

Guía Completa: Marketplace Inteligente con Asistente IA

Desarrollo Backend con Node.js, Express y Gemini AI

Información del aplicativo

- **Proyecto:** Sistema completo de marketplace con IA integrada
 - **Duración:** 7 módulos progresivos
 - **Nivel:** Intermedio (requiere conocimientos básicos de JavaScript, promesas, `async/await`)
 - **Tecnologías:** Node.js, Express, MySQL, JWT, bcrypt, Swagger, Gemini AI
-

Módulo 0: Introducción al Proyecto

Objetivos del Aprendizaje

Estimado aprendiz, al completar esta guía, serás capaz de:

- Diseñar y desarrollar APIs REST completas con Node.js y Express
- Implementar autenticación segura con JWT y bcrypt
- Gestionar diferentes tipos de peticiones HTTP (`query`, `params`, `body`, `files`, `headers`)
- Integrar inteligencia artificial (Gemini) en aplicaciones backend
- Aplicar validaciones robustas con `express-validator`
- Documentar APIs profesionalmente con Swagger
- Estructurar proyectos backend siguiendo mejores prácticas

Descripción del Proyecto: Marketplace Inteligente

Desarrollaremos un marketplace completo donde:

- **Usuarios** pueden registrarse como compradores o vendedores
- **Vendedores** publican productos con imágenes y descripciones
- **Compradores** buscan, filtran y compran productos
- **IA (Gemini)** asiste en:
 - Generar descripciones automáticas de productos

- Analizar y resumir reseñas de productos
- Sugerir categorías basadas en descripciones
- Responder preguntas automáticamente

Modelos del Sistema

1. Usuario (users)

- id (PK)
- nombre, email, password
- rol (comprador/vendedor/admin)
- fecha_registro

2. Producto (products)

- id (PK)
- vendedor_id (FK -> users.id)
- categoria_id (FK -> categories.id)
- nombre, descripcion, precio
- stock, imagen_url
- fecha_creacion

3. Orden (orders)

- id (PK)
- comprador_id (FK -> users.id)
- total, estado
- fecha_orden

4. Categoría (categories)

- id (PK)
- nombre, descripcion
- imagen_icono

Metodología de Aprendizaje

Enfoque Pedagógico Retador:

1. **Análisis:** Cada módulo inicia con preguntas de diseño y análisis
2. **Diseño:** Planificación antes de implementación
3. **Implementación Guiada:** Desarrollo paso a paso de UN modelo
4. **Desafío Autónomo:** Aplicar lo aprendido en OTROS modelos
5. **Reflexión:** Preguntas de comprensión y mejoras

Estructura por Módulo:

-  **Preguntas de Análisis (5-10 min)**
-  **Fase de Diseño (10-15 min)**

-  **Implementación Práctica** (30-45 min)
 -  **Desafío Autónomo** (45-60 min)
 -  **Reflexión y Mejoras** (10-15 min)
-

Módulo 1: Librerías, Fundamentos y Configuración

Morgan (npm i morgan)

Es un **middleware de registro (logging)** de solicitudes HTTP. Básicamente, Morgan se queda "escuchando" cada vez que alguien hace una petición a tu servidor y anota automáticamente los detalles en la consola o en un archivo de texto.

1. ¿Para qué sirve exactamente?

Cuando desarrollas una API, necesitas saber qué está pasando "bajo el capó". Morgan te responde preguntas como:

- ¿Qué ruta están visitando? (GET /users)
- ¿La respuesta fue exitosa o hubo un error? (Código 200, 404, 500)
- ¿Cuánto tardó el servidor en responder? (en milisegundos)
- ¿Quién hizo la petición? (IP, navegador, etc.)

2. Un ejemplo visual

Sin Morgan, tu consola está vacía. Con Morgan, cada vez que alguien entra a tu web, verás algo como esto en tu terminal:

GET /api/v1/productos 200 45.123 ms - 154

Traducción: Alguien pidió la lista de productos, todo salió bien (**200**), tardamos **45ms** y le enviamos **154 bytes**.

3. ¿Cómo se ve en el código?

Es extremadamente sencillo de implementar. Aquí tienes la configuración básica:

```
const express = require('express');
const morgan = require('morgan'); // 1. Importas
const app = express();

app.use(morgan('dev')); // 2. Lo usas como middleware

app.get('/', (req, res) => {
  res.send('¡Hola Mundo!');
});

app.listen(3000);
```

4. Los modos más comunes

Morgan tiene "sabores" preconfigurados según lo que necesites ver:

- **dev:** Es el más colorido y limpio. Ideal para cuando estás programando.
- **combined:** Estándar de Apache. Muy detallado (incluye IP, User-Agent, etc.), perfecto para servidores en producción.
- **tiny:** Lo mínimo indispensable.

Swagger

Para explicarlo de forma sencilla: **Swagger** es un conjunto de herramientas de código abierto que ayuda a los desarrolladores a diseñar, construir, documentar y consumir servicios web REST (APIs).

Si una API fuera un electrodoméstico nuevo, Swagger sería tanto el **manual de instrucciones** como el **panel de control** que te permite probar los botones para ver qué hacen.

1. ¿Cómo funciona?

Swagger se basa en la **Especificación OpenAPI (OAS)**. Imagina que OpenAPI es el idioma estándar (un archivo JSON o YAML) y Swagger es la herramienta que lee ese idioma para crear una interfaz visual amigable.

2. Los componentes principales

Swagger no es una sola cosa, sino un ecosistema:

- **Swagger UI:** Es lo más conocido. Transforma el archivo de código de la API en una página web interactiva donde puedes ver todos los "endpoints" (rutas), qué datos enviar y qué respuestas esperar.
 - **Swagger Editor:** Un editor en el navegador donde puedes escribir y editar la documentación de tu API y ver los cambios en tiempo real.
 - **Swagger Codegen:** Una herramienta que toma tu documentación y genera automáticamente código (SDKs) para que otros desarrolladores puedan integrar tu API más rápido.
-

3. ¿Por qué es tan importante?

Antes de Swagger, documentar una API era un caos: archivos de Word desactualizados o correos electrónicos confusos. Con Swagger:

1. **Documentación viva:** Si cambias algo en el código, la documentación se actualiza casi automáticamente.
2. **Pruebas interactivas:** No necesitas herramientas externas (como Postman) para probar la API; puedes hacer clic en "**Try it out**" directamente desde el navegador.
3. **Contrato claro:** Desarrolladores de Frontend y Backend se ponen de acuerdo en el "contrato" de la API antes de empezar a programar, evitando sorpresas.

Dato curioso: En 2015, la empresa SmartBear (dueña de Swagger) donó la especificación a la Linux Foundation, y ahí fue cuando pasó a llamarse oficialmente **OpenAPI**. Swagger hoy es el nombre comercial de las herramientas que usan ese estándar.

Express.js

Si Swagger es el manual y Morgan es el espía, **Express.js** es el **esqueleto** sobre el cual construyes todo.

Es el framework web más popular para **Node.js**. Su trabajo es facilitar enormemente la creación de servidores web y APIs, eliminando la complejidad de escribir código de red desde cero.

1. ¿Por qué todo el mundo lo usa?

Node.js por sí solo es muy potente, pero crear un servidor "a mano" es como intentar construir un coche pieza por pieza (tornillo por tornillo). Express te da el chasis, las ruedas y el motor listos para arrancar.

Sus pilares fundamentales:

- **Ruteo (Routing):** Te permite definir qué pasa cuando alguien visita una URL específica. Por ejemplo: `app.get('/perfil', ...)` le dice al servidor exactamente qué mostrar en esa sección.
 - **Middlewares:** Es un concepto clave. Son funciones que se ejecutan **entre** que llega la petición y se envía la respuesta. Aquí es donde entran herramientas como **Morgan** (para loguear) o **Swagger** (para documentar).
 - **Minimalismo:** No te obliga a usar una base de datos específica ni una estructura de carpetas rígida. Es un "lienzo en blanco" muy flexible.
-

2. Comparativa rápida

Para que veas la diferencia de "esfuerzo":

Característica	Node.js Puro (http module)	Con Express.js
Leer parámetros de URL	Manual y complejo	<code>req.params</code>
Manejar archivos estáticos	Requiere muchas líneas de código	<code>app.use(express.static('public'))</code>
Enviar un JSON	Debes configurar los Headers manualmente	<code>res.json({ hola: 'mundo' })</code>
Curva de aprendizaje	Empinada	Muy suave

3. El ciclo de vida: Petición -> Respuesta

Imagina que un usuario pide una lista de usuarios en tu API:

1. **Petición:** El cliente pide GET /users.
 2. **Middlewares:** Pasa por **Morgan** (anota la visita) y quizás por un validador de seguridad.
 3. **Ruta:** Express identifica que tiene una ruta para /users.
 4. **Lógica:** Tu código busca los datos en la base de datos.
 5. **Respuesta:** Express envía el JSON al cliente.
-

El "Combo Ganador"

Normalmente, en un entorno profesional, verás a estos tres trabajando juntos de esta manera:

1. **Express** maneja la lógica del servidor.
2. **Morgan** registra cada movimiento para que no te pierdas nada.
3. **Swagger** genera la página web donde otros pueden probar tu API de Express sin tocar el código.

Express-validator

Si **Express** es el esqueleto de tu servidor, **Express-validator** es el **filtro de seguridad** en la puerta.

Es una librería de middlewares que se encarga de revisar que los datos que envía el usuario (en el cuerpo del mensaje, la URL o los encabezados) sean exactamente lo que esperas. Si esperas un correo electrónico y te envían "hola", esta herramienta lo detiene antes de que llegue a tu base de datos.

1. Concepto básico: ¿Cómo funciona?

Funciona en dos pasos:

1. **Definir las reglas:** Qué campo quieras validar y qué requisitos debe cumplir.
 2. **Verificar errores:** Revisar si alguna regla falló y decidir qué responderle al usuario.
-

2. Ejemplos prácticos

Ejemplo A: Registro de usuario (Lo más común)

Imagina que quieres validar un formulario de registro con nombre, email y una contraseña segura.

```
const { body, validationResult } = require('express-validator');

app.post('/registro', [
  // 1. Definimos las reglas
  body('nombre').notEmpty().withMessage('El nombre es obligatorio'),
  body('email').isEmail().withMessage('Debe ser un correo válido'),
```

```

body('password').isLength({ min: 6 }).withMessage('Mínimo 6 caracteres')
], (req, res) => {
  // 2. Verificamos si hubo errores
  const errores = validationResult(req);
  if (!errores.isEmpty()) {
    return res.status(400).json({ errores: errores.array() });
  }

  res.send('¡Usuario registrado con éxito!');
});

```

Ejemplo B: Validaciones Avanzadas y Sanitización

No solo validamos, también **limpiamos** los datos (quitar espacios en blanco, convertir a minúsculas, etc.).

```

body('username')
  .trim()          // Quita espacios: " juan " -> "juan"
  .isLength({ min: 5 })
  .escape(),       // Convierte caracteres peligrosos (<, >, &) para evitar XSS

body('edad')
  .isInt({ min: 18, max: 99 })
  .withMessage('Debes ser mayor de edad'),

body('fecha_nacimiento')
  .isISO8601()    // Formato de fecha estándar YYYY-MM-DD
  .toDate()        // Convierte el string a un objeto Date de JS

```

Ejemplo C: Validaciones Personalizadas (Custom)

A veces las reglas estándar no alcanzan. Por ejemplo, confirmar que dos contraseñas coincidan:

```

body('confirmPassword').custom((value, { req }) => {
  if (value !== req.body.password) {
    throw new Error('Las contraseñas no coinciden');
  }
  return true;
})

```

O que un usuario con id X exista en la base de datos mongo en la colección Y.

```

body('id').custom((value, { req }) => {
  const existe = await Usuario.findById(id)
  if (!existe) {
    throw new Error(`Registro no existe ${id}`)
  }

  req.req.usuariobd = existe
},

```

O que al agregar no existan dos usuarios con el mismo email

```

body('email').custom((value, { req }) => {
  if (email) {
    const existe = await Usuario.findOne({ email })
    if (existe) {
      throw new Error(`Ya existe ese email en la base de datos!!! ${email}`)
    } else {

```

```
        throw new Error(`Ya existe ese email en la base de datos!!! ${email}`);
    }
},
,
```

3. Estructura profesional (Middleware reutilizable)

En un proyecto real, no querrás repetir el código de "verificar errores" en cada ruta. Lo mejor es crear un middleware aparte:

```
// validator.js

const { validationResult } = require('express-validator');

const validarCampos = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json(errors);
  }
  next(); // Si no hay errores, continúa a la lógica principal
};

// En tu ruta:

app.post('/login', [
  body('email').isEmail(),
  body('password').notEmpty(),
  validarCampos // <--- Aquí se ejecuta la limpieza
], (req, res) => {
  // Lógica de login limpia de validaciones
});
```

CORS

Si estás construyendo una API con **Express**, te vas a encontrar con **CORS** tarde o temprano (generalmente en forma de un error rojo y molesto en la consola del navegador).

CORS (*Cross-Origin Resource Sharing*) es un mecanismo de seguridad que permite (o restringe) que un sitio web acceda a recursos de otro dominio diferente.

1. ¿Por qué existe? (El problema)

Por seguridad, los navegadores prohíben que el JavaScript de un sitio (ej. mi-app-frontend.com) haga peticiones a un servidor de otro dominio (ej. mi-api-backend.com). Esto se llama **Política de Mismo Origen** (*Same-Origin Policy*).

Sin esta protección, un sitio malicioso podría intentar robar tus datos de sesión de Facebook o de tu banco mientras navegas por otra pestaña.

2. ¿Cómo funciona CORS?

Cuando tu Frontend intenta hablar con tu Backend, ocurre un "apretón de manos":

1. **La Petición:** El navegador envía una cabecera llamada Origin (diciendo quién es).
2. **La Verificación:** El servidor revisa si ese origen está en su "lista de invitados".

-
3. **La Respuesta:** Si el servidor acepta, responde con la cabecera Access-Control-Allow-Origin. Si no, el navegador bloquea la respuesta y lanza el famoso error de CORS.
-

3. ¿Cómo se soluciona en Express?

En lugar de configurar las cabeceras a mano, usamos el paquete cors, que es el estándar de la industria.

Instalación:

```
npm install cors
```

Uso básico (Abrir a todo el mundo):

Esto permite que **cualquier** dominio acceda a tu API (útil para APIs públicas).

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

app.use(cors()); // <--- CORS habilitado para todas las rutas
```

Uso profesional (Restringido):

En producción, solo quieres que **tu** frontend pueda entrar:

```
const corsOptions = {
  origin: 'https://mi-app-real.com', // Solo este dominio entra
  optionsSuccessStatus: 200
};

app.use(cors(corsOptions));
```

El equipo completo en una API Real

Ahora que conoces a todos los integrantes, así se ve una estructura profesional típica:

- **Express:** El motor que corre el servidor.
- **Morgan:** El diario que anota quién entra y qué pide.
- **CORS:** El portero que decide quién puede pasar según su origen.
- **Swagger:** El mapa que le entregas a los invitados para que sepan usar la casa.

bcrypt

es la **caja fuerte** donde guardas las joyas (las contraseñas) de tus usuarios. Es una librería diseñada específicamente para realizar el *hashing* de contraseñas de forma segura.

1. ¿Por qué no guardar contraseñas tal cual?

Si guardas la contraseña "123456" en tu base de datos y alguien hackea tu servidor, el hacker tendrá todas las cuentas. **Nunca, jamás**, se guardan contraseñas en texto plano.

¿Cómo lo soluciona bcrypt?

Bcrypt convierte "123456" en algo como: \$2b\$10\$nOUIs5kJ7naTuTFkBy1veuK0kS...

2. Conceptos clave: Salt y Round

- **Salt (Sal):** Es un pedazo de datos aleatorios que se añade a la contraseña antes de procesarla. Esto evita que dos usuarios con la misma contraseña tengan el mismo hash.
- **Cost Factor (Rounds):** Es el número de veces que se ejecuta el algoritmo. Cuanto más alto, más tarda (más difícil de hackear por fuerza bruta), pero más recursos consume tu servidor. El estándar actual es **10**.

Para que lo entiendan con una analogía:

Texto Plano (El Soplón): Una contraseña sin encriptar es un soplón que le cuenta todo al hacker en cuanto este entra a la base de datos.

Salt (El Camuflaje): Es como ponerle bigote y peluca a la contraseña para que, aunque dos personas usen "123456", sus hashes se vean totalmente diferentes.

Rounds (La Profundidad de la Bóveda): Cada "round" es una puerta blindada extra que el hacker debe derribar por fuerza bruta.

3. Ejemplos de uso en Express

Ejemplo A: Hashear al registrar un usuario (Encriptar)

Cuando el usuario crea su cuenta, "hasheas" la clave antes de guardarla en la base de datos.

```
const bcrypt = require('bcrypt');

const registrarUsuario = async (passwordPlano) => {
  const saltRounds = 10;

  // Generamos el hash (esto es asíncrono)
  const passwordHasheado = await bcrypt.hash(passwordPlano, saltRounds);

  console.log(passwordHasheado);
  // Resultado: $2b$10$X7... (Esto es lo que guardas en la DB)
};
```

Ejemplo B: Comparar al iniciar sesión (Login)

Bcrypt es un algoritmo de **una sola vía**. No puedes "desencriptar" el hash para ver la clave original. Para validar, bcrypt compara el texto que ingresa el usuario con el hash guardado.

```
const login = async (passwordIngresado, passwordEnBaseDatos) => {
  // Comparamos el texto plano con el hash guardado
  const esCorrecto = await bcrypt.compare(passwordIngresado, passwordEnBaseDatos);

  if (esCorrecto) {
    console.log("¡Bienvenido!");
  } else {
    console.log("Credenciales incorrectas");
```

```
    }  
};
```

Un consejo de seguridad

Siempre usa las funciones **asíncronas** (`bcrypt.hash` y `bcrypt.compare`) con `await`. Las versiones síncronas bloquean el servidor de Node.js mientras calculan el hash, lo que puede hacer que tu App se sienta lenta si muchos usuarios intentan entrar al mismo tiempo.

JSON Web Token

JSON Web Token (JWT) es el **pase VIP** que le entregas al usuario una vez que demostró quién es.

En el desarrollo web moderno, no queremos que el usuario nos envíe su usuario y contraseña en cada clic (sería lento e inseguro). En su lugar, usamos un JWT.

1. ¿Qué es un JWT?

Es un estándar (RFC 7519) para transmitir información de forma segura entre las partes como un objeto JSON. Esta información puede ser verificada y es confiable porque está **firmada digitalmente**.

La estructura de un Token

Un JWT tiene tres partes separadas por puntos (.):

1. **Header:** Indica el algoritmo de cifrado.
 2. **Payload:** Los datos del usuario (id, nombre, rol). **Ojo!** No guardes contraseñas aquí, ya que cualquiera puede decodificarlo.
 3. **Signature:** La firma que garantiza que el token no fue alterado.
-

2. El flujo de autenticación (La analogía del brazalete)

1. **Login:** El usuario envía sus credenciales.
 2. **Validación:** El servidor usa **bcrypt** para revisar la clave.
 3. **Emisión:** Si todo es correcto, el servidor genera un **JWT** firmado con una "llave secreta" y se lo entrega al usuario.
 4. **Acceso:** En las siguientes peticiones, el usuario envía el token en el Header. El servidor solo verifica la firma. Si es válida, lo deja pasar.
-

3. Ejemplo práctico con jsonwebtoken

A. Generar un Token (Al hacer Login)

```
const jwt = require('jsonwebtoken');  
  
const login = (user) => {
```

```

// Datos que queremos guardar en el token
const payload = { id: user.id, rol: 'admin' };

// Firmamos el token con una clave secreta y le damos expiración
const token = jwt.sign(payload, 'mi_llave_secreta_123', { expiresIn: '2h' });

return token;
}

```

B. Verificar un Token (Middleware de protección)

Este es el código que protege tus rutas privadas en **Express**:

```

const verificarToken = (req, res, next) => {
  const token = req.header('auth-token');
  if (!token) return res.status(401).send('Acceso denegado');

  try {
    const verificado = jwt.verify(token, 'mi_llave_secreta_123');
    req.user = verificado; // Guardamos los datos del usuario en la petición
    next(); // Continuamos a la ruta
  } catch (error) {
    res.status(400).send('Token no válido');
  }
};

```

// Uso en una ruta protegida

```

app.post('/login', [
  verificarToken,
  body('email').isEmail(),
  body('password').notEmpty(),
  validarCampos // <--- Aquí se ejecuta la limpieza
], (req, res) => {
  // Lógica de login limpia de validaciones
});

```

4. El "Dream Team" de tu API

Ahora todas las piezas encajan perfectamente:

1. **Express:** El servidor base.
2. **CORS:** Permite que tu Frontend pida el Token.
3. **Morgan:** Registra los intentos de acceso.
4. **Express-validator:** Revisa que el login traiga email y password.
5. **Bcrypt:** Compara la contraseña enviada con la de la DB.
6. **JWT:** Crea el token de acceso para que el usuario navegue sin loguearse otra vez.
7. **Swagger:** Documenta que la ruta /dashboard requiere un Token en el encabezado.

Un consejo vital:

Nunca suban la "llave secreta" a GitHub. Usa archivos de entorno (.env) para guardarla. Si alguien roba esa llave, podría crear tokens falsos y hacerse pasar por administrador de tu sistema.

dotenv

dotenv es el **guardaespalda de los secretos**. Es una librería que carga variables de entorno desde un archivo llamado .env hacia process.env. Su función principal es separar la configuración sensible (como contraseñas, llaves de API o puertos) del código fuente.

1. ¿Por qué es obligatorio usarlo?

Imagina que subes tu código a GitHub. Si en tu archivo de conexión a la base de datos escribiste: password: 'mi_clave_super_secreta_123', **todo el mundo podrá verla**.

Con **dotenv**, en tu código escribes una referencia variable, y el valor real se queda en un archivo local en tu computadora que **nunca** se sube a internet.

2. Cómo se implementa

Paso 1: Crear el archivo .env

En la raíz de tu proyecto (donde está el package.json), creas un archivo de texto llamado simplemente .env (ejemplo de contenido)

```
PORT=3000
DB_HOST=localhost
DB_USER=root
DB_PASS=mi_clave_secreta_123
JWT_SECRET=una_firma_muy_larga_y_segura
```

Paso 2: Cargar dotenv en tu App

Debes hacerlo lo más arriba posible en tu archivo principal (index.js o app.js):

```
require('dotenv').config(); // Esto carga las variables automáticamente

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

// Ahora puedes usar los valores
const puerto = process.env.PORT || 4000;

app.listen(puerto, () => {
  console.log(`Servidor corriendo en el puerto ${puerto}`);
});
```

Paso 3: Usarlo en otros módulos (ej. con mysql2)

```
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
```

```
password: process.env.DB_PASS,  
database: 'mi_base_de_datos'  
});
```

3. La Regla de Oro: .gitignore

Para que **dotenv** realmente funcione como medida de seguridad, debes crear un archivo llamado **.gitignore** y añadir dentro:

```
.env  
node_modules
```

Esto le dice a Git: "No subas el archivo **.env** a la nube".

Un truco profesional

Normalmente se crea un archivo llamado **.env.template** o **.env.example** con los nombres de las variables pero sin los valores reales. Así, cuando otro programador descargue tu proyecto, sabrá qué datos necesita configurar para que la App funcione.

mysql2

Por ultimo, necesitamos conectarnos a la base de datos, para este proyecto, vamos a usar **MySQL**, y para conectarnos desde **node** usaremos la librería **mysql2**. **Mysql2** Es un controlador (driver) para **Node.js** que es más rápido y seguro que el paquete **mysql** original. Su función principal es enviar consultas SQL desde tu código **JavaScript** y recibir los resultados.

1. ¿Por qué el "2"? (Diferencias clave)

Seguramente viste que existe **mysql** y **mysql2**. Aquí el porqué preferimos el 2:

- **Soporte de Promesas:** Te permite usar **async/await**, lo cual hace que tu código sea mucho más limpio (el original usa puros **callbacks**).
 - **Prepared Statements:** Es mucho más seguro contra ataques de **Inyección SQL**.
 - **Mayor Rendimiento:** Es significativamente más rápido procesando datos.
-

2. Cómo se usa en un entorno profesional

A. Configuración de la conexión (Pool)

En lugar de abrir y cerrar una conexión por cada clic, usamos un **Pool** (un grupo de conexiones abiertas que se reutilizan).

```
const mysql = require('mysql2/promise'); // Importante usar /promise  
  
const pool = mysql.createPool({  
  host: 'localhost',  
  user: 'root',  
  password: 'password_seguro',
```

```

    database: 'mi_api_db',
    waitForConnections: true,
    connectionLimit: 10
});

module.exports = pool;

```

B. Ejemplo: Registro con Bcrypt + MySQL2

Mira cómo se une todo lo que hemos visto:

```

const bcrypt = require('bcrypt');
const pool = require('./db'); // El archivo de arriba

app.post('/registrar', async (req, res) => {
  const { email, password } = req.body;

  // 1. Hasheamos la clave con Bcrypt
  const hash = await bcrypt.hash(password, 10);

  try {
    // 2. Insertamos en la DB usando mysql2
    const [result] = await pool.query(
      'INSERT INTO usuarios (email, password) VALUES (?, ?)',
      [email, hash] // Los "?" evitan inyección SQL
    );

    res.status(201).json({ id: result.insertId, mensaje: 'Usuario creado' });
  } catch (error) {
    res.status(500).json({ error: 'Error al guardar en DB' });
  }
});

```

Por qué usamos un Pool en lugar de conexiones individuales, usaremos la analogía de un Restaurante de Comida Rápida

Sin Pool: Cada cliente que llega obliga a contratar a un nuevo cocinero, comprarle utensilios y luego despedirlo cuando termina la hamburguesa. (Lento e ineficiente).

Con Pool: Tienes 10 cocineros ya listos (conexiones abiertas). El cliente llega, un cocinero libre lo atiende y, al terminar, el cocinero se queda en la cocina esperando al siguiente cliente.

¿ORM o Driver directo?

mysql2 es un **driver directo** (escribes SQL puro). Si prefieres no escribir SQL y manejar todo como objetos de JavaScript, existen los **ORMs** como **Sequelize** o **Prisma**, que por debajo usan mysql2 para funcionar, aunque para este proyecto lo vamos a hacer **SIN ORMs**.

Axios

Es una librería basada en **Promesas** que sirve para realizar peticiones HTTP. Es la evolución moderna y profesional del antiguo `fetch()` nativo del navegador.

1. ¿Por qué usar Axios y no fetch()?

Aunque fetch() ya viene en los navegadores, Axios es el estándar en la industria por varias razones:

- **Transformación JSON automática:** No tienes que hacer .json(), Axios ya te entrega los datos listos.
 - **Interceptores:** Puedes "detener" una petición antes de que salga (por ejemplo, para pegarle automáticamente el JWT que aprendimos antes).
 - **Soporte para navegadores antiguos:** Funciona en lugares donde fetch podría dar problemas.
 - **Manejo de errores:** Si el servidor responde un código **4xx** o **5xx**, Axios lanza un error automáticamente que puedes atrapar con un catch.
-

2. Ejemplos de uso (El cliente de tu API)

A. Petición GET (Obtener datos)

```
const axios = require('axios');

async function obtenerUsuarios() {
  try {
    const respuesta = await axios.get('https://tu-api.com/users');
    console.log(respuesta.data); // Los datos ya están convertidos a objeto
  } catch (error) {
    console.error("Error al obtener datos:", error.response.status);
  }
}
```

B. Petición POST con Seguridad (Enviando el JWT)

Aquí es donde conectamos todo lo que has aprendido. Enviamos datos validados y el token de acceso:

```
const nuevoPost = {
  titulo: 'Mi primer post',
  contenido: 'Contenido validado por express-validator'
};

axios.post('https://tu-api.com/posts', nuevoPost, {
  headers: {
    'auth-token': 'AQUÍ_VA_EL_JWT_QUE_GENERAMOS'
  }
})
.then(res => console.log('Post creado:', res.status))
.catch(err => console.log('Error de CORS o Validación:', err.message));
```

Multer

Es el **receptor de almacén** de tu API. **Multer** es un middleware para Express que se utiliza para manejar el envío de **archivos** (imágenes, PDFs, videos, etc.). En términos técnicos, se encarga de procesar peticiones con el formato multipart/form-data, que es el estándar cuando un formulario HTML incluye un <input type="file">.

1. ¿Por qué no usar simplemente req.body?

Por defecto, Express no sabe leer archivos. Si intentas enviar una foto y la buscas en req.body, verás que está vacío o corrupto. Los archivos son mucho más pesados y complejos que un simple texto, por eso necesitas a **Multer** para que:

1. Reciba el flujo de datos.
 2. Los guarde en una carpeta (o en memoria).
 3. Les cambie el nombre para que no se sobrescriban.
-

2. Ejemplo Práctico: Subir una foto de perfil

A. Configuración básica

Primero le decimos a Multer dónde guardar los archivos y cómo nombrarlos:

```
const multer = require('multer');
const path = require('path');

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/'); // Los archivos irán a la carpeta /uploads
  },
  filename: (req, file, cb) => {
    // Le ponemos un nombre único: timestamp + nombre original
    cb(null, Date.now() + path.extname(file.originalname));
  }
});

const upload = multer({ storage: storage });
```

B. Uso en una ruta de Express

```
//supongamos que 'fotoPerfil' es el nombre del campo que viene desde el Frontend (Axios)
app.post('/perfil/foto', upload.single('fotoPerfil'), (req, res) => {
  console.log(req.file); // Aquí está la info del archivo (tamaño, ruta, etc.)
  console.log(req.body); // Aquí están los otros campos de texto si los hay
  res.send('¡Imagen subida con éxito!');
});
```

3. Características clave

- **Filtros de seguridad:** Puedes restringir que solo se suban ciertos tipos de archivos (ej. solo .jpg o .png) y poner un límite de peso (ej. máximo 2MB).
- **Múltiples archivos:** Tiene funciones como .array() para subir varias fotos a la vez o .fields() para diferentes tipos de archivos en un solo formulario.
- **Almacenamiento en memoria:** Puedes elegir no guardar el archivo en el disco y procesarlo directamente (por ejemplo, para mandarlo a la nube como AWS S3 o Cloudinary).

```
mindflow-backend/
├── src/
│   ├── controllers/      # Lógica de las peticiones
│   ├── middlewares/     # Validaciones y Seguridad (JWT)
│   ├── models/           # Consultas SQL puras
│   ├── routes/           # Definición de endpoints
│   ├── utils/            # Ayudantes (como el cliente de Gemini)
│   └── index.js          # Punto de entrada de la app
├── .env                 # Credenciales sensibles
└── swagger.json         # Configuración de Swagger
└── package.json
```

🧐 Preguntas de Análisis Inicial

Antes de escribir código, responde:

1. Arquitectura de Directorios:

- ¿Por qué creen que separamos models, routes, controllers y middlewares?
- ¿Qué ventajas tiene esta estructura vs. un archivo monolítico?
- ¿Dónde ubicarías la configuración de base de datos? ¿Por qué?

2. Gestión de Dependencias:

- ¿En el package.json, cuál es la diferencia entre dependencies y devDependencies?
- ¿Por qué cors es necesario en una API?
- ¿Cuándo usarías morgan vs. un logger más robusto?

3. Conexión a Base de Datos:

- ¿Pool de conexiones vs. conexión única? ¿Cuándo usar cada uno?
- ¿Cómo manejarías diferentes configuraciones (dev, test, prod)?

📐 Fase de Diseño

Diseña antes de implementar:

1. Estructura del Proyecto Real

```

marketplace-api/
└── src/
    ├── config/          # ¿Qué va aquí?
    ├── controllers/     # ¿Responsabilidad?
    ├── middlewares/    # ¿Tipos de middleware necesarios?
    ├── models/          # ¿Solo consultas o también validaciones?
    ├── routes/          # ¿Agrupación por recurso?
    └── utils/           # ¿Qué utilidades comunes necesitamos?
    ├── uploads/         # ¿Estructura para archivos?
    ├── .env             # ¿Qué variables necesitamos?
    └── index.js         # ¿Responsabilidades del archivo principal?
...

```

2. Flujo de una Petición HTTP

Si han leído y entendido, saben que tenemos al menos estos roles en nuestra API:

- CORS (portero)
- Morgan (reportero)
- express.json()
- JWT middleware
- Controlador
- Modelo
- Base de datos

Entonces Un flujo exitoso seria:

Cliente → CORS → Morgan → Validaciones → JWT → Controlador → Modelo → DB →

Que viéndolo en palabras seria:

Cuando el **cliente** (por ejemplo un frontend hecho con React o una petición desde Postman) realiza una solicitud HTTP hacia nuestra API, la petición entra primero por **CORS**, que actúa como el portero del sistema verificando si el origen está autorizado para comunicarse con el servidor. Si el origen no está permitido, la solicitud se bloquea inmediatamente. Si pasa este filtro, la petición continúa hacia **Morgan**, que registra todos los detalles del intento (método, ruta, código de respuesta, tiempo de ejecución), permitiéndonos monitorear qué está ocurriendo en el servidor. Luego entran las **validaciones** (express-validator), donde se revisa que los datos enviados en body, params o query cumplan con las reglas definidas; si algo no cumple, la ejecución se detiene con un error 400. Si los datos son correctos y la ruta lo requiere, el flujo pasa por el middleware de **JWT**, que verifica la autenticidad del token y confirma la identidad y permisos del usuario. Solo si todo es válido, la petición llega al **controlador**, donde vive la lógica de negocio. El controlador delega el acceso a datos al **modelo**, que se encarga de construir y ejecutar las consultas contra la **base de datos (DB)**. Finalmente, la respuesta recorre el camino inverso hasta llegar nuevamente al cliente con un código HTTP que indica el resultado de toda la operación.

Ahora tenemos los siguientes escenarios:

- El token está vencido.

- El body está mal formado.
- El usuario no es admin.

respondan:

- Cual es el flujo en cada caso
- ¿Quién detiene la petición?
- ¿Qué código HTTP devuelve?
- ¿Quién responde?

👉 Si lo hacen a conciencia, esto hará que entiendan la arquitectura mentalmente.

3. Códigos de Respuesta HTTP

Los códigos de estado de respuesta HTTP indican si se ha completado satisfactoriamente una solicitud HTTP específica.

La referencia más confiable y visual para esto es la de **MDN Web Docs** (de Mozilla). Es el estándar de la industria.

Aquí tienes el enlace directo:

👉 [MDN - Códigos de estado de respuesta HTTP](#)

1. Resumen rápido por categorías

Para que no se pierdan, los códigos se dividen en 5 grupos según su primer número:

Rango	Categoría	Significado
1xx	Informativos	"Espera un momento, estamos procesando".
2xx	Éxito	"¡Todo salió bien!" (El más famoso es el 200 OK).
3xx	Redirección	"Lo que buscas se movió a otro lugar".
4xx	Error del Cliente	"Tú (el cliente) hiciste algo mal" (ej. 404 Not Found).
5xx	Error del Servidor	"Yo (el servidor) me rompí" (ej. 500 Internal Server Error).

2. Los más importantes para nuestra API de Express

Como estás usando **Express**, **Morgan** y **JWT**, estos son los que más vas a escribir en tu código:

- **200 OK**: Petición exitosa (ej. un GET de productos).
- **201 Created**: Éxito al crear algo (ej. después de un POST con **Bcrypt**).
- **400 Bad Request**: Los datos están mal (cuando **Express-validator** falla).
- **401 Unauthorized**: No hay token o es inválido (falla de **JWT**).

- **403 Forbidden:** El token es válido, pero el usuario no tiene permiso (ej. no es admin).
 - **404 Not Found:** La ruta o el recurso no existen.
 - **500 Internal Server Error:** Tu código explotó (un error de base de datos, por ejemplo).
-

3. Un toque de humor: HTTP Cats

Si quieren aprenderlos de una forma mucho más divertida y visual, existe una página famosa que explica cada código con **fotos de gatos**:

👉 [HTTP.cat](#)

Para cada escenario, ¿qué código HTTP usarías? Pongan el gato

- Usuario creado exitosamente: _____
- Email ya existe: _____
- Token inválido: _____
- Recurso no encontrado: _____
- Error del servidor: _____



1. “CSI: Error 500” – Laboratorio Forense de Bugs



Escenario: En una petición que hicimos recibimos este mensaje

```
{  
  "error": true,  
  "mensaje": "Error interno del servidor"  
}
```

En la consola observamos el siguiente log de Morgan:

POST /api/usuarios 500 123.45 ms – 89



Actividad

Debate con algún compañero y responde:

1. ¿El error fue del cliente o del servidor?
2. ¿En qué capa pudo ocurrir?
 - Validación
 - Controlador
 - Modelo
 - Base de datos
3. ¿Qué harían para evitar que vuelva a ocurrir?

 Manos a la obra, Implementation Práctica

Paso 1: Configuración del Proyecto

1. creen la carpeta de su proyecto
2. creen su proyecto (npm init)
3. creen la estructura de carpetas del proyecto
4. instalen librerías
5. creen su archivo .env y el .ignore
6. recomendable nodemon para el debug

Paso 2: Configuración Basica index.js

```
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');
require('dotenv').config();

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

// Middlewares globales
app.use(cors());
app.use(morgan('combined'));
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true }));

// Middleware para archivos estáticos
app.use('/uploads', express.static('uploads'));

// Rutas
app.get('/', (req, res) => {
  res.json({
    mensaje: 'Marketplace Inteligente API',
    version: '1.0.0',
    documentacion: '/api-docs'
  });
});

// Manejo de errores global
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(err.status || 500).json({
    error: true,
    mensaje: err.message || 'Error interno del servidor'
  });
});

// Middleware para rutas no encontradas
app.use('*', (req, res) => {
  res.status(404).json({
    error: true,
    mensaje: 'Endpoint no encontrado'
  });
});
```

```
app.listen(PORT, () => {
  console.log('🚀 Servidor corriendo en puerto ${PORT}');
  console.log('📝 Logs: ${process.env.NODE_ENV || 'development'}');
});
```

Paso 3: Configuración de Base de Datos (src/config/database.js)

```
const mysql = require('mysql2/promise');

const dbConfig = {
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'root',
  password: process.env.DB_PASSWORD || '',
  database: process.env.DB_NAME || 'marketplace',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
};

// Pool de conexiones
const pool = mysql.createPool(dbConfig);

// Función para probar la conexión
const testConnection = async () => {
  try {
    const connection = await pool.getConnection();
    console.log('✅ Base de datos conectada correctamente');
    connection.release();
  } catch (error) {
    console.error('❌ Error al conectar con la base de datos:', error.message);
    process.exit(1);
  }
};

// Función para ejecutar queries
const executeQuery = async (query, params = []) => {
  try {
    const [rows] = await pool.execute(query, params);
    return rows;
  } catch (error) {
    console.error('Error en query:', error.message);
    throw error;
  }
};

module.exports = { pool, testConnection, executeQuery };
```

“El Hacker Invisible” – Juego de Inyección SQL

Antes de continuar en el proyecto hablemos de algo muy importante, las vulnerabilidades.

Revisemos esta linea de codigo, simplemente estaria pidiendo todos los usuarios donde el email cumpla sea el de la variable.

```
Const query = `SELECT * FROM users WHERE email = '${email}'`;
```

El anterior es un código “vulnerable”

Ejecuta la anterior consulta enviando el siguiente input: **' OR 1=1 --**

Preguntas:

- ¿Qué devuelve la consulta?
- ¿Qué pasaría en producción?

Luego comparan con:

```
pool.query('SELECT * FROM users WHERE email = ?', [email]);
```

Y analizas:

- ¿Por qué ahora no funciona el ataque?
- ¿Qué hizo realmente mysql2?

 Este ejercicio les cambia el chip sobre seguridad.

Paso 4: Modelo Usuario (src/models/Usuario.js)

```
const { executeQuery } = require('../config/database');

class Usuario {
  static async crear(datosUsuario) {
    const { nombre, email, password, rol = 'comprador' } = datosUsuario;

    const query = `
      INSERT INTO users (nombre, email, password, rol, fecha_registro)
      VALUES (?, ?, ?, ?, NOW())
    `;

    try {
      const resultado = await executeQuery(query, [nombre, email, password, rol]);
      return {
        id: resultado.insertId,
```

```

        nombre,
        email,
        rol
    };
} catch (error) {
    if (error.code === 'ER_DUP_ENTRY') {
        throw new Error('El email ya está registrado');
    }
    throw error;
}
}

static async buscarPorEmail(email) {
    const query = 'SELECT * FROM users WHERE email = ?';
    const usuarios = await executeQuery(query, [email]);
    return usuarios.length > 0 ? usuarios[0] : null;
}

static async buscarPorId(id) {
    const query = 'SELECT id, nombre, email, rol, fecha_registro FROM users WHERE id = ?';
    const usuarios = await executeQuery(query, [id]);
    return usuarios.length > 0 ? usuarios[0] : null;
}

static async obtenerTodos(filtros = {}) {
    let query = 'SELECT id, nombre, email, rol, fecha_registro FROM users WHERE 1=1';
    const params = [];

    // Filtro por rol
    if (filtros.rol) {
        query += ' AND rol = ?';
        params.push(filtros.rol);
    }

    // Búsqueda por nombre o email
    if (filtros.busqueda) {
        query += ' AND (nombre LIKE ? OR email LIKE ?)';
        params.push(`%${filtros.busqueda}%`, `%${filtros.busqueda}%`);
    }

    // Paginación
    const limite = parseInt(filtros.limite) || 10;
    const offset = (parseInt(filtros.pagina) - 1) * limite || 0;

    query += ' ORDER BY fecha_registro DESC LIMIT ? OFFSET ?';
    params.push(limite, offset);

    return await executeQuery(query, params);
}

static async actualizarPassword(id, nuevaPassword) {
    const query = 'UPDATE users SET password = ? WHERE id = ?';
    return await executeQuery(query, [nuevaPassword, id]);
}
}

module.exports = Usuario;

```

Paso 5: Controlador Usuario (src/controllers/usuarioController.js)

```
const bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator');
const Usuario = require('../models/Usuario');

class UsuarioController {
  static async crear(req, res, next) {
    try {
      // Verificar errores de validación
      const errores = validationResult(req);
      if (!errores.isEmpty()) {
        return res.status(400).json({
          error: true,
          mensaje: 'Datos inválidos',
          errores: errores.array()
        });
      }

      const { nombre, email, password, rol } = req.body;

      // Verificar si el usuario ya existe
      const usuarioExistente = await Usuario.buscarPorEmail(email);
      if (usuarioExistente) {
        return res.status(409).json({
          error: true,
          mensaje: 'El email ya está registrado'
        });
      }

      // Encriptar contraseña
      const passwordEncriptada = await bcrypt.hash(password, 12);

      // Crear usuario
      const nuevoUsuario = await Usuario.crear({
        nombre,
        email,
        password: passwordEncriptada,
        rol
      });

      res.status(201).json({
        error: false,
        mensaje: 'Usuario creado exitosamente',
        usuario: nuevoUsuario
      });
    } catch (error) {
      next(error);
    }
  }

  static async obtener(req, res, next) {
    try {
      const { id } = req.params;
      const usuario = await Usuario.buscarPorId(id);
```

```

if (!usuario) {
  return res.status(404).json({
    error: true,
    mensaje: 'Usuario no encontrado'
  });
}

res.json({
  error: false,
  usuario
});

} catch (error) {
  next(error);
}
}

static async listar(req, res, next) {
try {
  const filtros = {
    rol: req.query.rol,
    busqueda: req.query.q,
    pagina: req.query.pagina,
    limite: req.query.limite
  };

  const usuarios = await Usuario.obtenerTodos(filtros);

  res.json({
    error: false,
    usuarios,
    filtros_aplicados: filtros
  });

} catch (error) {
  next(error);
}
}
}

module.exports = UsuarioController;

```

Paso 6: Validaciones (src/middlewares/validaciones.js)

```

const { body, param, query } = require('express-validator');

const validacionCrearUsuario = [
  body('nombre')
    .trim()
    .isLength({ min: 2, max: 100 })
    .withMessage('El nombre debe tener entre 2 y 100 caracteres')
    .matches(/^[a-zA-ZáéíóúÁÉÍÓÚñÑ\s]+$/)
    .withMessage('El nombre solo puede contener letras y espacios'),
  body('email')
    .trim()
    .isEmail()

```

```

.withMessage('Email inválido')
.normalizeEmail(),

body('password')
.isLength({ min: 8 })
.withMessage('La contraseña debe tener al menos 8 caracteres')
.matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
.withMessage('La contraseña debe contener al menos una minúscula, una mayúscula y un número'),

body('rol')
.optional()
.isIn(['comprador', 'vendedor', 'admin'])
.withMessage('Rol inválido')
];

const validacionParametroId = [
param('id')
.isInt({ min: 1 })
.withMessage('ID debe ser un número entero positivo')
];

module.exports = {
validacionCrearUsuario,
validacionParametroId
};

```

Paso 7: Rutas Usuario (src/routes/usuarioRoutes.js)

```

const express = require('express');
const UsuarioController = require('../controllers/usuarioController');
const {
  validacionCrearUsuario,
  validacionParametroId
} = require('../middlewares/validaciones');

const router = express.Router();

// POST /api/usuarios - Crear usuario
router.post('/', validacionCrearUsuario, UsuarioController.crear);

// GET /api/usuarios - Listar usuarios
router.get('/', UsuarioController.listar);

// GET /api/usuarios/:id - Obtener usuario por ID
router.get('/:id', validacionParametroId, UsuarioController.obtener);

module.exports = router;

```

Paso 8: Script SQL para Crear Tablas

```

-- Crear base de datos
CREATE DATABASE IF NOT EXISTS marketplace CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE marketplace;

-- Tabla usuarios
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(100) NOT NULL,

```

```

email VARCHAR(150) UNIQUE NOT NULL,
password VARCHAR(255) NOT NULL,
rol ENUM('comprador', 'vendedor', 'admin') DEFAULT 'comprador',
fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
INDEX idx_email (email),
INDEX idx_rol (rol)
);

-- Tabla categorías
CREATE TABLE categories (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    descripcion TEXT,
    imagen_icono VARCHAR(255),
    fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Tabla productos
CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    vendedor_id INT NOT NULL,
    categoria_id INT NOT NULL,
    nombre VARCHAR(200) NOT NULL,
    descripcion TEXT,
    precio DECIMAL(10, 2) NOT NULL,
    stock INT DEFAULT 0,
    imagen_url VARCHAR(255),
    fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (vendedor_id) REFERENCES users(id) ON DELETE CASCADE,
    FOREIGN KEY (categoria_id) REFERENCES categories(id),
    INDEX idx_vendedor (vendedor_id),
    INDEX idx_categoria (categoria_id),
    INDEX idx_precio (precio)
);

-- Tabla órdenes
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    comprador_id INT NOT NULL,
    total DECIMAL(10, 2) NOT NULL,
    estado ENUM('pendiente', 'confirmada', 'enviada', 'entregada', 'cancelada') DEFAULT 'pendiente',
    direccion_envio TEXT,
    notas TEXT,
    fecha_orden TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (comprador_id) REFERENCES users(id),
    INDEX idx_comprador (comprador_id),
    INDEX idx_estado (estado),
    INDEX idx_fecha (fecha_orden)
);

-- Tabla detalles de orden
CREATE TABLE order_details (
    id INT PRIMARY KEY AUTO_INCREMENT,
    orden_id INT NOT NULL,
    producto_id INT NOT NULL,
    cantidad INT NOT NULL,
    precio_unitario DECIMAL(10, 2) NOT NULL,

```

```
subtotal DECIMAL(10, 2) NOT NULL,  
FOREIGN KEY (orden_id) REFERENCES orders(id) ON DELETE CASCADE,  
FOREIGN KEY (producto_id) REFERENCES products(id),  
INDEX idx_orden (orden_id),  
INDEX idx_producto (producto_id)  
);
```

⌚ Desafío Autónomo

Ahora implementa lo siguiente:

Tarea 1: Endpoints PUT y DELETE para usuarios

Implementa la funcionalidad completa para:

- Actualizar usuario existente (PUT </api/usuarios/:id>)
- Eliminar usuario (DELETE </api/usuarios/:id>)

Preguntas de Diseño:

1. ¿Qué campos permitirías actualizar? ¿Por qué no todos?
2. ¿Eliminación física o lógica? ¿Por qué?
3. ¿Qué pasa con los productos del vendedor eliminado?

Tarea 2: Mejoras en filtros y paginación

Mejora el endpoint GET </api/usuarios> con:

- Filtro por fecha de registro (desde/hasta)
- Ordenamiento por diferentes campos
- Metadatos de paginación completos

Módulo 2: Autenticación y Seguridad

🧐 Preguntas de Análisis Inicial

1. Flujo de Autenticación:

- o ¿Cuál es la diferencia entre autenticación y autorización?
- o ¿Cómo manejarías la expiración de tokens?

2. Seguridad de Contraseñas:

- o ¿Qué es el "salt" y por qué es importante?
- o ¿Cómo balancear seguridad vs rendimiento en el hash?

💻 Implementación Práctica

Paso 1: Utilidades JWT (<src/utils/jwt.js>)

```
const jwt = require('jsonwebtoken');
```

```

const JWT_SECRET = process.env.JWT_SECRET || 'fallback-secret-key';
const JWT_EXPIRE = process.env.JWT_EXPIRE || '7d';

const generarToken = (usuario) => {
  const payload = {
    id: usuario.id,
    email: usuario.email,
    rol: usuario.rol,
    iat: Math.floor(Date.now() / 1000)
  };

  return jwt.sign(payload, JWT_SECRET, {
    expiresIn: JWT_EXPIRE,
    issuer: 'marketplace-api',
    audience: 'marketplace-users'
  });
};

const verificarToken = (token) => {
  try {
    return jwt.verify(token, JWT_SECRET, {
      issuer: 'marketplace-api',
      audience: 'marketplace-users'
    });
  } catch (error) {
    throw new Error('Token inválido o expirado');
  }
};

const extraerTokenDeHeader = (authHeader) => {
  if (!authHeader) {
    throw new Error('Token no proporcionado');
  }

  if (!authHeader.startsWith('Bearer ')) {
    throw new Error('Formato de token inválido. Usar: Bearer <token>');
  }

  return authHeader.substring(7);
};

module.exports = {
  generarToken,
  verificarToken,
  extraerTokenDeHeader
};

```

Paso 2: Middleware de Autenticación (src/middlewares/auth.js)

```

const { verificarToken, extraerTokenDeHeader } = require('../utils/jwt');
const Usuario = require('../models/Usuario');

const autenticar = async (req, res, next) => {
  try {
    const authHeader = req.headers.authorization;
    const token = extraerTokenDeHeader(authHeader);
  
```

```

const payload = verificarToken(token);

const usuario = await Usuario.buscarPorId(payload.id);
if (!usuario) {
  return res.status(401).json({
    error: true,
    mensaje: 'Usuario no válido'
  });
}

req.usuario = usuario;
next();

} catch (error) {
  return res.status(401).json({
    error: true,
    mensaje: error.message
  });
}
};

const requiereRol = (rolesPermitidos) => {
return (req, res, next) => {
  const roles = Array.isArray(rolesPermitidos) ? rolesPermitidos : [rolesPermitidos];

  if (!roles.includes(req.usuario.rol)) {
    return res.status(403).json({
      error: true,
      mensaje: 'No tienes permisos para realizar esta acción',
      rolRequerido: roles,
      tuRol: req.usuario.rol
    });
  }

  next();
};
};

module.exports = {
  autenticar,
  requiereRol
};

```

Paso 3: Controlador de Autenticación (src/controllers/authController.js)

```

const bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator');
const Usuario = require('../models/Usuario');
const { generarToken } = require('../utils/jwt');

class AuthController {
  static async registro(req, res, next) {
    try {
      const errores = validationResult(req);
      if (!errores.isEmpty()) {
        return res.status(400).json({
          error: true,

```

```
        mensaje: 'Datos de registro inválidos',
        errores: errores.array()
    });
}

const { nombre, email, password, rol } = req.body;

const usuarioExistente = await Usuario.buscarPorEmail(email);
if (usuarioExistente) {
    return res.status(409).json({
        error: true,
        mensaje: 'El email ya está registrado'
    });
}

const passwordHash = await bcrypt.hash(password, 12);

const nuevoUsuario = await Usuario.crear({
    nombre,
    email,
    password: passwordHash,
    rol: rol || 'comprador'
});

const token = generarToken(nuevoUsuario);

res.status(201).json({
    error: false,
    mensaje: 'Usuario registrado exitosamente',
    usuario: nuevoUsuario,
    token
});

} catch (error) {
    next(error);
}
}

static async login(req, res, next) {
try {
    const errores = validationResult(req);
    if (!errores.isEmpty()) {
        return res.status(400).json({
            error: true,
            mensaje: 'Credenciales inválidas',
            errores: errores.array()
        });
    }
}

const { email, password } = req.body;

const usuario = await Usuario.buscarPorEmail(email);
if (!usuario) {
    return res.status(401).json({
        error: true,
        mensaje: 'Credenciales incorrectas'
    });
}
```

```
}

const passwordValida = await bcrypt.compare(password, usuario.password);
if (!passwordValida) {
  return res.status(401).json({
    error: true,
    mensaje: 'Credenciales incorrectas'
  });
}

const usuarioLimpio = {
  id: usuario.id,
  nombre: usuario.nombre,
  email: usuario.email,
  rol: usuario.rol
};

const token = generarToken(usuarioLimpio);

res.json({
  error: false,
  mensaje: 'Inicio de sesión exitoso',
  usuario: usuarioLimpio,
  token
});

} catch (error) {
  next(error);
}
}

static async perfil(req, res, next) {
try {
  const usuario = await Usuario.buscarPorId(req.usuario.id);

  if (!usuario) {
    return res.status(404).json({
      error: true,
      mensaje: 'Usuario no encontrado'
    });
  }

  res.json({
    error: false,
    usuario
  });

} catch (error) {
  next(error);
}
}
}

module.exports = AuthController;
```

"Anatomía de un Token" – Autopsia del JWT

Ya implementamos el JWT, antes de avanzar en el proyecto, haz esto:

Actividad práctica

1. Decodifica este token en : <https://token.dev/>

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJyI6IjEyMzQ1Njc4OSIsIm5hbWUiOiJMYSBtdWplciBtYXJhdmlsbGEiLCJyb2wiOiJ1c3VhcmlvIiwiZXN0dWRpYSI6IkFEU08gQ0FUISJ9.QrR7nArmYTAJbqAfmQkH0yCJCHaOQq1SE-nfMVt68Bs
```

2. decodifiquenlo manualmente en:

1. header
2. payload
3. signature

ahora respondan:

- ¿Por qué cualquiera puede leer el payload?
 - ¿Dónde está realmente la seguridad?
 - ¿Qué pasa si modiflico el rol a "admin"?
3. Hagamos otra prueba, generen un token desde la aplicación, con el login que acaban de crear, vamos a simular un ataque:
 - Cambian el payload del token que acaban de generar
 - Intentan usar el token modificado en una petición cualquiera

¿Qué paso?

Continuemos con el proyecto ahora con un Desafío Autónomo

Implementa las siguientes funcionalidades de seguridad:

Tarea 1: Sistema de Cambio de Contraseña

- Endpoint para cambiar contraseña (requiere contraseña actual)
- Validaciones de seguridad robustas
- Invalidar tokens existentes después del cambio

Tarea 2: Middleware de Rate Limiting

- Protección contra ataques de fuerza bruta
- Límites diferentes por endpoint
- Bloqueo temporal de IPs sospechosas

Módulo 3: Gestión de Productos y Archivos

Preguntas de Análisis Inicial

1. Gestión de Archivos:

- ¿Local storage vs cloud storage? Ventajas y desventajas
- ¿Cómo validarías el tipo y tamaño de archivos?
- ¿Qué estrategia seguirías para nombres únicos de archivos?

2. Relaciones entre Modelos:

- ¿Cómo manejarías la relación Producto-Usuario-Categoría?
- ¿Qué pasa con los productos si se elimina un vendedor?

Implementación Práctica

Paso 1: Configuración de Multer (src/config/multer.js)

```
const multer = require('multer');
const path = require('path');
const fs = require('fs');

const crearDirectorios = () => {
  const directorios = [
    'uploads',
    'uploads/productos',
    'uploads/usuarios'
  ];

  directorios.forEach(dir => {
    if (!fs.existsSync(dir)) {
      fs.mkdirSync(dir, { recursive: true });
    }
  });
};

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    crearDirectorios();

    let carpeta = 'uploads/productos';
    if (file.fieldname === 'avatar_usuario') {
      carpeta = 'uploads/usuarios';
    }

    cb(null, carpeta);
  },
  filename: (req, file, cb) => {
    const extension = path.extname(file.originalname);
    const timestamp = Date.now();
    const random = Math.round(Math.random() * 1E9);
    cb(null, `${timestamp}${random}${extension}`);
  }
});
```

```

const nombreUnico = `${timestamp}-${random}${extension}`;

cb(null, nombreUnico);
}

});

const fileFilter = (req, file, cb) => {
const tiposPermitidos = [
'image/jpeg',
'image/jpg',
'image/png',
'image/webp'
];

if (tiposPermitidos.includes(file.mimetype)) {
cb(null, true);
} else {
cb(new Error('Tipo de archivo no permitido. Solo se aceptan: JPG, PNG, WEBP'), false);
}
};

const upload = multer({
storage,
fileFilter,
limits: {
fileSize: 5 * 1024 * 1024, // 5MB
files: 5
}
});

module.exports = {
subirImagenProducto: upload.single('imagen_producto'),
subirAvatarUsuario: upload.single('avatar_usuario')
};

```

Paso 2: Modelo Producto (src/models/Producto.js)

```

const { executeQuery } = require('../config/database');

class Producto {
static async crear(datosProducto) {
const {
vendedor_id,
categoria_id,
nombre,
descripcion,
precio,
stock,
imagen_url
} = datosProducto;

const query = `
INSERT INTO products
(vendedor_id, categoria_id, nombre, descripcion, precio, stock, imagen_url, fecha_creacion)
VALUES (?, ?, ?, ?, ?, ?, ?, NOW())
`;

```

```

try {
  const resultado = await executeQuery(query, [
    vendedor_id, categoria_id, nombre, descripcion, precio, stock, imagen_url
  ]);

  return {
    id: resultado.insertId,
    ...datosProducto
  };
} catch (error) {
  throw error;
}
}

static async buscarPorId(id) {
  const query = `
    SELECT
      p.*,
      u.nombre as vendedor_nombre,
      u.email as vendedor_email,
      c.nombre as categoria_nombre
    FROM products p
    LEFT JOIN users u ON p.vendedor_id = u.id
    LEFT JOIN categories c ON p.categoria_id = c.id
    WHERE p.id = ?
  `;

  const productos = await executeQuery(query, [id]);
  return productos.length > 0 ? productos[0] : null;
}

static async obtenerConFiltros(filtros = {}) {
  let query = `
    SELECT
      p.id, p.nombre, p.descripcion, p.precio, p.stock,
      p.imagen_url, p.fecha_creacion,
      u.nombre as vendedor_nombre,
      c.nombre as categoria_nombre
    FROM products p
    LEFT JOIN users u ON p.vendedor_id = u.id
    LEFT JOIN categories c ON p.categoria_id = c.id
    WHERE 1=1
  `;

  const params = [];

  if (filtros.categoria_id) {
    query += ' AND p.categoria_id = ?';
    params.push(filtros.categoria_id);
  }

  if (filtros.vendedor_id) {
    query += ' AND p.vendedor_id = ?';
    params.push(filtros.vendedor_id);
  }

  if (filtros.precio_min) {

```

```

query += ' AND p.precio >= ?';
params.push(filtros.precio_min);
}

if (filtros.precio_max) {
  query += ' AND p.precio <= ?';
  params.push(filtros.precio_max);
}

if (filtros.en_stock === 'true') {
  query += ' AND p.stock > 0';
}

if (filtros.busqueda) {
  query += ' AND (p.nombre LIKE ? OR p.descripcion LIKE ?)';
  const busquedaParam = `%${filtros.busqueda}%`;
  params.push(busquedaParam, busquedaParam);
}

query += ' ORDER BY p.fecha_creacion DESC';

const limite = Math.min(parseInt(filtros.limite) || 10, 50);
const offset = (parseInt(filtros.pagina) - 1) * limite || 0;

query += ' LIMIT ? OFFSET ?';
params.push(limite, offset);

return await executeQuery(query, params);
}

static async actualizar(id, datosActualizados) {
  const camposPermitidos = ['nombre', 'descripcion', 'precio', 'stock', 'categoria_id', 'imagen_url'];
  const campos = [];
  const valores = [];

  Object.keys(datosActualizados).forEach(campo =>{
    if (camposPermitidos.includes(campo) && datosActualizados[campo] !== undefined) {
      campos.push(`${campo} = ?`);
      valores.push(datosActualizados[campo]);
    }
  });

  if (campos.length === 0) {
    throw new Error('No hay campos válidos para actualizar');
  }

  valores.push(id);

  const query = `UPDATE products SET ${campos.join(', ')} WHERE id = ?`;

  const resultado = await executeQuery(query, valores);

  if (resultado.affectedRows === 0) {
    throw new Error('Producto no encontrado');
  }

  return await this.buscarPorId(id);
}

```

```

}

static async eliminar(id) {
  const query = 'DELETE FROM products WHERE id = ?';
  const resultado = await executeQuery(query, [id]);

  if (resultado.affectedRows === 0) {
    throw new Error('Producto no encontrado');
  }

  return { eliminado: true };
}
}

module.exports = Producto;

```

🎯 Desafío Autónomo

Implementa el sistema completo de Categorías:

Tarea 1: Modelo y Controlador Categoría

Siguiendo el patrón de Producto, implementa:

- CRUD completo de categorías
- Validaciones apropiadas
- Gestión de imágenes de iconos

Tarea 2: Múltiples Imágenes por Producto

Extiende el sistema para:

- Permitir múltiples imágenes por producto
- Galería de imágenes con imagen principal
- Optimización automática de imágenes

Módulo 4: Filtros Avanzados y Validaciones

💻 Implementación Práctica

Utilidades de Filtros (src/utils/filtros.js)

```

const parsearLista = (valor, tipo = 'string') => {
  if (!valor) return null;

  const lista = valor.split(',').map(item => item.trim()).filter(Boolean);

  switch (tipo) {
    case 'int':
      return lista.map(item => parseInt(item)).filter(num => !isNaN(num));
    case 'float':
      return lista.map(item => parseFloat(item)).filter(num => !isNaN(num));
    default:
  }
}

```

```

        return lista;
    }
};

const parsearOrdenamiento = (orden) => {
    if (!orden) return [];

    const camposPermitidos = ['precio', 'fecha_creacion', 'nombre', 'stock', 'id'];

    return orden.split(',')
        .map(item => {
            const [campo, direccion = 'desc'] = item.trim().split(':');
            return {
                campo: campo.toLowerCase(),
                direccion: direccion.toUpperCase()
            };
        })
        .filter(item =>
            camposPermitidos.includes(item.campo) &&
            ['ASC', 'DESC'].includes(item.direccion)
        );
};

const construirFiltrosSQL = (filtros) => {
    const condiciones = [];
    const parametros = [];

    if (filtros.categorias && filtros.categorias.length > 0) {
        const placeholders = filtros.categorias.map(() => '?').join(',');
        condiciones.push(`p.categoria_id IN (${placeholders})`);
        parametros.push(...filtros.categorias);
    }

    if (filtros.precio) {
        if (filtros.precio.min !== undefined) {
            condiciones.push('p.precio >= ?');
            parametros.push(filtros.precio.min);
        }
        if (filtros.precio.max !== undefined) {
            condiciones.push('p.precio <= ?');
            parametros.push(filtros.precio.max);
        }
    }

    if (filtros.busqueda) {
        condiciones.push('(p.nombre LIKE ? OR p.descripcion LIKE ?)');
        parametros.push(`%${filtros.busqueda}%`, `%${filtros.busqueda}%`);
    }

    return {
        whereClause: condiciones.length > 0 ? `AND ${condiciones.join(' AND ')}` : '',
        parametros
    };
};

module.exports = {
    parsearLista,

```

```
parsearOrdenamiento,  
construirFiltrosSQL  
};
```

🎯 Desafío Autónomo

Implementa filtros avanzados para todos los modelos:

Tarea 1: Sistema de Headers HTTP

- Procesamiento de filtros via headers
- Headers de paginación en respuestas
- Metadatos completos en respuestas

Tarea 2: Validaciones de Reglas de Negocio

- Stock disponible antes de crear órdenes
- Validación de precios coherentes
- Verificación de permisos granulares

Módulo 5: Órdenes e Integración IA

💻 Implementación Práctica

Cliente Gemini AI (src/config/gemini.js)

```
const axios = require('axios');

const GEMINI_API_KEY = process.env.GEMINI_API_KEY;
const GEMINI_BASE_URL = 'https://generativelanguage.googleapis.com/v1beta';

class GeminiClient {
  constructor() {
    this.client = axios.create({
      baseURL: GEMINI_BASE_URL,
      timeout: 30000,
      headers: {
        'Content-Type': 'application/json'
      }
    });
  }

  async generarContenido(prompt, opciones = {}) {
    if (!GEMINI_API_KEY) {
      throw new Error('API Key de Gemini no configurado');
    }

    try {
      const payload = {
        contents: [
          {
            parts: [{ text: prompt }]
        }
      ]
    }
  }
}
```

```

        },
        ],
        generationConfig: {
            temperature: opciones.temperatura || 0.7,
            topK: opciones.topK || 40,
            topP: opciones.topP || 0.95,
            maxOutputTokens: opciones.maxTokens || 1024
        }
    };

const response = await this.client.post(
    `/models/gemini-pro:generateContent?key=${GEMINI_API_KEY}`,
    payload
);

if (!response.data.candidates || response.data.candidates.length === 0) {
    throw new Error('No se recibió respuesta válida de Gemini');
}

const contenido = response.data.candidates[0].content.parts[0].text;

return {
    contenido,
    metadata: {
        tokens_utilizados: response.data.usageMetadata?.totalTokenCount || 0
    }
};

} catch (error) {
    console.error(`❌ Error en Gemini: ${error.message}`);
    if (error.response?.status === 429) {
        throw new Error('Límite de tasa de Gemini excedido. Intenta más tarde.');
    }

    throw new Error(`Error de Gemini: ${error.message}`);
}
}

async generarDescripcionProducto(nombre, categoria, caracteristicas = "") {
    const prompt = `

Eres un copywriter experto en e-commerce. Genera una descripción atractiva y profesional para este producto:

PRODUCTO: ${nombre}
CATEGORÍA: ${categoria}
CARACTERÍSTICAS: ${caracteristicas}`

    INSTRUCCIONES:
    - Máximo 200 palabras
    - Destaca beneficios, no solo características
    - Usa un tono persuasivo pero profesional
    - Estructura en párrafos cortos
}

```

Responde SOLO con la descripción, sin explicaciones adicionales.

`;

```
const resultado = await this.generarContenido(prompt, {
  temperatura: 0.8,
  maxTokens: 300
});

return resultado.contenido.trim();
}
```

```
async sugerirCategorias(nombreProducto, descripcion) {
  const prompt = `
```

Analiza este producto y sugiere las 3 categorías más apropiadas:

PRODUCTO: \${nombreProducto}

DESCRIPCIÓN: \${descripcion}

CATEGORÍAS DISPONIBLES:

- Electrónicos
- Ropa y Accesorios
- Hogar y Jardín
- Deportes y Aire Libre
- Salud y Belleza
- Libros y Medios
- Juguetes y Juegos

Responde con exactamente 3 categorías de la lista, separadas por comas, sin explicaciones.

`;

```
const resultado = await this.generarContenido(prompt, {
  temperatura: 0.3,
  maxTokens: 100
});
```

```
return resultado.contenido.trim().split(',').map(c => c.trim());
}
```

```
async analizarPatronCompras(ordenes) {
  const resumen = ordenes.map(orden => ({
    total: orden.total,
    fecha: orden.fecha_orden,
    productos: orden.productos?.length || 0
  }));
}
```

const prompt = `

Analiza estos patrones de compra y genera insights:

DATOS: \${JSON.stringify(resumen)}

GENERA:

1. Patrón de frecuencia de compra
2. Promedio de gasto por orden
3. Tendencias temporales
4. Recomendaciones para el usuario

Responde en formato JSON con esta estructura:

```
{  
  "frecuencia": "descripción",  
  "promedio_gasto": número,
```

```

    "tendencias": "descripción",
    "recomendaciones": ["recomendación1", "recomendación2"]
}
`;

try {
  const resultado = await this.generarContenido(prompt, {
    temperatura: 0.5,
    maxTokens: 400
  });

  return JSON.parse(resultado.contenido);
} catch (error) {
  console.error('Error parseando análisis de patrones:', error);
  return {
    frecuencia: 'No se pudo analizar',
    promedio_gasto: 0,
    tendencias: 'Datos insuficientes',
    recomendaciones: []
  };
}
}

module.exports = new GeminiClient();

```

Modelo Orden con Transacciones (src/models/Orden.js)

```

const { executeQuery, pool } = require('../config/database');

class Orden {
  static async crear(datosOrden, detallesProductos) {
    const connection = await pool.getConnection();

    try {
      await connection.beginTransaction();

      const { comprador_id, total, direccion_envio, notas } = datosOrden;

      // Crear la orden principal
      const queryOrden = `
        INSERT INTO orders (comprador_id, total, estado, direccion_envio, notas, fecha_orden)
        VALUES (?, ?, 'pendiente', ?, ?, NOW())
      `;

      const [resultadoOrden] = await connection.execute(queryOrden, [
        comprador_id, total, direccion_envio, notas
      ]);

      const orderId = resultadoOrden.insertId;

      // Validar stock y crear detalles
      for (const detalle of detallesProductos) {
        const { producto_id, cantidad, precio_unitario } = detalle;

        // Verificar stock actual con bloqueo
        const [productos] = await connection.execute(

```

```

'SELECT stock, precio FROM products WHERE id = ? FOR UPDATE',
[producto_id]
);

if (productos.length === 0) {
  throw new Error(`Producto ${producto_id} no encontrado`);
}

const producto = productos[0];

if (producto.stock < cantidad) {
  throw new Error(
    `Stock insuficiente para producto ${producto_id}. ` +
    `Disponible: ${producto.stock}, Solicitado: ${cantidad}`
  );
}

// Verificar precio
const diferenciaPrecio = Math.abs(producto.precio - precio_unitario) / producto.precio;
if (diferenciaPrecio > 0.05) {
  throw new Error(`El precio del producto ${producto_id} ha cambiado. Actualiza tu carrito.`);
}

// Reducir stock
await connection.execute(
  'UPDATE products SET stock = stock - ? WHERE id = ?',
  [cantidad, producto_id]
);

// Crear detalle de orden
await connection.execute(`
  INSERT INTO order_details (orden_id, producto_id, cantidad, precio_unitario, subtotal)
  VALUES (?, ?, ?, ?, ?)
  `, [ordenId, producto_id, cantidad, precio_unitario, cantidad * precio_unitario]);
}

await connection.commit();
return await this.buscarPorId(ordenId);

} catch (error) {
  await connection.rollback();
  throw error;
} finally {
  connection.release();
}
}

static async buscarPorId(id) {
  const queryOrden = `
    SELECT
      o.*,
      u.nombre as comprador_nombre,
      u.email as comprador_email
    FROM orders o
    LEFT JOIN users u ON o.comprador_id = u.id
    WHERE o.id = ?
  `;
}

```

```

const ordenes = await executeQuery(queryOrden, [id]);
if (ordenes.length === 0) return null;

const orden = ordenes[0];

// Obtener detalles de la orden
const queryDetalles = `
SELECT
od.*,
p.nombre as producto_nombre,
p.imagen_url,
u.nombre as vendedor_nombre
FROM order_details od
LEFT JOIN products p ON od.producto_id = p.id
LEFT JOIN users u ON p.vendedor_id = u.id
WHERE od.order_id = ?
`;

const detalles = await executeQuery(queryDetalles, [id]);
orden.detalles = detalles;

return orden;
}

static async actualizarEstado(id, nuevoEstado, notas = null) {
const estadosValidos = ['pendiente', 'confirmada', 'enviada', 'entregada', 'cancelada'];

if (!estadosValidos.includes(nuevoEstado)) {
throw new Error('Estado inválido');
}

let query = 'UPDATE orders SET estado = ?';
let params = [nuevoEstado];

if (notas) {
query += ', notas = ?';
params.push(notas);
}

query += ' WHERE id = ?';
params.push(id);

const resultado = await executeQuery(query, params);

if (resultado.affectedRows === 0) {
throw new Error('Orden no encontrada');
}

return await this.buscarPorId(id);
}

module.exports = Orden;

```

Desafío Autónomo

Implementa funcionalidades avanzadas con IA:

Tarea 1: Sistema de Recomendaciones

- Análisis de patrones de compra con IA
- Sugerencias de productos personalizadas
- Sistema de puntuación de recomendaciones

Tarea 2: Moderación de Contenido

- Revisión automática de descripciones de productos
- Detección de contenido inapropiado
- Sistema de aprobación/rechazo automático

Módulo 6: Documentación API con Swagger

Implementación Práctica

Configuración Swagger (src/config/swagger.js)

```
const swaggerJSDoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Marketplace Inteligente API',
      version: '1.0.0',
      description: 'API completa para marketplace con integración de IA',
      contact: {
        name: 'Equipo de Desarrollo',
        email: 'dev@marketplace.com'
      }
    },
    servers: [
      {
        url: process.env.BASE_URL || 'http://localhost:3000',
        description: 'Servidor de desarrollo'
      }
    ],
    components: {
      securitySchemes: {
        BearerAuth: {
          type: 'http',
          scheme: 'bearer',
          bearerFormat: 'JWT'
        }
      },
      schemas: {
        ...
      }
    }
};
```

```
Usuario: {
  type: 'object',
  required: ['nombre', 'email', 'password'],
  properties: {
    id: {
      type: 'integer',
      description: 'ID único del usuario',
      example: 1
    },
    nombre: {
      type: 'string',
      description: 'Nombre completo del usuario',
      example: 'Juan Pérez'
    },
    email: {
      type: 'string',
      format: 'email',
      description: 'Email único del usuario',
      example: 'juan@ejemplo.com'
    },
    rol: {
      type: 'string',
      enum: ['comprador', 'vendedor', 'admin'],
      description: 'Rol del usuario en el sistema',
      example: 'comprador'
    },
    fecha_registro: {
      type: 'string',
      format: 'date-time',
      description: 'Fecha de registro del usuario'
    }
  }
},
Producto: {
  type: 'object',
  required: ['nombre', 'precio', 'categoria_id'],
  properties: {
    id: {
      type: 'integer',
      description: 'ID único del producto'
    },
    nombre: {
      type: 'string',
      description: 'Nombre del producto',
      example: 'iPhone 15 Pro'
    },
    descripcion: {
      type: 'string',
      description: 'Descripción detallada del producto'
    },
    precio: {
      type: 'number',
      format: 'float',
      description: 'Precio del producto',
      example: 999.99
    },
    stock: {
```

```

        type: 'integer',
        description: 'Cantidad disponible',
        example: 50
    },
    imagen_url: {
        type: 'string',
        description: 'URL de la imagen del producto'
    },
    categoria_id: {
        type: 'integer',
        description: 'ID de la categoría'
    },
    vendedor_id: {
        type: 'integer',
        description: 'ID del vendedor'
    }
},
Error: {
    type: 'object',
    properties: {
        error: {
            type: 'boolean',
            example: true
        },
        mensaje: {
            type: 'string',
            example: 'Mensaje de error descriptivo'
        },
        errores: {
            type: 'array',
            items: {
                type: 'object',
                properties: {
                    campo: { type: 'string' },
                    mensaje: { type: 'string' }
                }
            }
        }
    }
},
apis: ['./src/routes/*.js'] // Rutas donde están las anotaciones Swagger
};

const specs = swaggerJSDoc(options);

const swaggerOptions = {
    explorer: true,
    customCss: '.swagger-ui .topbar { display: none }',
    customSiteTitle: 'Marketplace API Docs'
};

module.exports = {
    specs,

```

```
    swaggerUi,  
    swaggerOptions  
};
```

Documentación en Rutas (ejemplo src/routes/usuarioRoutes.js)

```
const express = require('express');  
const UsuarioController = require('../controllers/usuarioController');  
const { autenticar, requiereRol } = require('../middlewares/auth');
```

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

```
/**  
 * @swagger  
 * /api/usuarios:  
 *   post:  
 *     summary: Crear un nuevo usuario  
 *     tags: [Usuarios]  
 *     requestBody:  
 *       required: true  
 *       content:  
 *         application/json:  
 *           schema:  
 *             type: object  
 *             required:  
 *               - nombre  
 *               - email  
 *               - password  
 *             properties:  
 *               nombre:  
 *                 type: string  
 *                 example: Juan Pérez  
 *               email:  
 *                 type: string  
 *                 format: email  
 *                 example: juan@ejemplo.com  
 *               password:  
 *                 type: string  
 *                 minLength: 8  
 *                 example: MiPassword123!  
 *               rol:  
 *                 type: string  
 *                 enum: [comprador, vendedor]  
 *                 default: comprador  
 *   responses:  
 *     201:  
 *       description: Usuario creado exitosamente  
 *       content:  
 *         application/json:  
 *           schema:  
 *             type: object  
 *             properties:  
 *               error:  
 *                 type: boolean  
 *                 example: false  
 *               mensaje:  
 *                 type: string
```

```

*      example: Usuario creado exitosamente
*      usuario:
*      $ref: '#/components/schemas/Usuario'
* 400:
*      description: Datos inválidos
*      content:
*          application/json:
*              schema:
*                  $ref: '#/components/schemas/Error'
* 409:
*      description: Email ya registrado
*/
router.post('/', UsuarioController.create);

/**
 * @swagger
 * /api/usuarios:
 *  get:
 *      summary: Listar usuarios (solo admin)
 *      tags: [Usuarios]
 *      security:
 *          - BearerAuth: []
 *      parameters:
 *          - in: query
 *              name: rol
 *              schema:
 *                  type: string
 *                  enum: [comprador, vendedor, admin]
 *              description: Filtrar por rol
 *          - in: query
 *              name: q
 *              schema:
 *                  type: string
 *              description: Búsqueda por nombre o email
 *          - in: query
 *              name: pagina
 *              schema:
 *                  type: integer
 *                  minimum: 1
 *                  default: 1
 *              description: Número de página
 *          - in: query
 *              name: limite
 *              schema:
 *                  type: integer
 *                  minimum: 1
 *                  maximum: 100
 *                  default: 10
 *              description: Usuarios por página
 *      responses:
 *          200:
 *              description: Lista de usuarios
 *              content:
 *                  application/json:
 *                      schema:
 *                          type: object
 *                          properties:

```

```
*      error:
*        type: boolean
*        example: false
*      usuarios:
*        type: array
*        items:
*          $ref: '#/components/schemas/Usuario'
*      filtros_aplicados:
*        type: object
*
*    401:
*      description: Token no válido
*    403:
*      description: Sin permisos de administrador
*/
router.get('/', autenticar, requiereRol('admin'), UsuarioController.listar);

module.exports = router;
```

⌚ Desafío Autónomo

Documenta completamente tu API:

Tarea 1: Documentación Completa

- Documenta todos los endpoints de productos, órdenes y categorías
- Incluye ejemplos de respuesta reales
- Agrega códigos de error específicos

Tarea 2: Ejemplos Interactivos

- Configura ejemplos que funcionen en Swagger UI
 - Incluye casos de prueba para diferentes escenarios
 - Documenta los endpoints de IA
-

5. “API Battle Royale” – Swagger vs Postman vs Axios

para comprender aun mas el tema de documentación, haremos el siguiente ejercicio

Actividad:

Cada grupo debe:

1. Compartir sus endpoints con otro grupo.
2. Esos endpoints, deben estar ya bien documentados con Swagger.
3. El otro grupo debe consumirlo SOLO usando la documentación.
4. El grupo que consume el endpoint debe anexar los resultados a su informe y compartirlo con el grupo de origen.

Que le lleva el informe:

- ¿La documentación está mal? Exactamente donde.
- ¿El contrato no está claro?
- ¿Falta un status code?

Y todo lo que crean que haga falta...

Si la API de sus compañeros pasa todas las pruebas, deben entrar a [HTTP.cat](#) y elegir el **Gato 200 (OK)** o el **Gato 201 (Created)** como su medalla de graduación y enviársela por whatsapp, para que adjunten el pantallazo.



💡 deben aprender que Swagger no es decoración, es contrato.

Módulo 7: Proyecto Final - Categorías

🎯 Desafío Final – Parte A - Completar API

Este es tu momento de demostrar todo lo aprendido. Implementa el sistema completo de Categorías aplicando TODOS los conceptos:

Requerimientos Obligatorios:

1. CRUD Completo con Validaciones

- Crear, leer, actualizar, eliminar categorías
- Validaciones robustas (nombre único, descripción, imagen)
- Manejo de errores específicos

2. Autenticación y Autorización

- Solo admin puede crear/modificar/eliminar categorías

- Cualquier usuario puede ver categorías
- Middleware de permisos apropiado

3. Gestión de Archivos

- Subida de imagen de ícono para cada categoría
- Validación de tipo y tamaño de archivo
- Limpieza de archivos al eliminar categorías

4. Filtros y Búsqueda

- Búsqueda por nombre de categoría
- Filtros por fecha de creación
- Ordenamiento múltiple
- Paginación completa con metadatos

5. Integración con IA

- Generar descripciones automáticas de categorías
- Sugerir categorías similares o relacionadas
- Analizar productos por categoría

6. Relaciones con Productos

- Endpoint para obtener productos de una categoría
- Estadísticas de productos por categoría
- Manejar categorías con productos asociados

7. Documentación Swagger

- Todos los endpoints documentados
- Ejemplos de peticiones y respuestas
- Esquemas de datos detallados

Estructura Esperada:

```

src/
└── models/Categoría.js
└── controllers/categoríaController.js
└── routes/categoríaRoutes.js
└── middlewares/validaciones.js (actualizar)
└── tests/ (opcional pero recomendado)
    └── categoría.test.js
  
```

Endpoints Mínimos a Implementar:

```
GET    /api/categorias          # Listar categorías (público)
GET    /api/categorias/:id      # Obtener categoría (público)
POST   /api/categorias          # Crear categoría (admin)
PUT    /api/categorias/:id      # Actualizar categoría (admin)
DELETE /api/categorias/:id      # Eliminar categoría (admin)
GET    /api/categorias/:id/productos # Productos de la categoría
POST   /api/categorias/ia/generar  # Generar descripción con IA (admin)
GET    /api/categorias/:id/estadisticas # Estadísticas (admin)
```

Criterios de Evaluación:

8. Funcionalidad Completa (30%)

- Todos los endpoints funcionan correctamente
- Manejo apropiado de errores
- Validaciones exhaustivas

9. Calidad del Código (25%)

- Estructura clara y organizada
- Reutilización de patrones aprendidos
- Comentarios donde es necesario

10. Seguridad (20%)

- Autenticación y autorización correctas
- Validación de datos de entrada
- Manejo seguro de archivos

11. Integración IA (15%)

- Uso creativo de Gemini
- Manejo de errores de API externa
- Funcionalidades que aporten valor

12. Documentación (10%)

- Swagger completo y funcional
- Ejemplos claros
- Descripción de funcionalidades

🔍 Preguntas de Reflexión Final

Después de completar el proyecto, reflexiona sobre:

13. Arquitectura:

- ¿Cómo mejorarías la estructura del proyecto?
- ¿Qué patrones de diseño identificas en tu código?
- ¿Cómo escalarías esta API para miles de usuarios?

14. Performance:

- ¿Qué optimizaciones implementarías?
- ¿Dónde agregarías caché?
- ¿Cómo manejarías la carga de archivos grandes?

15. Seguridad:

- ¿Qué vulnerabilidades potenciales identificas?
- ¿Cómo implementarías rate limiting por usuario?
- ¿Qué auditoría de seguridad realizarías?

16. Integración IA:

- ¿Cómo mejorarías la interacción con Gemini?
- ¿Qué otras funcionalidades de IA agregarías?
- ¿Cómo manejarías costos de API externa?

👉 Desafío Final – Parte B – Analizar Escenarios y actuar en consecuencia

🧙 Escenario 1: El Misterio de las Conexiones Agotadas

Pregunta: "El servidor está lento y la base de datos dice que hay demasiadas conexiones abiertas. ¿Qué falló en el Pool?"

La Respuesta: El fallo principal suele ser la **fuga de conexiones (connection leak)** o una mala gestión del ciclo de vida de las mismas. En el contexto de tu guía, esto ocurre por:

- **No liberar la conexión manualmente:** Cuando solicitas una conexión directamente del pool usando pool.getConnection(), es obligatorio llamar a connection.release() al terminar. Si el código lanza un error y no tienes un bloque finally que asegure la liberación, esa conexión se queda "zombie" y ocupada.
- **Olvidar el uso de pool.execute o pool.query:** Estas funciones de alto nivel en mysql2 se encargan de pedir, usar y devolver la conexión al pool automáticamente. Si el desarrollador intentó gestionar la conexión a mano y olvidó cerrarla, el pool se llena rápido.
- **Configuración del límite insuficiente:** Si el connectionLimit es muy bajo (ej. 10) para el tráfico actual y las consultas son lentas, todas las conexiones estarán ocupadas simultáneamente, haciendo que nuevas peticiones esperen hasta el *timeout*.

🌐 Escenario 2: El Intruso en la Base de Datos

Pregunta: "Un usuario logró registrarse con el nombre <script>alert('hack')</script>. ¿Qué guardia de seguridad (middleware) se quedó dormido?"

La Respuesta: Aquí hubo un fallo en la cadena de mando de la validación y limpieza de datos. Los "guardias" que fallaron fueron:

- **Express-validator (El guardia de la entrada):** Específicamente, faltó la **sanitización** con el método `.escape()`. Este método convierte caracteres peligrosos (como `< y >`) en entidades HTML seguras (como `< y >`), evitando ataques de **XSS (Cross-Site Scripting)**.
 - **Validación de Formato (Regex):** En el `usuarioController.js`, el middleware de validación debería haber aplicado una regla `.matches()` para permitir solo letras y espacios, bloqueando cualquier símbolo especial de código.
 - **El Validador de Resultados:** Si las reglas estaban escritas pero el código continuó la ejecución, significa que el controlador no revisó `validationResult(req).isEmpty()`, dejando pasar al "intruso" a pesar de haber sido detectado.
-

¿Cómo evitar que esto pase en el proyecto final?

Para que no tengan estos sustos, asegúrense de que en el **Módulo 7** implemente:

1. **Bloques Try-Catch-Finally:** Especialmente en el modelo de `Orden.js` donde se usan transacciones complejas.
2. **Middleware de Limpieza:** Un uso estricto de `.trim()`, `.escape()` y `.normalizeEmail()` en todas las entradas de texto del usuario.

Finalizandoooo.....

Archivo de Configuración Final – Así debería lucir

```
# Servidor
PORT=3000
NODE_ENV=development
BASE_URL=http://localhost:3000

# Base de datos
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=tu_password
DB_NAME=marketplace

# JWT
JWT_SECRET=tu_secreto_muy_seguro_aqui_con_al_menos_32_caracteres
JWT_EXPIRE=7d

# Gemini API
GEMINI_API_KEY=tu_api_key_de_gemini_aqui

# Archivos
UPLOAD_PATH=uploads
MAX_FILE_SIZE=5242880

# Email (opcional para notificaciones)
EMAIL_SERVICE=gmail
EMAIL_USER=tu_email@gmail.com
EMAIL_PASS=tu_password_de_aplicacion
```

```
# Seguridad
BCRYPT_ROUNDS=12
RATE_LIMIT_WINDOW=15
RATE_LIMIT_MAX=100
```

Package.json Final

```
{
  "name": "marketplace-inteligente-api",
  "version": "1.0.0",
  "description": "API completa para marketplace con integración de IA",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "jest",
    "test:watch": "jest --watch",
    "docs": "node -e \"console.log('Swagger docs en http://localhost:3000/api-docs')\""",
    "db:migrate": "node scripts/migrate.js",
    "db:seed": "node scripts/seed.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mysql2": "^3.6.5",
    "cors": "^2.8.5",
    "morgan": "^1.10.0",
    "dotenv": "^16.3.1",
    "express-validator": "^7.0.1",
    "bcryptjs": "^2.4.3",
    "jsonwebtoken": "^9.0.2",
    "multer": "^1.4.5-lts.1",
    "swagger-jsdoc": "^6.2.8",
    "swagger-ui-express": "^5.0.0",
    "axios": "^1.6.2",
    "express-rate-limit": "^7.1.5",
    "helmet": "^7.1.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.2",
    "jest": "^29.7.0",
    "supertest": "^6.3.3"
  },
  "keywords": ["api", "marketplace", "nodejs", "express", "ai", "gemini"],
  "author": "Tu Nombre",
  "license": "MIT"
}
```

Extra: “la peor API posible”

Para terminar, si nuestra API fuera diseñada sin:

- Sin validaciones

- Sin hash de contraseña
- Sin JWT
- Sin CORS
- Sin prepared statements

Analicen:

- ¿Qué vulnerabilidades tendrían?
- ¿Cómo la rompería un atacante?



¡Felicitaciones aprendices!

Has completado una guía exhaustiva para crear APIs robustas con Node.js, Express y tecnologías modernas. Este proyecto te ha proporcionado experiencia práctica en:

- Arquitectura de APIs REST profesionales**
- Autenticación y autorización** con JWT
- Gestión de archivos** y validaciones robustas
- Integración de IA** con APIs externas
- Documentación profesional** con Swagger
- Manejo de transacciones** y operaciones complejas
- Filtros avanzados** y optimización de queries
- Patrones de desarrollo** escalables y mantenibles