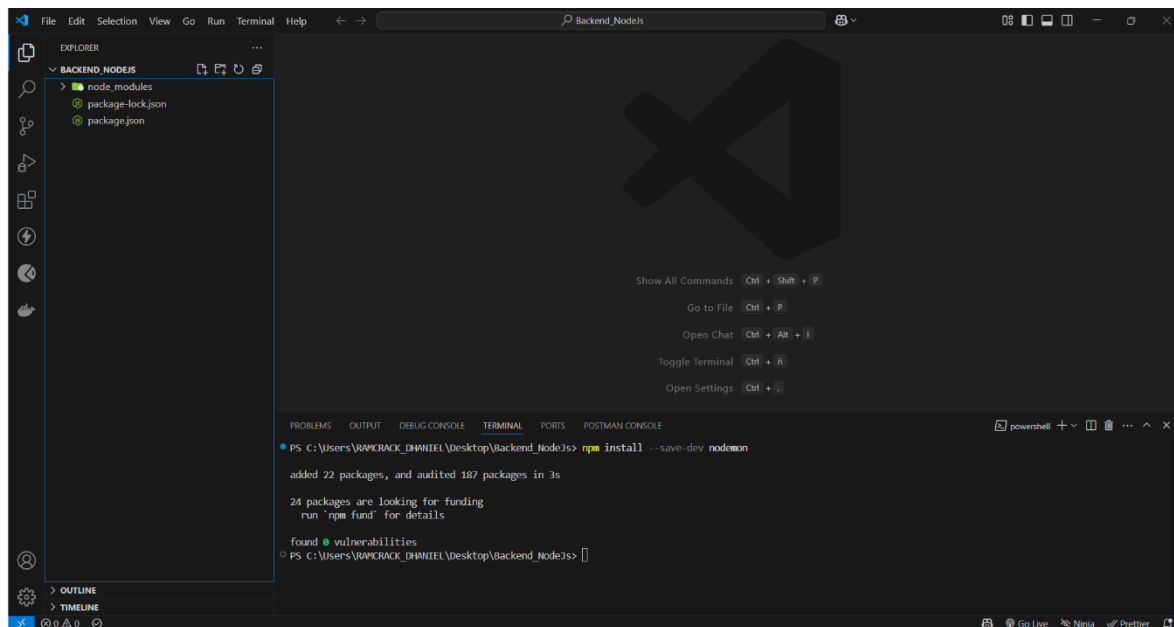


Ilustración 11. Nodemon completamente instalado.



express

- ¿Qué es?
Es un framework minimalista para construir servidores web y APIs en Node.js.
- ¿Para qué sirve?
Permite definir rutas (endpoints), middlewares y manejar solicitudes HTTP de manera sencilla.
- Ejemplo:

Ilustración 12. archivo app.js

```
const express = require('express');
const app = express();

app.get('/api/hello', (req, res) => {
  res.send('Hola mundo');
});
```

mysql2

- ¿Qué es?
Un cliente para conectarse a bases de datos MySQL desde Node.js.
- ¿Para qué sirve?
Permite ejecutar consultas SQL usando promesas o callbacks.
- Ejemplo:

Ilustración 13. archivo db.js

```
const mysql = require('mysql2');
const db = mysql.createPool({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'mi_base'
});
```

dotenv

- **¿Qué es?**
Una librería para cargar variables de entorno desde un archivo .env.
- **¿Para qué sirve?**
Mantiene credenciales y configuraciones sensibles fuera del código fuente.
- **Ejemplo:**

Ilustración 14. archivo .env

```
DB_HOST=localhost  
DB_USER=root  
DB_PASS=1234
```

Ilustración 15. archivo db.js

```
require('dotenv').config();  
const host = process.env.DB_HOST;
```

cors

- **¿Qué es?**
Un middleware para habilitar CORS (Cross-Origin Resource Sharing).
- **¿Para qué sirve?**
Permite que el frontend (por ejemplo, en React o Angular) se comuniquen con el backend aunque estén en dominios diferentes.
- **Ejemplo:**

Ilustración 16. archivo app.js

```
const cors = require('cors');  
app.use(cors());
```

multer

- ¿Qué es?
Un middleware para manejar cargas de archivos multipart/form-data.
- ¿Para qué sirve?
Permite subir imágenes, documentos, etc., desde formularios HTML.
- Ejemplo:

Ilustración 17. archivo uploads.js

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('archivo'), (req, res) => {
  res.send('Archivo subido');
});
```

bcrypt

- ¿Qué es?
Una librería para encriptar contraseñas.
- ¿Para qué sirve?
Protege las contraseñas de los usuarios guardándolas de forma segura en la base de datos.
- Ejemplo:

Ilustración 18. archivo controllers.js

```
const bcrypt = require('bcrypt');
const hashed = await bcrypt.hash('mipassword', 10);
const valido = await bcrypt.compare('mipassword', hashed);
```

jsonwebtoken

- **¿Qué es?**
Una librería para crear y verificar tokens JWT (JSON Web Tokens).
- **¿Para qué sirve?**
Para autenticación y autorización de usuarios en la aplicación.
- **Ejemplo:**

Ilustración 19. archivo controllers.js

```
const jwt = require('jsonwebtoken');  
const token = jwt.sign({ id: 1, rol: 'admin' }, 'secreto', { expiresIn: '1h' });  
  
const payload = jwt.verify(token, 'secreto');
```

nodemon

- **¿Qué es?**
Un monitor de archivos que reinicia el servidor automáticamente al detectar cambios.
- **¿Para qué sirve?**
Mejora la productividad durante el desarrollo.
- **Ejemplo:**

Ilustración 20. archivo package.json

```
"scripts": {  
  "dev": "nodemon server.js"  
}
```

Ilustración 21. Comando para ejecutar Nodemon

```
npm run dev
```

Tabla 1. Dependencias Instaladas y su utilidad

Dependencias / Librerías / Paquetes	Utilidades
express	Framework web para rutas y manejo de solicitudes
mysql2	Conexión con MySQL desde Node.js
dotenv	Para usar variables de entorno desde .env
cors	Permitir peticiones desde el frontend en otro dominio o puerto
multer	Para recibir archivos desde formularios, útil para subir imágenes
bcrypt	Encriptar contraseñas u otros datos sensibles
jsonwebtoken	Manejar autenticación con tokens (JWT)
Nodemon (desarrollador)	Reinicia el servidor automáticamente al detectar cambios

Package.json Configuración

Para ejecutar el proyecto, es necesario editar el archivo package.json. Dentro de la sección scripts, se debe especificar el nombre del archivo que se desea ejecutar en este caso sería server.js, tanto para el entorno de desarrollo como para el de producción.

Ilustración 22. Configuración final del package.json

```

1 {
2   "name": "backend_nodejs",
3   "version": "1.0.0",
4   "main": "index.js",
5   "scripts": {
6     "test": "echo \\\"Error: no test specified\\\" && exit 1",
7     "start": "node server.js",
8     "dev": "nodemon server.js"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "description": "",
14  "dependencies": {
15    "bcrypt": "^5.1.1",
16    "cors": "^2.8.5",
17    "dotenv": "^16.5.0",
18    "express": "^5.1.0",
19    "jsonwebtoken": "^9.0.2",
20    "multer": "^1.4.5-lts.2",
21    "mysql2": "^3.14.0"
22  },
23  "devDependencies": {
24    "nodemon": "^3.1.10"
25  }
26 }

```

Archivos .json ¿Qué son?

Un archivo .json (JavaScript Object Notation) es un formato de texto ligero para el intercambio de datos. Se utiliza para representar objetos y estructuras de datos de una manera legible para humanos y máquinas.

Ilustración 23. Archivo .json

```
{
  "nombre": "Juan",
  "edad": 30,
  "activo": true
}
```

Un archivo JSON es utilizado para:

- Configuraciones (como package.json en Node.js).
- Enviar y recibir datos entre frontend y backend (como respuestas de APIs).
- Almacenar información estructurada.

Creación básica en MySql de la base de datos

Para crear la base de datos, es necesario tener instalado el sistema gestor de bases de datos MySQL junto con MySQL Workbench. Además, se requiere contar con permisos de administrador y utilizar el usuario root con la contraseña root, ya que este mini proyecto ha sido configurado para funcionar bajo esas credenciales.

Ilustración 24. Script de la base de datos crud y la tabla personas en MySql.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'SCHEMAS' tree with 'crud' selected. The main editor window contains a SQL script with the following content:

```
1 -- Crear la base de datos si no existe
2 CREATE DATABASE IF NOT EXISTS crud;
3 USE crud;
4
5 -- Crear tabla personas
6 CREATE TABLE IF NOT EXISTS personas (
7   id_persona INT AUTO_INCREMENT PRIMARY KEY,
8   nombre VARCHAR(100),
9   apellido VARCHAR(100),
10  tipo_identificacion VARCHAR(50),
11  nuiip INT,
12  email VARCHAR(100),
13  clave VARCHAR(500),
14  salario DECIMAL(10,2),
15  activo BOOLEAN DEFAULT TRUE,
16  fecha_registro DATE DEFAULT (CURRENT_DATE),
17  imagen LONGLOB
18 );
19
20 -- Ver los registros actuales de la tabla personas
21 SELECT * FROM personas;
22
```

Below the script, the 'Result Grid' shows the execution results of the SQL statements. The 'Output' pane at the bottom displays the execution log:

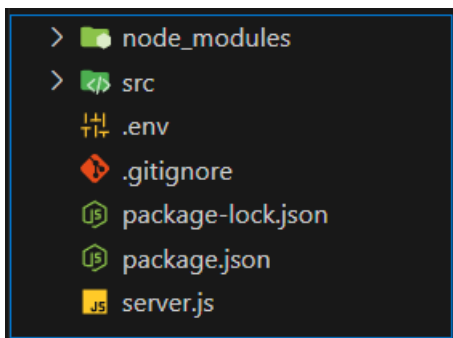
Time	Action	Message	Duration / Fetch
1 20:39:51	CREATE DATABASE IF NOT EXISTS crud	1 row(s) affected	0.125 sec
2 20:39:51	USE crud	0 row(s) affected	0.015 sec
3 20:39:51	CREATE TABLE IF NOT EXISTS personas (id_persona INT AUTO_INCREMENT PRIMARY KEY, ...)	0 row(s) affected	0.094 sec
4 20:39:51	SELECT * FROM personas LIMIT 0.1000	0 row(s) returned	0.016 sec / 0.000 sec

Estructura inicial de carpetas del proyecto con Node.Js

Estructura de carpetas básicas

Luego de instalar las dependencias y paquetes para que el proyecto con Node.Js pueda funcionar, se procede a crear la carpeta principal la cual será **src** (source “origen”, por el momento dejar esa carpeta vacía), luego crear un archivo **.env** (vacío), un archivo **.gitignore** (vacío) y un archivo **server.js** (vacío). Por lo tanto y de esta manera se tendrá la configuración inicial del proyecto de Node.js establecido.

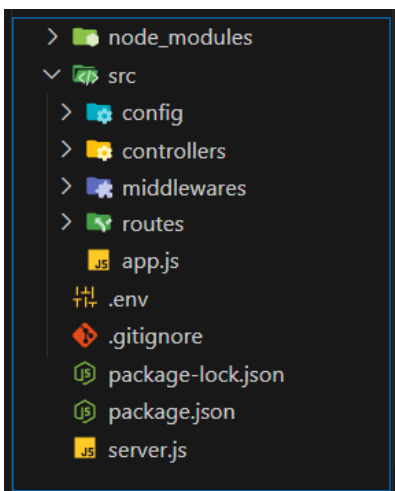
Ilustración 25. Estructura de carpetas básicas



Estructura completa de carpetas para el proyecto backend

Una vez configurado el estado inicial del proyecto, se ingresa a **src** y se establece la siguiente estructura de carpetas: **config**, **controllers**, **middlewares**, **routes**, y el archivo **app.js**. Se recomienda dejar tanto las carpetas como el archivo vacío por el momento.

Ilustración 26. Estructura de carpetas completa



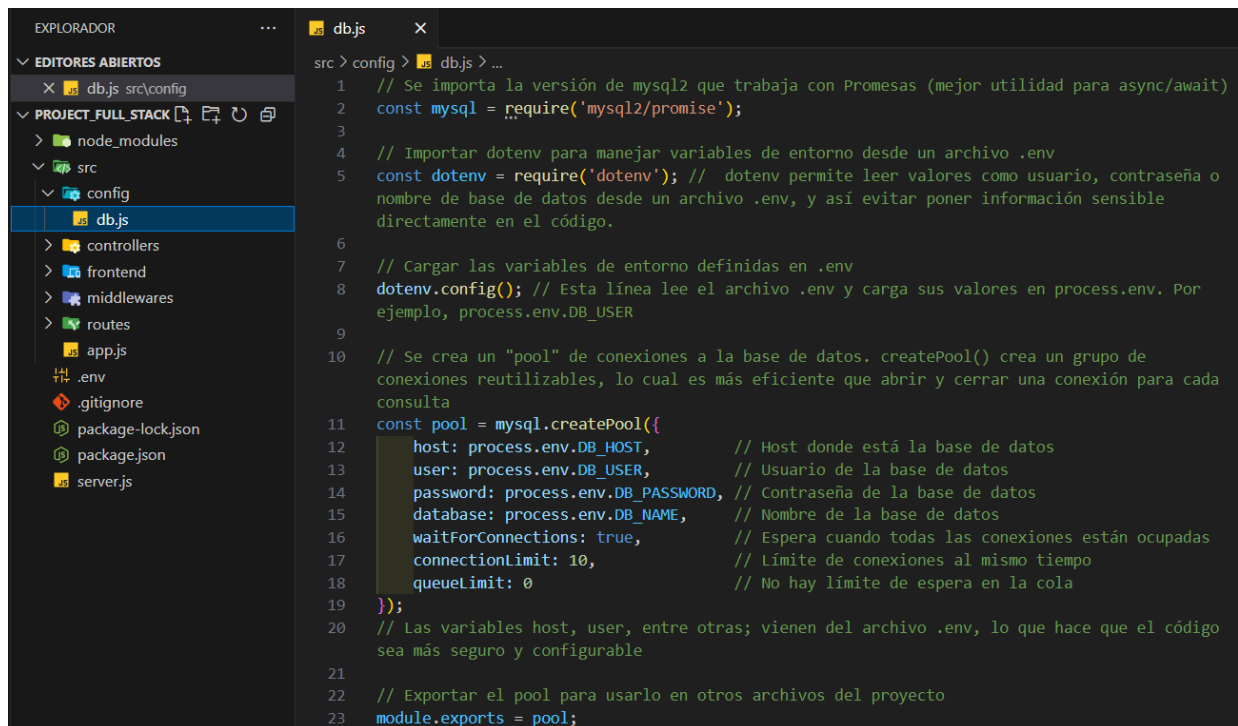
Configuración de los archivos dentro de cada carpeta

Luego de establecer la estructura de carpetas para el proyecto de Node.Js. Se procede ahora a crear y configurar cada archivo según su utilidad dentro del proyecto.

Carpeta **config** creación del archivo **db.js**

Se crea dentro de la carpeta **config** un archivo llamado **db.js** el cual tendrá el código de configuración para la conexión con la base de datos.

Ilustración 27. Archivo db.js y su configuración



```
src > config > db.js > ...
1 // Se importa la versión de mysql2 que trabaja con Promesas (mejor utilidad para async/await)
2 const mysql = require('mysql2/promise');
3
4 // Importar dotenv para manejar variables de entorno desde un archivo .env
5 const dotenv = require('dotenv'); // dotenv permite leer valores como usuario, contraseña o
  nombre de base de datos desde un archivo .env, y así evitar poner información sensible
  directamente en el código.
6
7 // Cargar las variables de entorno definidas en .env
8 dotenv.config(); // Esta línea lee el archivo .env y carga sus valores en process.env. Por
  ejemplo, process.env.DB_USER
9
10 // Se crea un "pool" de conexiones a la base de datos. createPool() crea un grupo de
  conexiones reutilizables, lo cual es más eficiente que abrir y cerrar una conexión para cada
  consulta
11 const pool = mysql.createPool({
12   host: process.env.DB_HOST,           // Host donde está la base de datos
13   user: process.env.DB_USER,           // Usuario de la base de datos
14   password: process.env.DB_PASSWORD,   // Contraseña de la base de datos
15   database: process.env.DB_NAME,       // Nombre de la base de datos
16   waitForConnections: true,             // Espera cuando todas las conexiones están ocupadas
17   connectionLimit: 10,                  // Límite de conexiones al mismo tiempo
18   queueLimit: 0,                        // No hay límite de espera en la cola
19 });
20 // Las variables host, user, entre otras; vienen del archivo .env, lo que hace que el código
  sea más seguro y configurable
21
22 // Exportar el pool para usarlo en otros archivos del proyecto
23 module.exports = pool;
```


Tabla 2. Utilidad específica mysql/promise

Característica	Callbacks (mysql2)	Promesas (mysql2/promise)
Estilo de código	Anidado y más difícil de leer	Limpio con async / await
Manero de errores	If (err) dentro de callbacks	try / catch más claro
Código asíncrono fácil	No	Sí
Escalabilidad	Más difícil	Más flexible y moderno
Recomendado en nuevos proyectos	No	Sí

Ilustración 28. Ejemplo de uso con pool async/await

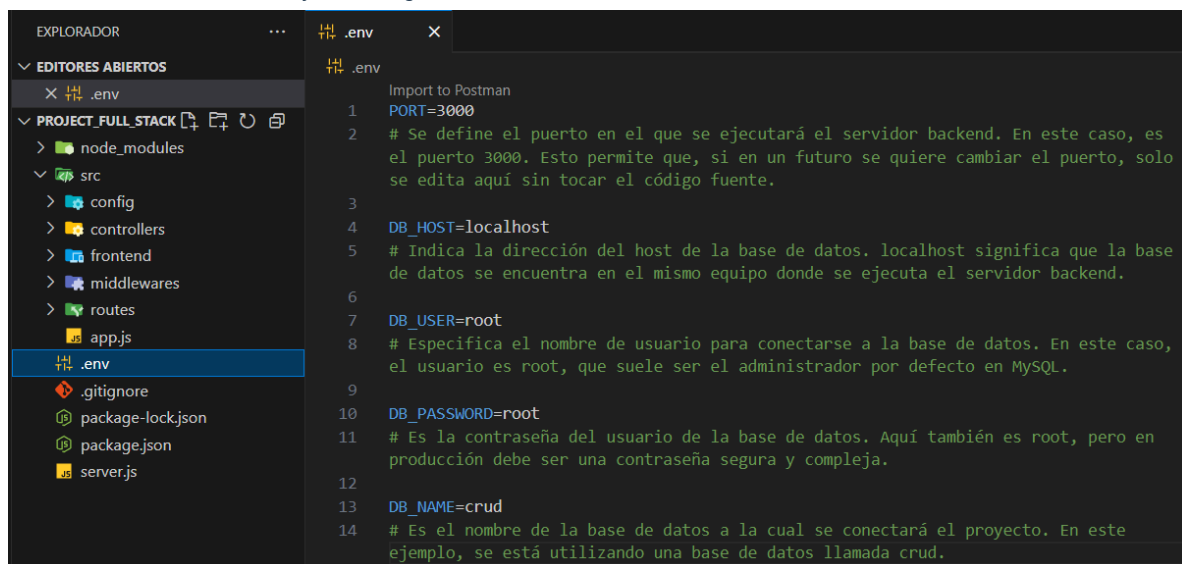
```
const pool = require('./db');

async function getUsuarios() {
  try {
    const [rows] = await pool.query('SELECT * FROM usuarios');
    console.log(rows);
  } catch (error) {
    console.error('Error al obtener usuarios:', error);
  }
}
```

Creación del archivo **.env** en la raíz del proyecto

Ahora bien, se debe configurar el archivo **.env** definiendo las variables de entorno las cuales permiten separar la configuración sensible o cambiante del código fuente, facilitando el mantenimiento, la seguridad y la portabilidad del proyecto.

Ilustración 29. Archivo **.env** y su configuración



¿Por qué utilizar .env?

- Evita que credenciales sensibles queden expuestas en el código.
- Permite modificar configuraciones fácilmente según el entorno (desarrollo, pruebas o producción).
- Mejora la portabilidad del proyecto.

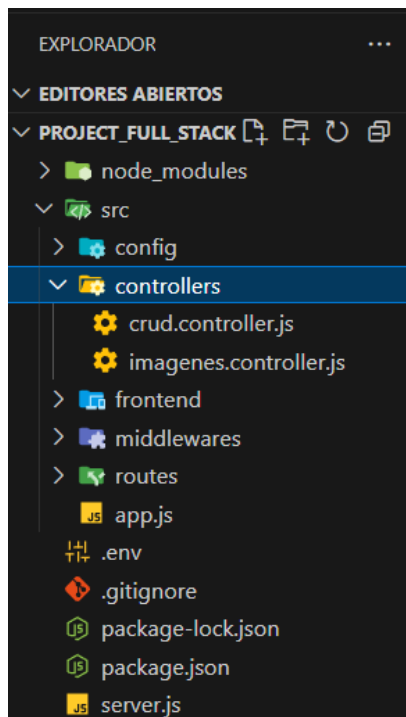
Importante:

No subir el archivo .env a repositorios **públicos** (como GitHub). Para evitarlo, es mejor agrégalo al archivo .gitignore.

Creación de los controladores: archivos crud.controller.js e imágenes.controller.js dentro de la carpeta controllers.

Una vez finalizada la configuración inicial, se procede a crear los controladores que manejarán la lógica del CRUD y la comunicación entre el servidor y la base de datos. Para ello, se crean los archivos **crud.controller.js** e **imagenes.controller.js** dentro de la carpeta **controllers**. Estos controladores contendrán el código necesario para ejecutar las consultas SQL en el gestor de base de datos MySQL, gestionando así las operaciones como crear, leer, actualizar y eliminar datos.

Ilustración 30. Creación de archivos controladores



Es importante tener en cuenta la razón por la que se suele nombrar un archivo como **crud.controller.js** en lugar de simplemente **crud.js** tiene que ver principalmente con convenciones de organización y claridad del código en proyectos Node.js. Es decir:

¿Por qué **crud.controller.js** y no solo **crud.js**?

1. Claridad semántica:

El nombre **crud.controller.js** indica claramente qué rol cumple ese archivo dentro de la aplicación.

Al leer ese nombre, cualquier desarrollador puede entender de inmediato que se trata de un controlador, es decir, la parte del backend encargada de manejar la lógica entre las rutas y la base de datos.

2. Mejora la organización del proyecto:

En una aplicación más grande, se podrían tener muchos archivos con funciones similares pero en diferentes capas (como **crud.model.js**, **crud.routes.js**, **crud.service.js**), y el sufijo **.controller.js** evita confusiones al identificar el propósito específico de cada archivo.

3. Sigue la convención de arquitectura MVC (Modelo - Vista - Controlador):

En MVC, el Controlador es quien gestiona las solicitudes, llama a los modelos, y devuelve respuestas.

Nombrarlo como **.controller.js** mantiene la convención clara y útil para desarrolladores que están familiarizados con el patrón.

¿Se puede usar solo **crud.js**?

Técnicamente se puede utilizar cualquier nombre, puede ser **crud.js**, y funcionará igual. Sin embargo, si todos los archivos se llaman genéricamente (**app.js**, **crud.js**, **db.js**, entre otros.), el proyecto puede volverse difícil de entender conforme crece. En cambio, el uso de nombres como **crud.controller.js**, **db.config.js**, **imagenes.routes.js** ayuda a mantener el código escalable y más profesional.

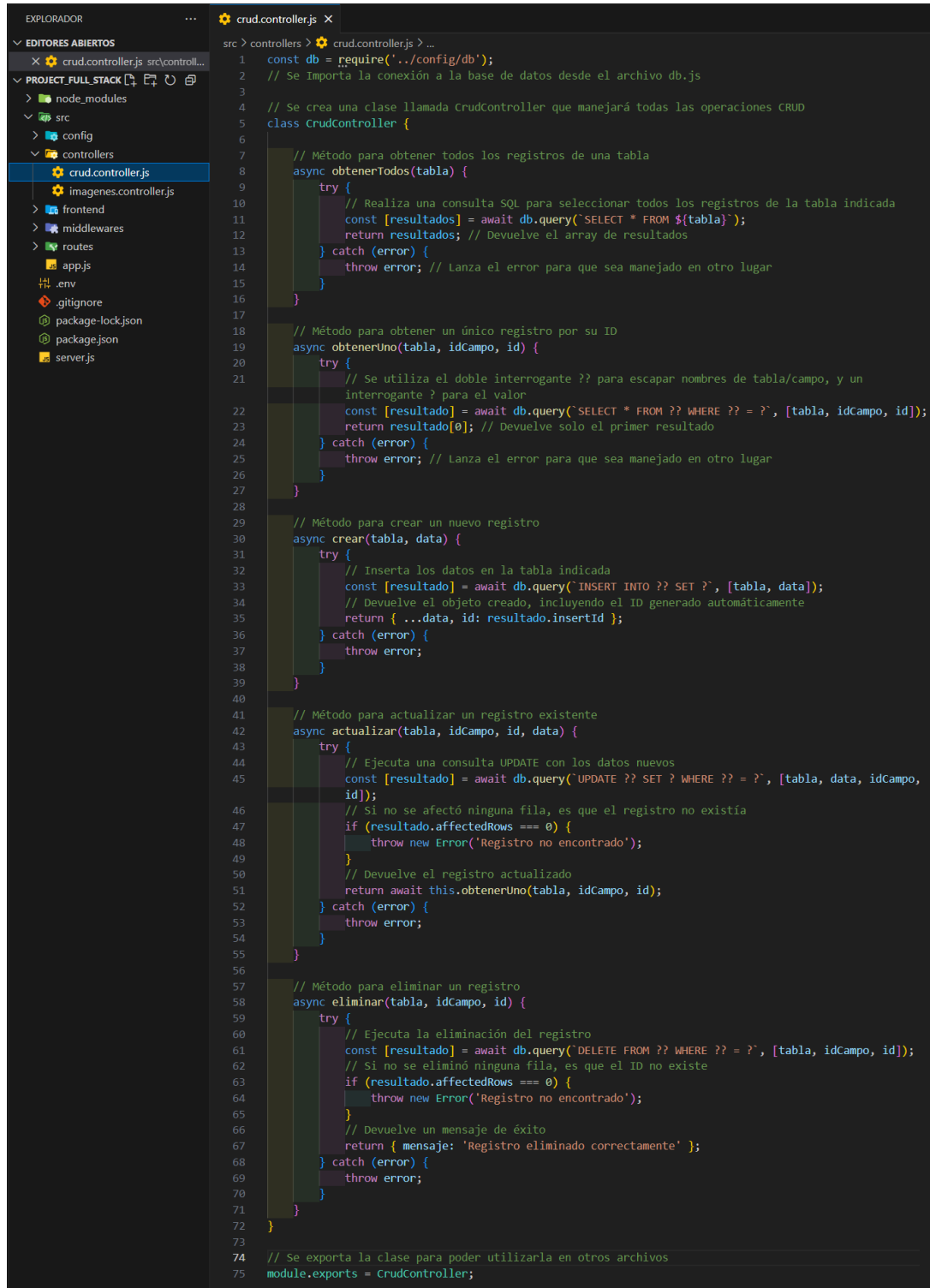
Tabla 3. Comparativa de Nombres de Archivos en Node.js

Nombre de archivo	Propósito	Buenas prácticas	Ejemplo en proyecto
crud.js	Genérico, no indica claramente su función	Poco claro	controllers/crud.js ¿modelo, ruta o controlador?
crud.controller.js	Indica que es un controlador que gestiona la lógica del CRUD	Muy claro	controllers/crud.controller.js
crud.model.js	Representa el modelo de datos, interacción directa con la base de datos	Muy claro	models/crud.model.js
crud.routes.js	Define las rutas del CRUD, recibe las peticiones HTTP	Muy claro	routes/crud.routes.js
db.config.js	Archivo de configuración de la base de datos	Muy claro	config/db.config.js

Configuración del archivo crud.controller.js

Teniendo en claro lo anterior, se procede a configurar el contenido del archivo **crud.controller.js**.

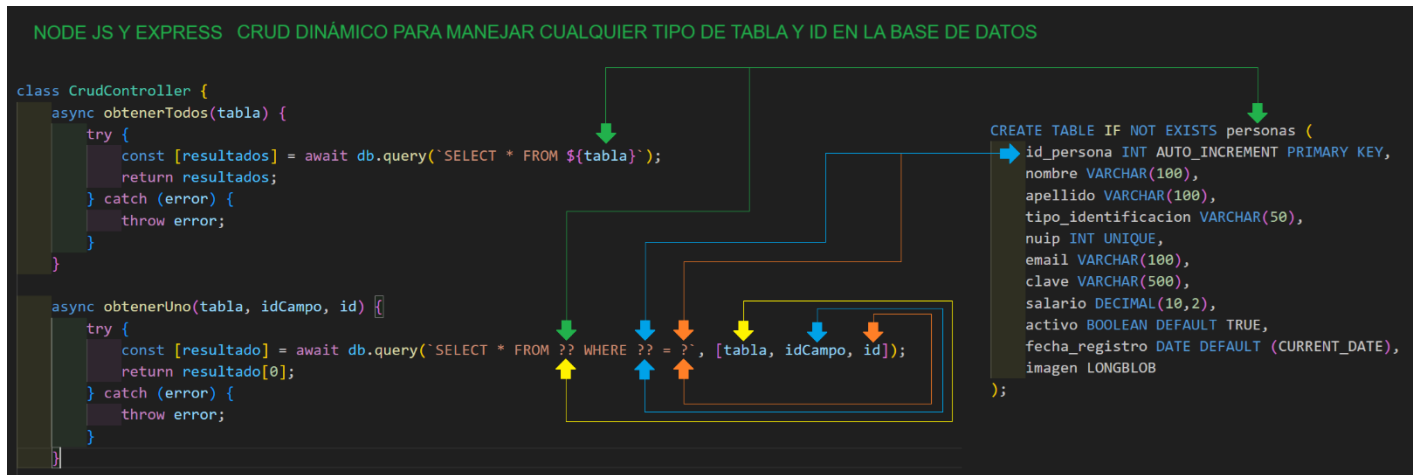
Ilustración 31. Configuración del archivo crud.controller.js



```
src > controllers > crud.controller.js > ...
1  const db = require('../config/db');
2  // Se importa la conexión a la base de datos desde el archivo db.js
3
4  // Se crea una clase llamada CrudController que manejará todas las operaciones CRUD
5  class CrudController {
6
7      // Método para obtener todos los registros de una tabla
8      async obtenerTodos(tabla) {
9          try {
10             // Realiza una consulta SQL para seleccionar todos los registros de la tabla indicada
11             const [resultados] = await db.query('SELECT * FROM ${tabla}');
12             return resultados; // Devuelve el array de resultados
13         } catch (error) {
14             throw error; // Lanza el error para que sea manejado en otro lugar
15         }
16     }
17
18     // Método para obtener un único registro por su ID
19     async obtenerUno(tabla, idCampo, id) {
20         try {
21             // Se utiliza el doble interrogante ?? para escapar nombres de tabla/campo, y un
22             // interrogante ? para el valor
23             const [resultado] = await db.query('SELECT * FROM ?? WHERE ?? = ?', [tabla, idCampo, id]);
24             return resultado[0]; // Devuelve solo el primer resultado
25         } catch (error) {
26             throw error; // Lanza el error para que sea manejado en otro lugar
27         }
28     }
29
30     // Método para crear un nuevo registro
31     async crear(tabla, data) {
32         try {
33             // Inserta los datos en la tabla indicada
34             const [resultado] = await db.query('INSERT INTO ?? SET ?', [tabla, data]);
35             // Devuelve el objeto creado, incluyendo el ID generado automáticamente
36             return { ...data, id: resultado.insertId };
37         } catch (error) {
38             throw error;
39         }
40     }
41
42     // Método para actualizar un registro existente
43     async actualizar(tabla, idCampo, id, data) {
44         try {
45             // Ejecuta una consulta UPDATE con los datos nuevos
46             const [resultado] = await db.query('UPDATE ?? SET ? WHERE ?? = ?', [tabla, data, idCampo, id]);
47             // Si no se afectó ninguna fila, es que el registro no existía
48             if (resultado.affectedRows === 0) {
49                 throw new Error('Registro no encontrado');
50             }
51             // Devuelve el registro actualizado
52             return await this.obtenerUno(tabla, idCampo, id);
53         } catch (error) {
54             throw error;
55         }
56     }
57
58     // Método para eliminar un registro
59     async eliminar(tabla, idCampo, id) {
60         try {
61             // Ejecuta la eliminación del registro
62             const [resultado] = await db.query('DELETE FROM ?? WHERE ?? = ?', [tabla, idCampo, id]);
63             // Si no se eliminó ninguna fila, es que el ID no existe
64             if (resultado.affectedRows === 0) {
65                 throw new Error('Registro no encontrado');
66             }
67             // Devuelve un mensaje de éxito
68             return { mensaje: 'Registro eliminado correctamente' };
69         } catch (error) {
70             throw error;
71         }
72     }
73
74     // Se exporta la clase para poder utilizarla en otros archivos
75     module.exports = CrudController;
```

¿Cómo funciona?

Ilustración 32. Funcionamiento del archivo crud.controller.js



Escapar: Hacer seguro un dato antes de usarlo en SQL

Tabla 4. Escapar y su significado en programación

Término Técnico	Significado / Sinónimos
Escapar	Proteger el valor contra inyecciones SQL Convertir en seguro para bases de datos Sanitizar el dato Neutralizar caracteres peligrosos Codificar el valor para que no se interprete como código Blindar el valor antes de usarlo en una consulta Filtrar caracteres maliciosos

Tabla 5. Diferencias entre ? y ??

Símbolo	Representa	Realiza
?	Valor o dato	Escapa el valor para evitar inyección SQL (Escapa valores peligrosos como strings, números, entre otros.)
??	Identificador (Nombre de tabla o nombre de columna)	Escapa identificadores (Nombres de campos o tablas) Escapa nombres de campos o tablas (como id, usuarios, entre otros.)

¿Por qué se utiliza el ?? en tabla e idCampo y no en id?

Ilustración 33. Funcionamiento del doble interrogante ??

```
await db.query('SELECT * FROM ?? WHERE ?? = ?', [tabla, idCampo, id]);
```

- ?? → tabla (porque es el nombre de la tabla, por ejemplo "personas")
- ?? → idCampo (porque es el nombre del campo, por ejemplo "id_persona")
- ? → id (porque es el valor del campo, como 3, 45, o cualquier número o string que identifica).

¿Qué sucede si se utiliza ? en vez de esto ???

Ilustración 34. Funcionamiento y comparativa de ? ó ??

```
await db.query('SELECT * FROM ? WHERE ? = ?', [tabla, idCampo, id]);
```

Va a lanzar un error o generar una consulta inválida como esta:

Ilustración 35. Error de consulta SQL

```
SELECT * FROM 'personas' WHERE 'id_persona' = 5;
```

Esta consulta sería incorrecta puesto que, los nombres de tablas o campos no van entre comillas simples ('), esto lo realiza el (?), que escapa valores, no identificadores.

¿Qué es una inyección SQL?

Es cuando un atacante intenta "colarse" en una base de datos ingresando código SQL en un campo de entrada. Por ejemplo:

Ilustración 36. Inyección SQL por atacante.

```
const usuario = "'; DROP TABLE usuarios; --";  
const query = `SELECT * FROM usuarios WHERE nombre = '${usuario}'`;
```

Esto en SQL podría ejecutar:

Ilustración 37. Inyección SQL por atacante.

```
SELECT * FROM usuarios WHERE nombre = ''; DROP TABLE usuarios; --'
```

¡Adiós a la tabla usuarios!

¿Qué función tiene el `?`?

Cuando se utiliza el `?` en lugar de insertar el valor directo, la librería escapa caracteres peligrosos automáticamente, así:

Ilustración 38. Protección escapando caracteres peligrosos.

```
db.query('SELECT * FROM usuarios WHERE nombre = ?', [usuario]);
```

Internamente lo transforma en algo como:

Ilustración 39. Transformación del carácter peligroso.

```
SELECT * FROM usuarios WHERE nombre = '\'; DROP TABLE usuarios; --'
```

Ahora el valor se trata como un string literal seguro, no como código.

Configuración del archivo `imagenes.controller.js`

Luego de configurar el CRUD general, se procede a configurar el contenido del archivo **imagenes.controller.js**. Este, será un controlador reutilizable que permitirá gestionar imágenes asociadas a registros de cualquier tabla en una base de datos. Proporciona funciones para:

- Subir una imagen
- Obtener una imagen
- Eliminar una imagen
- Insertar (o actualizar) una imagen si no existe
- Procesar automáticamente subida u obtención según si hay imagen base64 o no

Todo esto lo hace de forma dinámica y genérica, recibiendo el nombre de la tabla y el campo ID como parámetros, lo que evita crear un CRUD de imágenes por cada tabla.

Escalabilidad del controlador

Este controlador es altamente escalable debido a que es genérico por lo que no está atado a una tabla específica. Se puede utilizar para cualquier tabla que tenga una columna imagen, simplemente pasando el nombre de la tabla y el campo ID correspondiente. Además, permite mantener una sola lógica centralizada, lo que simplifica el mantenimiento y evita duplicación de código. Si dentro del proyecto mucho más adelante se requiere agregar una tabla productos, usuarios, artistas, entre otras; solo hay que asegurarse de que tengan un campo tipo imagen y este controlador ya puede manejarlo sin modificar el código existente.

Ilustración 40. Configuración archivo imágenes.controller.js

```

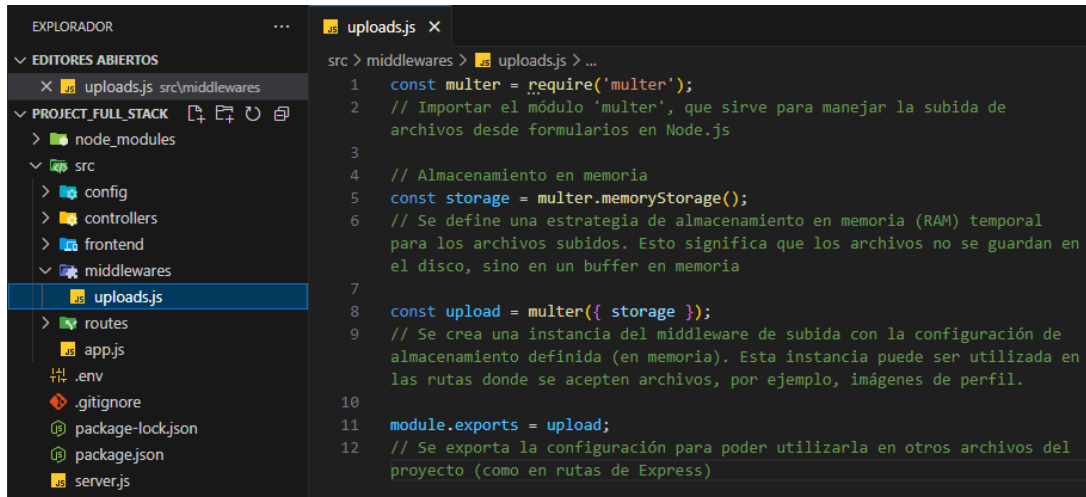
1 // Importar la conexión a la base de datos desde el archivo de configuración
2 const db = require("../config/db");
3
4 // Crear la clase ImagenesController que manejará las operaciones relacionadas con imágenes
5 class ImagenesController {
6
7   // Método para subir o actualizar una imagen codificada en base64 a un registro específico
8   async subirImagen(tabla, campoid, id, imagenBase64) {
9     try {
10       // Consultar si el registro con el ID existe
11       const [registro] = await db.query("SELECT * FROM ?? WHERE ?? = ?", [tabla, campoid, id]);
12
13       // Si no existe, retornar un error
14       if (registro.length === 0) {
15         return { error: "No se encontró el registro con el ID proporcionado." };
16       }
17
18       // Convertir la imagen de base64 a un buffer (formato binario)
19       const bufferImagen = Buffer.from(imagenBase64, "base64");
20
21       // Crear la consulta para actualizar el campo 'imagen' del registro
22       const query = "UPDATE ?? SET imagen = ? WHERE ?? = ?";
23       const [result] = await db.query(query, [tabla, bufferImagen, campoid, id]);
24
25       // Validar si la actualización fue exitosa
26       if (result.affectedRows > 0) {
27         return { message: "Imagen actualizada correctamente." };
28       } else {
29         return { error: "Error al actualizar la imagen." };
30       }
31     } catch (error) {
32       console.error("Error al subir la imagen:", error);
33       throw error;
34     }
35   }
36
37   // Método para obtener una imagen desde un registro y devolverla en formato base64
38   async obtenerImagen(tabla, campoid, id) {
39     try {
40       // Consultar el campo 'imagen' del registro
41       const [rows] = await db.query("SELECT imagen FROM ?? WHERE ?? = ?", [tabla, campoid, id]);
42
43       // Validar si se encontró el registro
44       if (rows.length === 0) {
45         return { error: "Registro no encontrado" };
46       }
47
48       // Verificar si el campo imagen está vacío
49       if (rows[0].imagen) {
50         return { error: "No hay imagen asociada a este registro" };
51       }
52
53       // Convertir la imagen de binario a base64
54       const imagenBase64 = rows[0].imagen.toString("base64");
55
56       // Retornar la imagen codificada
57       return { imagen: imagenBase64 };
58     } catch (error) {
59       console.error("Error al obtener la imagen:", error);
60       throw error;
61     }
62   }
63
64   // Método para eliminar una imagen (establece el campo 'imagen' como NULL)
65   async eliminarImagen(tabla, campoid, id) {
66     try {
67       // Verificar que el registro existe
68       const [registro] = await db.query("SELECT * FROM ?? WHERE ?? = ?", [tabla, campoid, id]);
69
70       // Si no existe, retornar un error
71       if (registro.length === 0) {
72         return { error: "No se encontró el registro con el ID proporcionado." };
73       }
74
75       // Establecer el campo 'imagen' como NULL
76       const query = "UPDATE ?? SET imagen = NULL WHERE ?? = ?";
77       const [result] = await db.query(query, [tabla, campoid, id]);
78
79       // Validar si se eliminó correctamente
80       if (result.affectedRows > 0) {
81         return { message: "Imagen eliminada correctamente." };
82       } else {
83         return { error: "Error al eliminar la imagen." };
84       }
85     } catch (error) {
86       console.error("Error al eliminar la imagen:", error);
87       throw error;
88     }
89   }
90
91   // Método que inserta una imagen si no existe o actualiza si ya hay una
92   async insertarImagen(tabla, campoid, id, imagenBase64) {
93     try {
94       // Verificar que el registro existe
95       const [registro] = await db.query("SELECT * FROM ?? WHERE ?? = ?", [tabla, campoid, id]);
96
97       // Si no existe, retornar un error
98       if (registro.length === 0) {
99         return { error: "No se encontró el registro con el ID proporcionado." };
100       }
101
102       // Convertir la imagen a formato binario
103       const bufferImagen = Buffer.from(imagenBase64, "base64");
104
105       // Consultar si ya hay una imagen existente
106       const [imagenExistente] = await db.query("SELECT imagen FROM ?? WHERE ?? = ?", [tabla, campoid, id]);
107
108       // Si ya hay una imagen, actualizar
109       if (imagenExistente[0].imagen) {
110         const query = "UPDATE ?? SET imagen = ? WHERE ?? = ?";
111         const [result] = await db.query(query, [tabla, bufferImagen, campoid, id]);
112
113         if (result.affectedRows > 0) {
114           return { message: "Imagen actualizada correctamente." };
115         } else {
116           return { error: "Error al actualizar la imagen." };
117         }
118       } else {
119         // Si no hay imagen, insertar una nueva
120         const query = "UPDATE ?? SET imagen = ? WHERE ?? = ?";
121         const [result] = await db.query(query, [tabla, bufferImagen, campoid, id]);
122
123         if (result.affectedRows > 0) {
124           return { message: "Imagen insertada correctamente." };
125         } else {
126           return { error: "Error al insertar la imagen." };
127         }
128       }
129     } catch (error) {
130       console.error("Error al insertar la imagen:", error);
131       throw error;
132     }
133   }
134
135   // Método general que decide si subir una imagen o solo obtenerla
136   async procesarImagen(tabla, campoid, id, imagenBase64 = null) {
137     // Si se pasa una imagen, la sube
138     if (imagenBase64) {
139       return await this.subirImagen(tabla, campoid, id, imagenBase64);
140     } else {
141       // Si no, intenta recuperarla
142       return await this.obtenerImagen(tabla, campoid, id);
143     }
144   }
145
146   // Exportar una instancia del controlador para su uso en rutas u otros módulos
147   module.exports = new ImagenesController();
148 }

```


Configuración del archivo uploads.js

El archivo upload.js en la carpeta middlewares sirve para gestionar la subida de archivos (como imágenes) desde formularios del frontend, utilizando multer con almacenamiento en memoria, para luego procesarlos o guardarlos en la base de datos sin escribirlos en disco.

Ilustración 41. Configuración archivo uploads.js



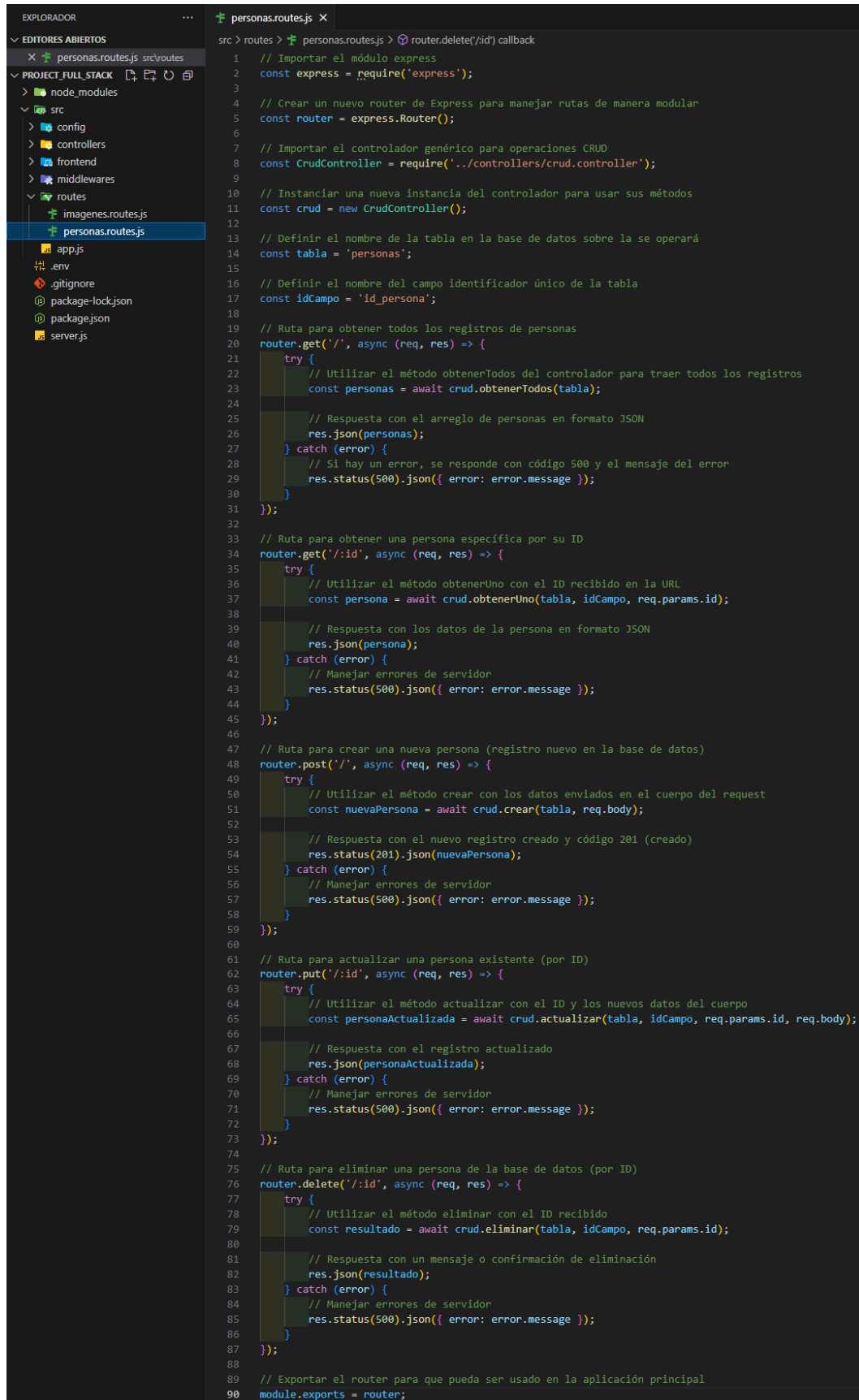
```
EXPLORADOR
...
EDITORES ABIERTOS
  uploads.js src/middlewares
PROJECT_FULL_STACK
  node_modules
  src
    config
    controllers
    frontend
    middlewares
      uploads.js
    routes
    app.js
    .env
    .gitignore
    package-lock.json
    package.json
    server.js

src > middlewares > uploads.js > ...
1  const multer = require('multer');
2  // Importar el módulo 'multer', que sirve para manejar la subida de
   archivos desde formularios en Node.js
3
4  // Almacenamiento en memoria
5  const storage = multer.memoryStorage();
6  // Se define una estrategia de almacenamiento en memoria (RAM) temporal
   para los archivos subidos. Esto significa que los archivos no se guardan en
   el disco, sino en un buffer en memoria
7
8  const upload = multer({ storage });
9  // Se crea una instancia del middleware de subida con la configuración de
   almacenamiento definida (en memoria). Esta instancia puede ser utilizada en
   las rutas donde se acepten archivos, por ejemplo, imágenes de perfil.
10
11 module.exports = upload;
12 // Se exporta la configuración para poder utilizarla en otros archivos del
   proyecto (como en rutas de Express)
```

Configuración del archivo personas.routes.js

Este archivo de rutas del lado del servidor organiza y gestiona las peticiones relacionadas con la entidad personas, permitiendo realizar operaciones CRUD (Crear, Leer, Actualizar y Eliminar) mediante el uso del framework Express. Actúa como intermediario entre las solicitudes HTTP recibidas y la lógica de base de datos implementada en el controlador **crud.controller**. Por ejemplo, cuando el cliente (**frontend**) envía una solicitud **POST** a la ruta /personas, el servidor la recibe en el archivo **routes**, y este llama al método **crear()** del controlador, enviando como argumentos el nombre de la tabla (personas) y los datos contenidos en **req.body** tal y como mencionan Palmera Quintero, Ríos Baron, León, & Chinchilla Torres (2024). Esta estructura modular facilita el mantenimiento del código y su escalabilidad, ya que permite aplicar la misma lógica a otras entidades simplemente cambiando los nombres de la tabla y del campo clave, manteniendo el código limpio, reutilizable y eficiente.

Ilustración 42. Configuración del archivo `personas.routes.js`



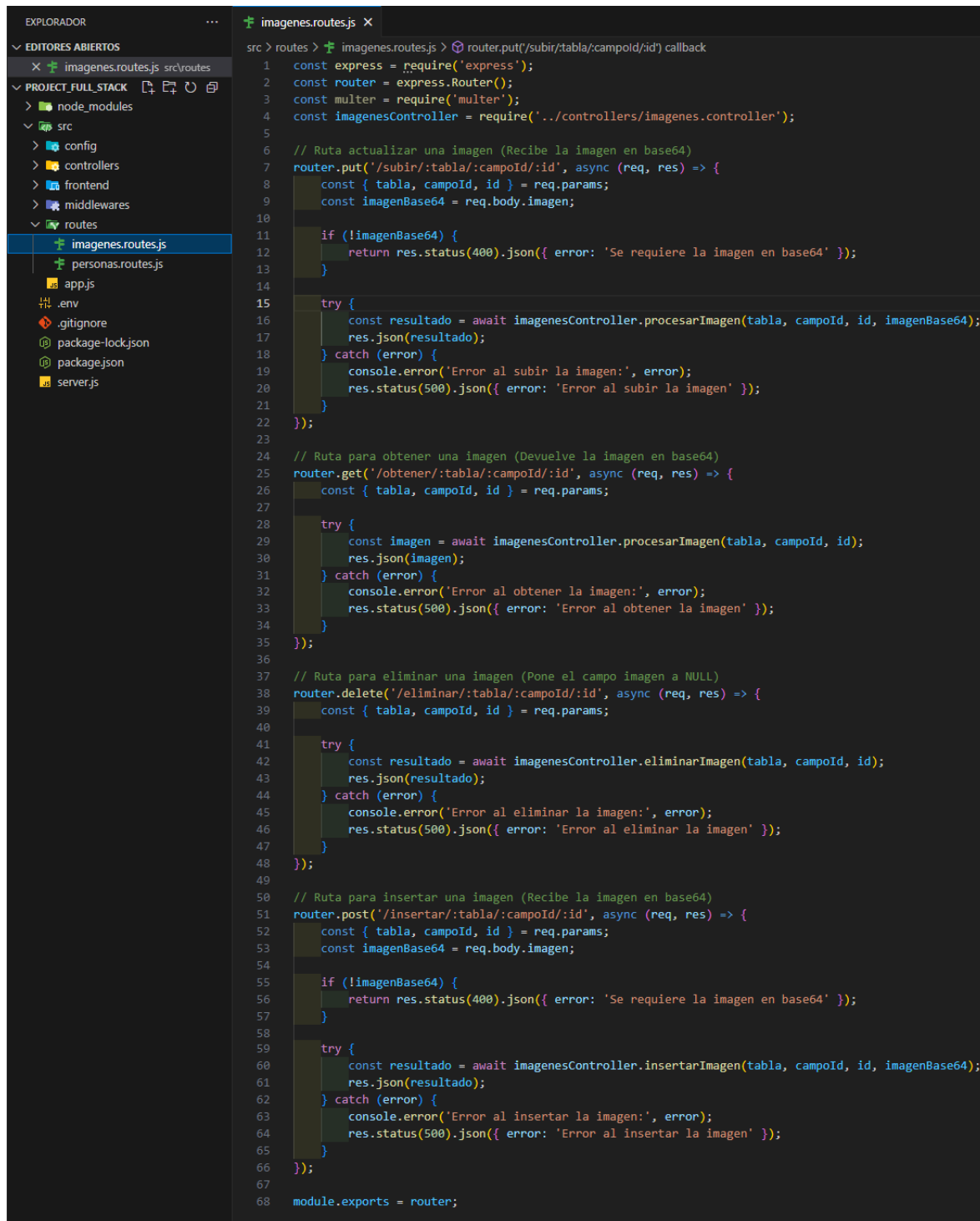
```
EXPLORADOR
...
EDITORES ABIERTOS
  X personas.routes.js src/routes
PROJECT_FULL_STACK
  > node_modules
  > src
    > config
    > controllers
    > frontend
    > middlewares
    > routes
      > imagenes.routes.js
      X personas.routes.js
  > app.js
  > .env
  > .gitignore
  > package-lock.json
  > package.json
  > server.js

src > routes > personas.routes.js > router.delete('/:id') callback
1 // Importar el módulo express
2 const express = require('express');
3
4 // Crear un nuevo router de Express para manejar rutas de manera modular
5 const router = express.Router();
6
7 // Importar el controlador genérico para operaciones CRUD
8 const CrudController = require('../controllers/crud.controller');
9
10 // Instanciar una nueva instancia del controlador para usar sus métodos
11 const crud = new CrudController();
12
13 // Definir el nombre de la tabla en la base de datos sobre la se operará
14 const tabla = 'personas';
15
16 // Definir el nombre del campo identificador único de la tabla
17 const idCampo = 'id_persona';
18
19 // Ruta para obtener todos los registros de personas
20 router.get('/', async (req, res) => {
21   try {
22     // Utilizar el método obtenerTodos del controlador para traer todos los registros
23     const personas = await crud.obtenerTodos(tabla);
24
25     // Respuesta con el arreglo de personas en formato JSON
26     res.json(personas);
27   } catch (error) {
28     // Si hay un error, se responde con código 500 y el mensaje del error
29     res.status(500).json({ error: error.message });
30   }
31 });
32
33 // Ruta para obtener una persona específica por su ID
34 router.get('/:id', async (req, res) => {
35   try {
36     // Utilizar el método obtenerUno con el ID recibido en la URL
37     const persona = await crud.obtenerUno(tabla, idCampo, req.params.id);
38
39     // Respuesta con los datos de la persona en formato JSON
40     res.json(persona);
41   } catch (error) {
42     // Manejar errores de servidor
43     res.status(500).json({ error: error.message });
44   }
45 });
46
47 // Ruta para crear una nueva persona (registro nuevo en la base de datos)
48 router.post('/', async (req, res) => {
49   try {
50     // Utilizar el método crear con los datos enviados en el cuerpo del request
51     const nuevaPersona = await crud.crear(tabla, req.body);
52
53     // Respuesta con el nuevo registro creado y código 201 (creado)
54     res.status(201).json(nuevaPersona);
55   } catch (error) {
56     // Manejar errores de servidor
57     res.status(500).json({ error: error.message });
58   }
59 });
60
61 // Ruta para actualizar una persona existente (por ID)
62 router.put('/:id', async (req, res) => {
63   try {
64     // Utilizar el método actualizar con el ID y los nuevos datos del cuerpo
65     const personaActualizada = await crud.actualizar(tabla, idCampo, req.params.id, req.body);
66
67     // Respuesta con el registro actualizado
68     res.json(personaActualizada);
69   } catch (error) {
70     // Manejar errores de servidor
71     res.status(500).json({ error: error.message });
72   }
73 });
74
75 // Ruta para eliminar una persona de la base de datos (por ID)
76 router.delete('/:id', async (req, res) => {
77   try {
78     // Utilizar el método eliminar con el ID recibido
79     const resultado = await crud.eliminar(tabla, idCampo, req.params.id);
80
81     // Respuesta con un mensaje o confirmación de eliminación
82     res.json(resultado);
83   } catch (error) {
84     // Manejar errores de servidor
85     res.status(500).json({ error: error.message });
86   }
87 });
88
89 // Exportar el router para que pueda ser usado en la aplicación principal
90 module.exports = router;
```

Configuración del archivo `imagenes.routes.js`

Este archivo de rutas del lado del servidor gestiona todas las operaciones relacionadas con el manejo de imágenes en la aplicación. A través de Express, define rutas específicas para subir, obtener, insertar y eliminar imágenes, recibiendo y enviando los datos en formato Base64. Cuando el cliente realiza una solicitud HTTP (por ejemplo, un PUT a `/subir/:tabla/:campold/:id` para actualizar una imagen), el servidor recibe dicha solicitud y la redirige hacia el controlador `imagenes.controller`, enviando como parámetros el nombre de la tabla, el campo clave y el ID del registro correspondiente. Esta estructura flexible permite aplicar la misma lógica de gestión de imágenes a múltiples entidades de la base de datos, simplemente especificando en la ruta el nombre de la tabla y del campo clave. Gracias a su diseño reutilizable y modular, este archivo facilita la escalabilidad del sistema, permitiendo integrar fácilmente el manejo de imágenes en futuras tablas sin necesidad de volver a escribir la lógica.

Ilustración 43. Configuración del archivo `imagenes.routes.js`



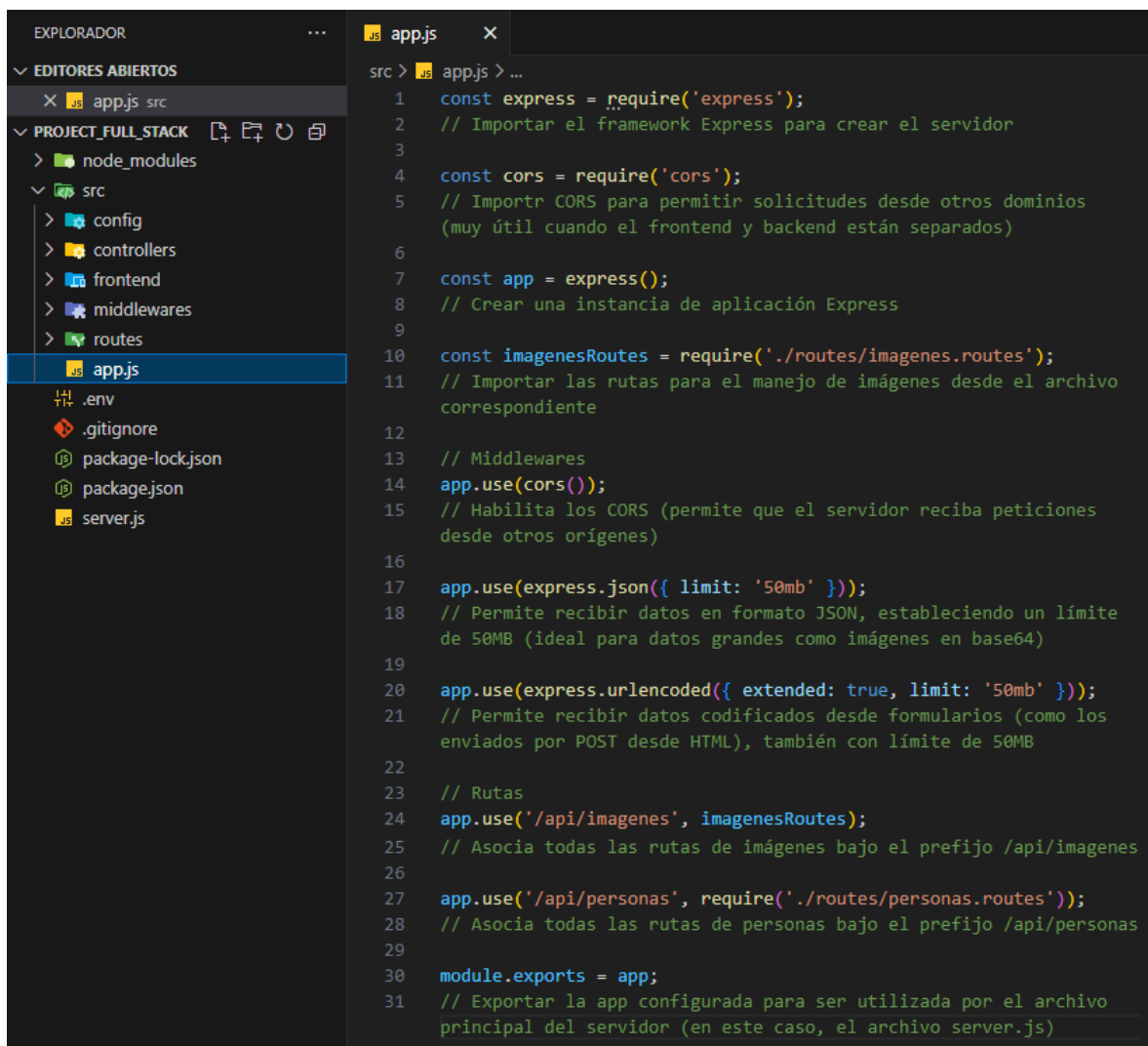
```
EXPLORADOR
...
EDITORES ABIERTOS
  X imagenes.routes.js src/routes
PROJECT_FULL_STACK
  > node_modules
  > src
    > config
    > controllers
    > frontend
    > middlewares
    > routes
      + imagenes.routes.js
      + personas.routes.js
      app.js
      .env
      .gitignore
      package-lock.json
      package.json
      server.js

src > routes > + imagenes.routes.js > router.put('/subir/tabla/campoId/:id') callback
1  const express = require('express');
2  const router = express.Router();
3  const multer = require('multer');
4  const imagenesController = require('../controllers/imagenes.controller');
5
6  // Ruta actualizar una imagen (Recibe la imagen en base64)
7  router.put('/subir/:tabla/:campoId/:id', async (req, res) => {
8    const { tabla, campoId, id } = req.params;
9    const imagenBase64 = req.body.imagen;
10
11    if (!imagenBase64) {
12      return res.status(400).json({ error: 'Se requiere la imagen en base64' });
13    }
14
15    try {
16      const resultado = await imagenesController.procesarImagen(tabla, campoId, id, imagenBase64);
17      res.json(resultado);
18    } catch (error) {
19      console.error('Error al subir la imagen:', error);
20      res.status(500).json({ error: 'Error al subir la imagen' });
21    }
22  });
23
24  // Ruta para obtener una imagen (Devuelve la imagen en base64)
25  router.get('/obtener/:tabla/:campoId/:id', async (req, res) => {
26    const { tabla, campoId, id } = req.params;
27
28    try {
29      const imagen = await imagenesController.procesarImagen(tabla, campoId, id);
30      res.json(imagen);
31    } catch (error) {
32      console.error('Error al obtener la imagen:', error);
33      res.status(500).json({ error: 'Error al obtener la imagen' });
34    }
35  });
36
37  // Ruta para eliminar una imagen (Pone el campo imagen a NULL)
38  router.delete('/eliminar/:tabla/:campoId/:id', async (req, res) => {
39    const { tabla, campoId, id } = req.params;
40
41    try {
42      const resultado = await imagenesController.eliminarImagen(tabla, campoId, id);
43      res.json(resultado);
44    } catch (error) {
45      console.error('Error al eliminar la imagen:', error);
46      res.status(500).json({ error: 'Error al eliminar la imagen' });
47    }
48  });
49
50  // Ruta para insertar una imagen (Recibe la imagen en base64)
51  router.post('/insertar/:tabla/:campoId/:id', async (req, res) => {
52    const { tabla, campoId, id } = req.params;
53    const imagenBase64 = req.body.imagen;
54
55    if (!imagenBase64) {
56      return res.status(400).json({ error: 'Se requiere la imagen en base64' });
57    }
58
59    try {
60      const resultado = await imagenesController.insertarImagen(tabla, campoId, id, imagenBase64);
61      res.json(resultado);
62    } catch (error) {
63      console.error('Error al insertar la imagen:', error);
64      res.status(500).json({ error: 'Error al insertar la imagen' });
65    }
66  });
67
68  module.exports = router;
```

Configuración del archivo app.js

El archivo `app.js`, ubicado dentro de la carpeta **src**, cumple una función fundamental en la arquitectura del servidor. Es el núcleo donde se configuran e integran los distintos componentes del backend. Aquí se incorporan los **middlewares** necesarios, como **cors** y la capacidad de interpretar cuerpos **JSON** grandes, además de las rutas principales del sistema como **personas.routes.js** e **imagenes.routes.js**. Estas rutas gestionan operaciones específicas, como el CRUD de personas y la manipulación de imágenes. A su vez, estas rutas se comunican con los controladores correspondientes, que son quienes realizan la lógica sobre la base de datos. Por ejemplo, cuando el cliente (**frontend**) envía una solicitud POST a `/personas`, el servidor la recibe en el archivo de rutas, que llama al método **crear()** del controlador, enviando como argumentos el nombre de la tabla (`personas`) y los datos en **req.body**. Esta estructura modular favorece un desarrollo limpio, reutilizable y escalable. Finalmente, **app.js** es exportado para ser utilizado en el archivo principal del servidor, como el archivo **server.js**, que se encarga de levantar la aplicación en el puerto deseado y poner todo el sistema en funcionamiento.

Ilustración 44. Configuración del archivo `app.js`



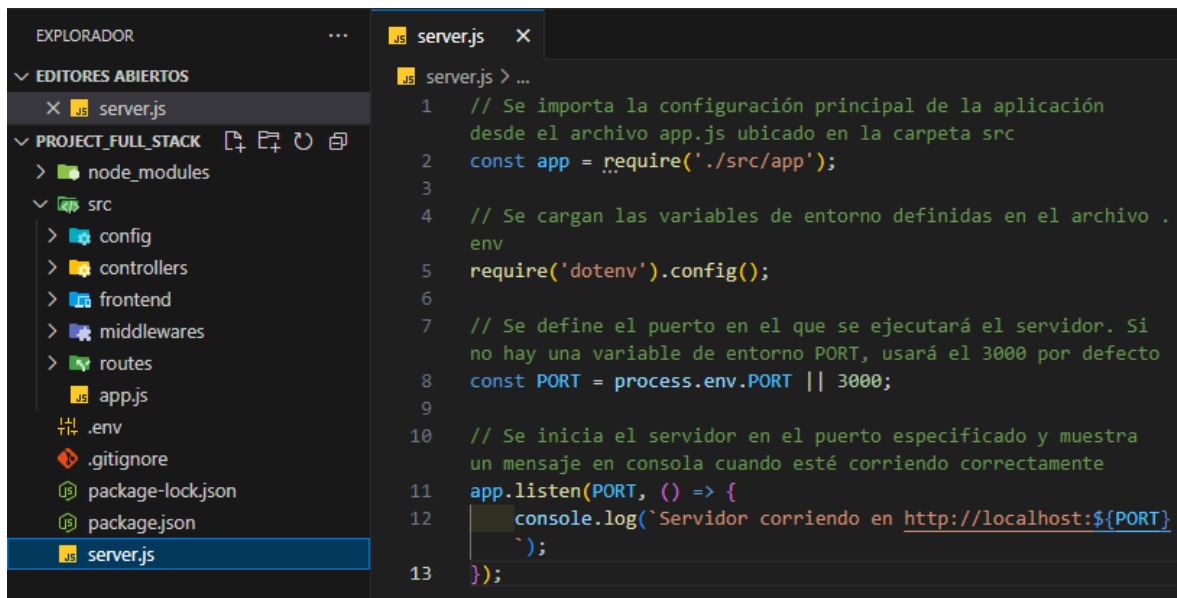
```
EXPLORADOR
...
EDITORES ABIERTOS
  X app.js src
PROJECT_FULL_STACK
  > node_modules
  > src
    > config
    > controllers
    > frontend
    > middlewares
    > routes
    X app.js
.env
.gitignore
package-lock.json
package.json
server.js

src > app.js > ...
1  const express = require('express');
2  // Importar el framework Express para crear el servidor
3
4  const cors = require('cors');
5  // Importar CORS para permitir solicitudes desde otros dominios
6  // (muy útil cuando el frontend y backend están separados)
7
8  const app = express();
9  // Crear una instancia de aplicación Express
10
11 const imagenesRoutes = require('./routes/imagenes.routes');
12 // Importar las rutas para el manejo de imágenes desde el archivo
13 // correspondiente
14
15 // Middlewares
16 app.use(cors());
17 // Habilita los CORS (permite que el servidor reciba peticiones
18 // desde otros orígenes)
19
20 app.use(express.json({ limit: '50mb' }));
21 // Permite recibir datos en formato JSON, estableciendo un límite
22 // de 50MB (ideal para datos grandes como imágenes en base64)
23
24 app.use(express.urlencoded({ extended: true, limit: '50mb' }));
25 // Permite recibir datos codificados desde formularios (como los
26 // enviados por POST desde HTML), también con límite de 50MB
27
28 // Rutas
29 app.use('/api/imagenes', imagenesRoutes);
30 // Asocia todas las rutas de imágenes bajo el prefijo /api/imagenes
31
32 app.use('/api/personas', require('./routes/personas.routes'));
33 // Asocia todas las rutas de personas bajo el prefijo /api/personas
34
35 module.exports = app;
36 // Exportar la app configurada para ser utilizada por el archivo
37 // principal del servidor (en este caso, el archivo server.js)
```

Configuración del archivo server.js

El archivo `server.js`, ubicado en la raíz del proyecto, es el punto de entrada del servidor backend. Su principal función es iniciar la aplicación importando el módulo principal `app.js` (definido en `src/app.js`, donde están configurados los `middlewares` y las `rutas`). Además, utiliza el paquete `dotenv` para leer variables de entorno desde un archivo `.env`, lo cual permite configurar dinámicamente el puerto (`PORT`) y otras variables sensibles sin modificar el código. Una vez determinado el puerto (por ejemplo, 3000 por defecto), el método `listen()` arranca el servidor y deja la API disponible, mostrando un mensaje en consola que indica que el servidor está activo y en qué `URL` local puede ser accedido. Este archivo es esencial para poner en marcha todo el ecosistema del backend.

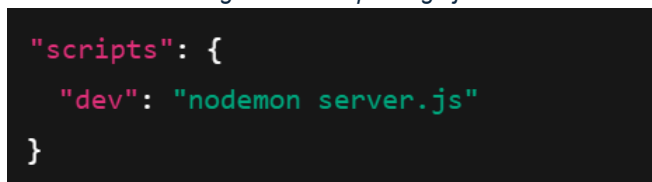
Ilustración 45. Configuración del `server.js`



Inicializar el backend con el comando `npm run dev`

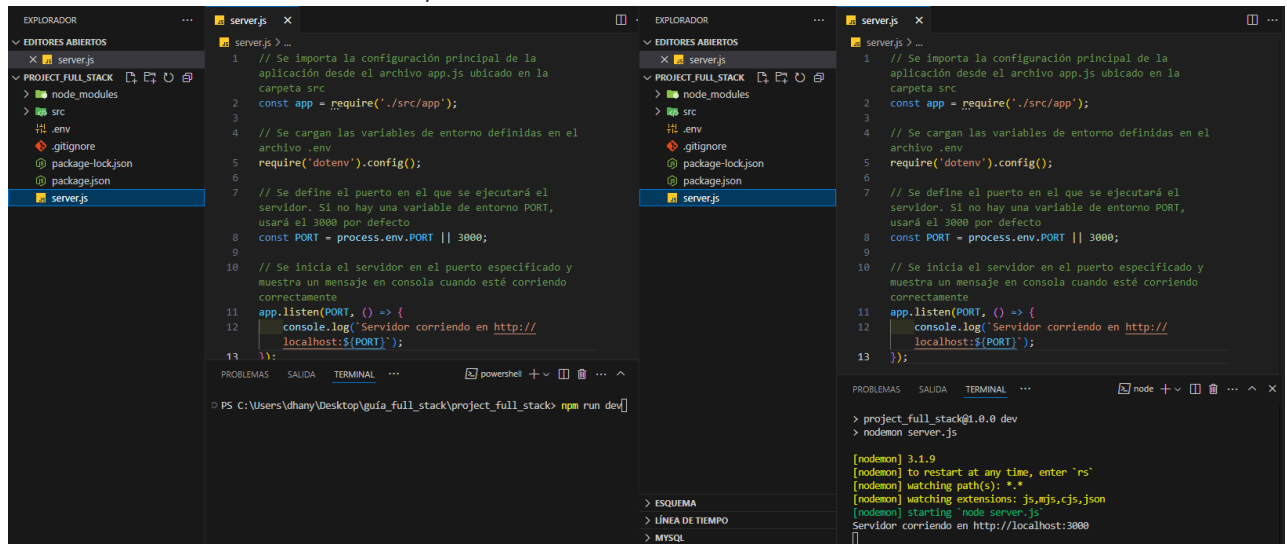
En la consola, este comando ejecuta el archivo `server.js` utilizando `nodemon`, una herramienta que reinicia automáticamente el servidor cada vez que detecta cambios en los archivos del proyecto. Esto es útil durante el desarrollo, ya que evita tener que detener y volver a iniciar el servidor manualmente. Para que este comando funcione, debe estar definido en el archivo `package.json` bajo la sección de `scripts`.

Ilustración 46. Configuración del `package.json` con `nodemon`



Este comando inicia todo el backend, incluyendo la carga de rutas, middlewares y controladores, permitiendo que la API esté disponible para atender las solicitudes del frontend o de herramientas de prueba como Postman o Insomnia.

Ilustración 47. Servidor corriendo en el puerto 3000



Archivo .gitignore: archivos, dependencias, paquetes y librerías a ignorar

El archivo .gitignore se utiliza para excluir archivos y carpetas que no deberían subirse al repositorio de Git (GitHub, GitLab, entre otros.). Esto ayuda a mantener el proyecto limpio, seguro y ligero.

¿Por qué es importante?

En Node.js, hay archivos que:

- Se generan automáticamente (como node_modules).
- Contienen configuraciones sensibles (como claves API o variables de entorno en .env).
- No son necesarios para compartir el código.

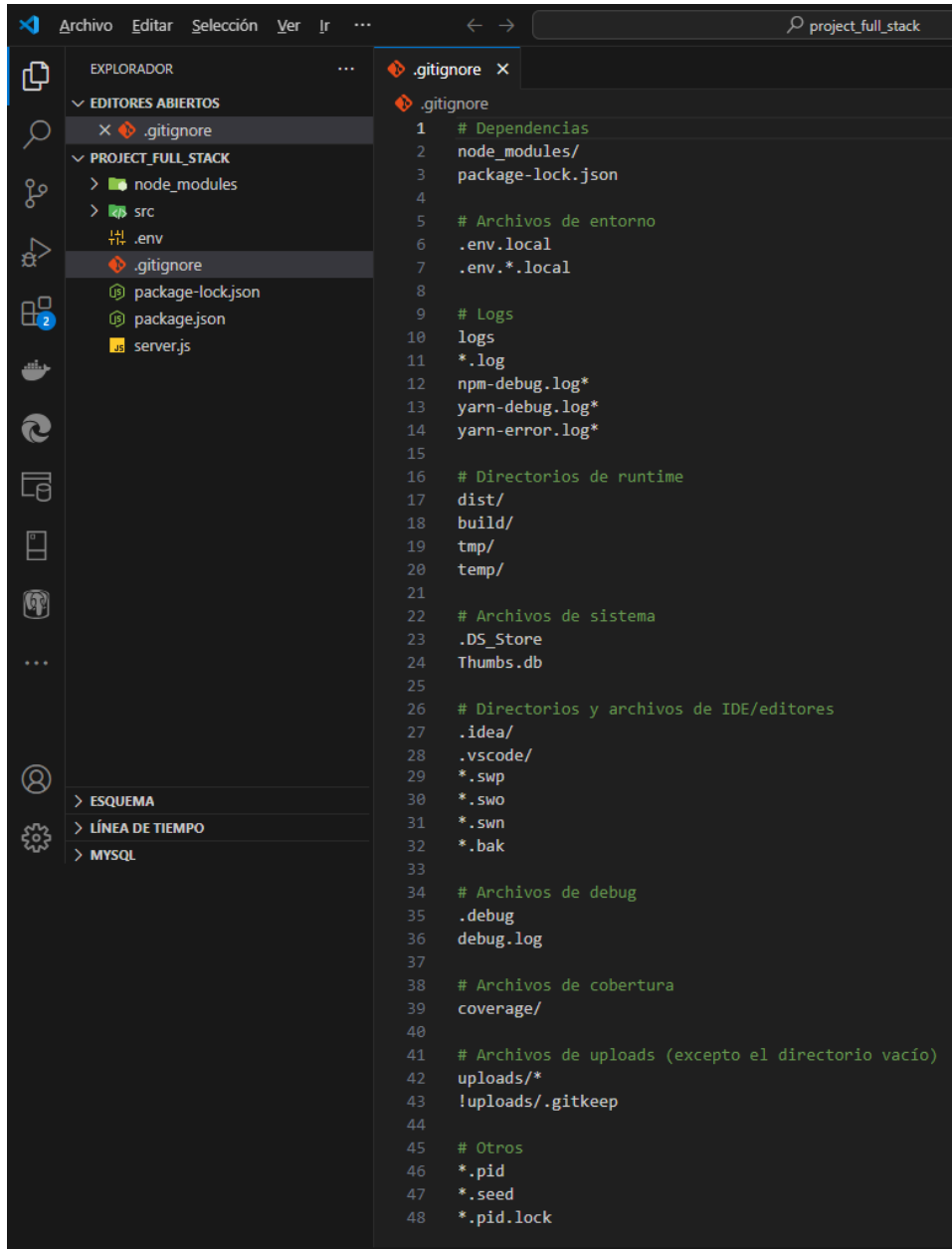
Subir estos archivos puede:

- Hacer el repositorio muy pesado.
- Exponer información confidencial.

¿Cómo funciona?

- Crear un archivo llamado `.gitignore` en la raíz del proyecto.
- Añadir las rutas o archivos que se quieren versionar.
- Git ignorará estos archivos al hacer `git add`, manteniéndolos fuera del repositorio.

Ilustración 48. Configuración del archivo `.gitignore`



Referencias bibliográficas

- Esquivel Paula, G. G., Quisaguano Collaguazo, L. R., Caluña Guaman, A. P., & Llambo Alvarez, S. J. (19 de Noviembre de 2024). Frameworks del lado del Servidor: Caso de Estudio Node JS, Django y Laravel. *593 Digital Publisher CEIT*, 10(1), 403-414. Recuperado el 18 de Abril de 2025, de <https://dialnet.unirioja.es/servlet/articulo?codigo=9966614>
- Haro, E., Guarda, T., Zambrano Peñaherrera, A. O., & Ninahualpa Quiña, G. (2019). Desarrollo backend para aplicaciones web, Servicios Web Restful: Node.js vs Spring Boot. *RISTI. Revista Ibérica de Sistemas e Tecnologias de Informação*(E17), 309-321. Recuperado el 18 de Abril de 2025, de <https://www.proquest.com/openview/a78cfaa62708fd24f38ac8d1025050eb/1?cbl=10063&pq-origsite=gscholar>
- Palmera Quintero, L. M., Ríos Baron, D. J., León, K., & Chinchilla Torres, F. (2024). Desarrollo de un aplicativo móvil con Node.js para la venta de productos agrícolas en MiPymes. *Revista Temario Científico*, 4(2), 1-12. Recuperado el 18 de Abril de 2025, de <https://alinin.org/ojs/index.php/temariocientifico/article/view/157/400>