MANUAL DE TÉCNICO

Practica #4 Desarrollo Web

Jonathan Daniel López Ruano

Carnet: 202300576

Nota: Para arrancar la base de datos en la terminal usar el comando: net start MySQL94

Nota: Para finalizar la base de datos usar el comando: net stop MySQL94

Nota: Para arrancar el backend en la terminal colocarse en su ruta hasta llegar al /Backend y usar el comando: npm run dev

Nota: Para finalizar el backend usar el comando: ctrl+shift+c

Nota: Para arrancar el Frontend en la terminal colocarse en su ruta hasta llegar al /Frontend usar el comando: npm run dev

Nota: Para finalizar el Frontend usar el comando: ctrl+shift+c

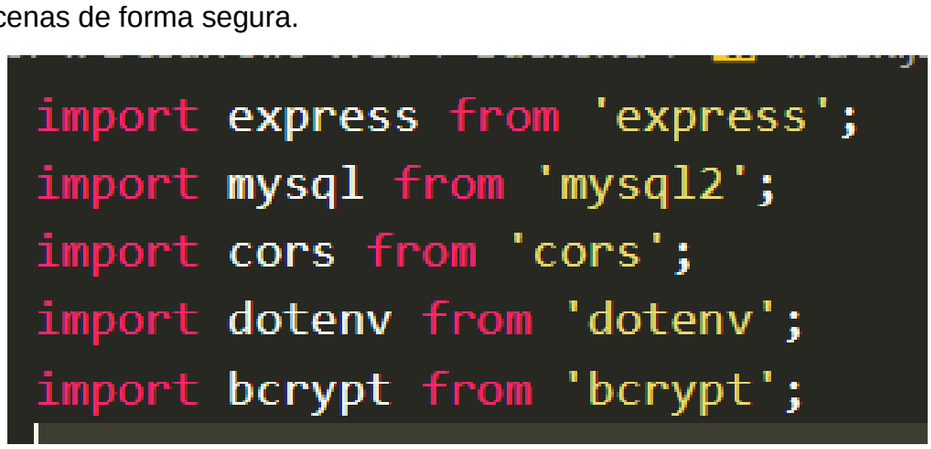
import express from 'express';: Importa el framework Express, que es una de las bibliotecas más populares para crear servidores y APIs con Node.js.

import mysql from 'mysql2';: Importa el paquete mysql2, que permite que tu servidor interactúe con bases de datos MySQL. Es decir, te facilita la conexión, consulta, inserción y actualización de datos.

import cors from 'cors';: Importa la biblioteca CORS (Cross-Origin Resource Sharing), que habilita tu servidor para aceptar peticiones de un origen diferente. Esto es crucial cuando tu frontend y backend se ejecutan en dominios o puertos distintos.

import dotenv from 'dotenv';: Importa el módulo dotenv, que carga variables de entorno desde un archivo .env en tu aplicación. Esto es una práctica de seguridad esencial para mantener información sensible, como las credenciales de la base de datos, fuera de tu código fuente.

import bcrypt from 'bcrypt';: Importa la biblioteca bcrypt, que se utiliza para cifrar y verificar contraseñas. Esto es fundamental para proteger la información de los usuarios, ya que en lugar de almacenar las contraseñas en texto plano, las almacenas de forma segura.



dotenv.config();: Esta línea carga las variables de entorno definidas en un archivo .env en el directorio raíz de tu proyecto. Esto es crucial para mantener información sensible, como credenciales de bases de datos o claves de API, separada de tu código fuente, mejorando la seguridad y la portabilidad.

const app = express();: Aquí se crea una instancia de la aplicación Express. app se convierte en el objeto principal que utilizarás para definir rutas, configurar middleware y gestionar las solicitudes entrantes.

app.use(cors());: Esta línea habilita el middleware CORS (Cross-Origin Resource Sharing). Esto permite que tu servidor responda a solicitudes provenientes de

diferentes dominios, protocolos o puertos. Es fundamental cuando tu frontend y backend se desarrollan por separado o se alojan en ubicaciones distintas.

app.use(express.json());: Esta línea integra el middleware express.json(). Este middleware parsea las peticiones entrantes con cuerpos JSON. Significa que si un cliente envía datos en formato JSON en el cuerpo de una petición (por ejemplo, al hacer una solicitud POST o PUT), Express podrá leer y procesar esos datos fácilmente, haciéndolos disponibles en req.body.

```
// Cargar variables de entorno desde el archivo .env
dotenv.config();
const app = express();
app.use(cors());
app.use(express.json());
```

const pool = mysql.createPool({...}).promise();: Crea un grupo de conexiones (connection pool). Este pool mantiene un conjunto de conexiones a la base de datos que están listas para ser utilizadas, lo que reduce la latencia y la carga del servidor. La llamada a .promise() convierte el pool en una versión que usa promesas, lo que permite manejar las operaciones de la base de datos con código asíncrono más limpio (usando async/await) en lugar de callbacks.

Configuración con variables de entorno: Las propiedades de configuración (como host, user, password, y database) obtienen sus valores del archivo de variables de entorno .env. Si la variable de entorno no está definida (por ejemplo, process.env.DB_HOST), se utiliza un valor predeterminado ('localhost'). Esto es una excelente práctica de seguridad y flexibilidad, ya que:

Seguridad: Evita que las credenciales sensibles (como la contraseña) queden expuestas en el código fuente.

Flexibilidad: Permite cambiar fácilmente la configuración de la base de datos entre entornos de desarrollo y producción sin modificar el código.

Propiedades específicas del pool:

waitForConnections: true: Permite que las peticiones se pongan en cola si todas las conexiones del pool están en uso.

connectionLimit: 10: Establece el número máximo de conexiones simultáneas que el pool puede mantener. En este caso, el servidor puede gestionar hasta 10 conexiones a la base de datos al mismo tiempo.

```
// Configuración de la conexión a la base de datos
const pool = mysql.createPool({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'root',
  password: process.env.DB_PASS || 'Jlpz.10107076',
  database: process.env.DB_NAME || 'califica_catedratico',
    waitForConnections: true,
    connectionLimit: 10
}).promise();
```

app.post("/usuarios/register", async (req, res) => { ... });: Define la ruta que escucha las peticiones POST a la dirección /usuarios/register. La función es async, lo que permite el uso de await para manejar operaciones asíncronas como la interacción con la base de datos y el cifrado de la contraseña.

try...catch: La lógica está envuelta en un bloque try...catch para manejar posibles errores, como datos incorrectos o fallos en la base de datos. Si todo va bien, el código del bloque try se ejecuta. Si ocurre un error, la ejecución salta al bloque catch.

const { registro_academico, nombres, apellidos, correo, contrasena } = req.body;: Utiliza la desestructuración de objetos para extraer los datos enviados en el cuerpo de la petición (req.body). Esto hace que el código sea más limpio y legible.

const hashed = await bcrypt.hash(contrasena, 10);: Esta es la parte más importante para la seguridad. Antes de almacenar la contraseña, se utiliza bcrypt para cifrarla. bcrypt.hash() genera un hash (una cadena de texto irreversible) de la contraseña original. El número 10 es el factor de salt (o costo), que determina la complejidad del cifrado. Un valor más alto hace que el hash sea más lento de calcular, lo que dificulta los ataques de fuerza bruta.

await pool.query(...): Esta línea ejecuta una sentencia SQL de inserción para guardar los datos del nuevo usuario en la tabla usuarios. Es crucial notar que se utiliza una consulta parametrizada (con los signos ?). Esto previene ataques de inyección SQL, que son una vulnerabilidad común y peligrosa. En lugar de concatenar la cadena de consulta, se pasan los valores en un array, y mysql2 se encarga de escapar y sanitizar los datos de forma segura.

res.status(201).json(...): Si la inserción es exitosa, el servidor responde con un código de estado 201 (Created) y un mensaje JSON que confirma que el usuario ha sido registrado.

res.status(500).json(...): Si hay un error, el servidor responde con un código de estado 500 (Internal Server Error) y un mensaje de error que ayuda a identificar el problema.

app.put('/usuarios/updateUser/:id', async (req, res) => { ... });: Define la ruta que escucha las peticiones PUT a la dirección /usuarios/updateUser/ seguida de un identificador. El :id en la URL es un parámetro de ruta que captura el id del usuario que se va a actualizar. Esto permite que el mismo endpoint pueda actualizar a cualquier usuario, haciendo la API más flexible.

try...catch: Al igual que en el endpoint de registro, la lógica está envuelta en un bloque try...catch para manejar errores de manera robusta.

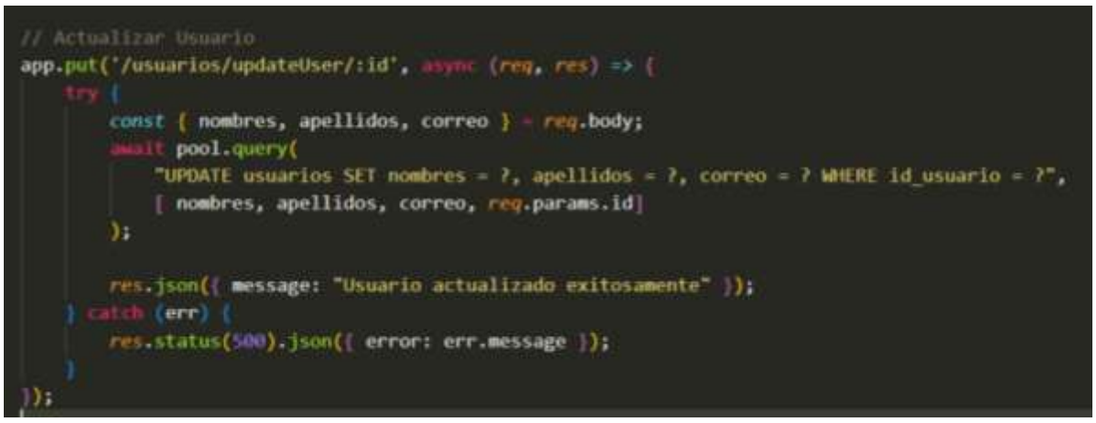
const { nombres, apellidos, correo } = req.body;: Extrae los datos que el usuario desea actualizar (el nombre, los apellidos y el correo) del cuerpo de la petición.

await pool.query(...): Esta línea ejecuta una sentencia SQL de actualización (UPDATE). La sentencia actualiza los campos nombres, apellidos y correo en la tabla usuarios únicamente para el usuario cuyo id_usuario coincide con el valor proporcionado en el parámetro de la ruta (req.params.id).

Seguridad (inyección SQL): Nuevamente, se utiliza una consulta parametrizada (con los signos ?) para pasar los valores de manera segura y evitar la inyección de código SQL malicioso.

res.json({ message: "Usuario actualizado exitosamente" });: Si la actualización se realiza sin problemas, el servidor responde con un mensaje JSON de éxito. A diferencia del registro (201), se usa un código de estado 200 (implícito en res.json()), que es el estándar para peticiones exitosas sin contenido (OK).

res.status(500).json(...): En caso de cualquier error (por ejemplo, el usuario no existe o un fallo en la base de datos), se envía una respuesta con el código de estado 500 y el mensaje de error correspondiente.



```
// Actualizar Usuario
app.put('/usuarios/updateUser/:id', async (req, res) => {
    try {
        const { nombres, apellidos, correo } = req.body;
        await pool.query(
            "UPDATE usuarios SET nombres = ?, apellidos = ?, correo = ? WHERE id_usuario = ?",
            [ nombres, apellidos, correo, req.params.id]
        );

        res.json({ message: "Usuario actualizado exitosamente" });
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.post('/usuarios/recuperar', async (req, res) => { ... });: Define una ruta que escucha las peticiones POST en /usuarios/recuperar. Esta ruta se usa para enviar la información necesaria para el proceso de recuperación.

try...catch: Como en los bloques anteriores, se utiliza para un manejo de errores robusto. Si ocurre algún problema durante el proceso, se capturará y se enviará una respuesta de error al cliente.

const { registro_academico, correo, nuevaContrasena } = req.body;: Desestructura el cuerpo de la petición para obtener el registro_academico, el correo del usuario y la nuevaContrasena que desean establecer.

const [rows] = await pool.query('SELECT * FROM usuarios WHERE registro_academico = ? AND correo = ?', [registro_academico, correo]);: Se ejecuta una consulta SQL para buscar un usuario que coincida tanto con el registro_academico como con el correo proporcionados. El uso de ? indica parámetros que se sustituyen de forma segura para prevenir inyecciones SQL.

if (rows.length === 0) { return res.status(404).json({ error: 'Datos incorrectos' }); }: Si la consulta no devuelve ningún resultado (rows.length === 0), significa que los datos proporcionados no corresponden a ningún usuario en la base de datos. En este caso, se responde con un código de estado 404 (Not Found) y un mensaje indicando que los datos son incorrectos.

const hashed = await bcrypt.hash(nuevaContrasena, 10);: Si se encuentra un usuario coincidente, la nuevaContrasena se cifra utilizando bcrypt.hash() con un factor de salt de 10. Esto asegura que la nueva contraseña se almacene de forma segura en la base de datos.

await pool.query('UPDATE usuarios SET contrasena = ? WHERE id_usuario = ?', [hashed, rows[0].id_usuario]);: Se ejecuta una sentencia UPDATE para actualizar el

campo contrasena del usuario encontrado con la nueva contraseña cifrada. Se utiliza rows[0].id_usuario para asegurar que la actualización se aplique al usuario correcto.

res.json({ message: 'Contraseña actualizada exitosamente' });: Si la actualización se completa con éxito, el servidor responde con un mensaje JSON que confirma la actualización.

res.status(500).json({ error: err.message });: Si ocurre algún error durante el proceso de consulta o actualización, se responde con un código de estado 500 (Internal Server Error) y el mensaje de error.

Endpoint de Recuperación de Contraseña

app.post('/usuarios/recuperar', async (req, res) => { ... });: Esta ruta se utiliza cuando un usuario ha olvidado su contraseña y quiere restablecerla. Espera recibir el registro académico, el correo electrónico y la nueva contraseña en el cuerpo de la petición.

const [rows] = await pool.query('SELECT * FROM usuarios WHERE registro_academico = ? AND correo = ?', ...);: Primero, se realiza una consulta a la base de datos para verificar que el registro_academico y el correo coincidan con un usuario existente. Esto es una medida de seguridad para asegurarse de que solo se pueda cambiar la contraseña si el usuario puede proporcionar ambos datos correctamente.

if (rows.length === 0) { ... }: Si la consulta no devuelve ningún resultado, significa que la combinación de datos es incorrecta, por lo que el servidor responde con un código de estado 404 (Not Found) y un mensaje de error claro.

const hashed = await bcrypt.hash(nuevaContrasena, 10);: Si la validación es exitosa, la nueva contraseña se cifra con bcrypt antes de ser almacenada. Esto es crucial para la seguridad, ya que la contraseña no se guarda en texto plano en la base de datos.

await pool.query('UPDATE usuarios SET contrasena = ? WHERE id_usuario = ?', ...);: Finalmente, se ejecuta una consulta UPDATE para reemplazar la contraseña antigua con el nuevo hash. Se utiliza el id_usuario (rows[0].id_usuario) para asegurarse de que se actualice la contraseña del usuario correcto.

res.json({ message: 'Contraseña actualizada exitosamente' });: Si todo el proceso se completa sin errores, se envía una respuesta de éxito al cliente.

Endpoint de Inicio de Sesión

app.post('/usuarios/login', async (req, res) => { ... });: Esta ruta maneja el proceso de autenticación de usuarios. Espera recibir el correo electrónico y la contraseña en el cuerpo de la petición.

const [rows] = await pool.query('SELECT * FROM usuarios WHERE correo = ?', ...);: Se realiza una consulta para encontrar al usuario en la base de datos basándose en el correo electrónico.

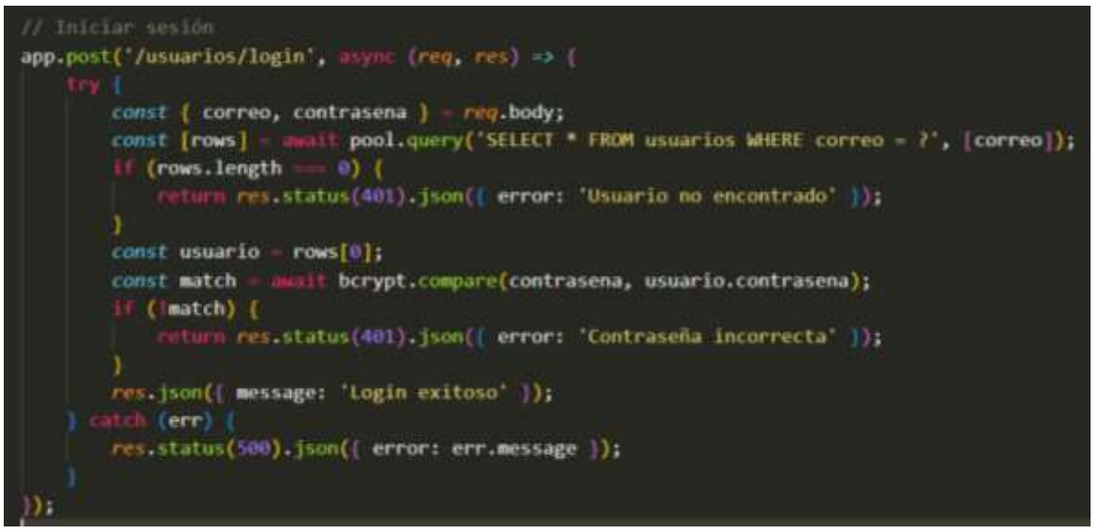
Validación de usuario y contraseña:

if (rows.length === 0) { ... }: Si no se encuentra el correo, el servidor responde con un código 401 (Unauthorized) y un mensaje de "Usuario no encontrado", evitando dar pistas a posibles atacantes sobre la existencia del correo en la base de datos.

const match = await bcrypt.compare(contrasena, usuario.contrasena);: Aquí se utiliza bcrypt.compare() para comparar la contraseña enviada (en texto plano) con el hash almacenado en la base de datos. bcrypt realiza este proceso de forma segura y devuelve true si la contraseña coincide con el hash y false si no.

if (!match) { ... }: Si las contraseñas no coinciden, se envía otro código 401 con un mensaje de "Contraseña incorrecta".

res.json({ message: 'Login exitoso' });: Si tanto el usuario como la contraseña son correctos, el servidor responde con un mensaje de éxito.



```
// Iniciar sesión
app.post('/usuarios/login', async (req, res) => {
    try {
        const { correo, contrasena } = req.body;
        const [rows] = await pool.query('SELECT * FROM usuarios WHERE correo = ?', [correo]);
        if (rows.length === 0) {
            return res.status(401).json({ error: 'Usuario no encontrado' });
        }
        const usuario = rows[0];
        const match = await bcrypt.compare(contrasena, usuario.contrasena);
        if (!match) {
            return res.status(401).json({ error: 'Contraseña incorrecta' });
        }
        res.json({ message: 'Login exitoso' });
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.get('/verPerfil/:id', async (req, res) => { ... });:

Define una ruta que escucha las peticiones GET en la URL /verPerfil/.

El :id en la URL es un parámetro de ruta. Esto significa que la ruta espera recibir un valor específico después de /verPerfil/ (por ejemplo, /verPerfil/123), y este valor se capturará y estará disponible como req.params.id.

try...catch: Como es habitual, se utiliza un bloque try...catch para manejar cualquier error que pueda ocurrir durante la ejecución, ya sea al interactuar con la base de datos o por algún otro problema.

const [rows] = await pool.query('SELECT id_usuario, registro_academico, nombres, apellidos, correo FROM usuarios WHERE id_usuario = ?', [req.params.id]);:

Se ejecuta una consulta SQL para seleccionar columnas específicas (id_usuario, registro_academico, nombres, apellidos, correo) de la tabla usuarios.

La cláusula WHERE id_usuario = ? filtra los resultados para obtener únicamente el registro del usuario cuyo id_usuario coincide con el valor proporcionado en el parámetro de la ruta (req.params.id).

Se utiliza una consulta parametrizada para garantizar la seguridad, previniendo así ataques de inyección SQL.

if (rows.length === 0) { return res.status(404).json({ error: 'Usuario no encontrado' }); }:

Si la consulta SELECT no devuelve ninguna fila (es decir, rows.length es 0), significa que no existe ningún usuario con el id proporcionado.

En este caso, el servidor responde con un código de estado 404 (Not Found) y un mensaje de error indicando que el usuario no fue encontrado. La palabra clave return asegura que la función termine aquí y no continúe ejecutando el resto del código.

res.json(rows[0]);:

Si se encuentra al usuario (la consulta devuelve al menos una fila), se toma la primera fila encontrada (rows[0]) que contiene los datos del perfil del usuario.

Esta información se envía de vuelta al cliente en formato JSON como respuesta exitosa. Por defecto, Express enviará un código de estado 200 (OK).

Endpoint para Ver Perfil de Usuario

app.get('/verPerfil/:id', async (req, res) => { ... });: Esta ruta responde a peticiones GET y está diseñada para mostrar el perfil de un usuario específico. El :id en la URL es un parámetro de ruta que identifica al usuario.

const [rows] = await pool.query('SELECT id_usuario, registro_academico, nombres, apellidos, correo FROM usuarios WHERE id_usuario = ?', [req.params.id]);: La consulta SQL busca al usuario por su id. Es una buena práctica de seguridad seleccionar solo los campos necesarios y evitar devolver información sensible como la contraseña. En este caso, solo se recuperan los datos públicos del usuario.

if (rows.length === 0) { ... }: Si la consulta no encuentra un usuario con el id proporcionado, el servidor responde con un código de estado 404 (Not Found), indicando que el recurso no existe.

res.json(rows[0]);: Si se encuentra el usuario, el servidor responde con un objeto JSON que contiene los datos del perfil. rows[0] se usa para acceder al primer (y único) registro del array de resultados.

Endpoint para Obtener Todos los Profesores

app.get('/profesores/obtenerProfesores', async (req, res) => { ... });: Este endpoint se encarga de listar todos los profesores que están registrados en la base de datos.

const [rows] = await pool.query(...): La consulta SQL es simple y directa: selecciona el id, el nombre y los apellidos de la tabla profesores.

ORDER BY nombres ASC;: La cláusula ORDER BY asegura que la lista de profesores se devuelva ordenada alfabéticamente por nombre, lo que mejora la usabilidad para el cliente.

res.json(rows);: Finalmente, el servidor envía un array de objetos JSON, donde cada objeto representa a un profesor.

app.get('/profesores/:id_profesor', async (req, res) => { ... });: Esta línea configura una ruta GET que espera un parámetro de ruta (:id_profesor). Este parámetro es clave para identificar de forma específica al profesor que se desea buscar.

const { id_profesor } = req.params;: Aquí se utiliza la desestructuración de objetos para extraer el id_profesor del objeto req.params, haciendo el código más limpio y legible.

const [rows] = await pool.query("SELECT id_profesor, nombres, apellidos FROM profesores WHERE id_profesor = ?", [id_profesor]);: Se ejecuta una consulta SQL para buscar un registro en la tabla profesores. Es una buena práctica de seguridad seleccionar únicamente los campos necesarios y no toda la fila (*), lo que minimiza la cantidad de datos enviados. El uso de la consulta parametrizada (?) previene ataques de inyección SQL.

if (rows.length === 0) { ... }: Esta es una verificación crucial. Si la consulta devuelve un array vacío (rows.length === 0), significa que no se encontró un profesor con el id proporcionado. En este caso, el servidor responde con un código de estado 404 (Not Found) y un mensaje de error explicativo.

res.json(rows[0]);: Si el profesor es encontrado, el servidor responde con un objeto JSON que contiene los datos del profesor (id, nombres y apellidos). Se utiliza rows[0] para acceder directamente al primer (y único) registro del resultado.

app.get('/profesores/:id_profesor/cursos', async (req, res) => { ... });: Esta ruta utiliza el id del profesor como parte de la URL. Este enfoque es común en el diseño de APIs REST y se conoce como rutas anidadas, ya que muestra la relación entre profesores y cursos.

const { id_profesor } = req.params;: Se extrae el id_profesor del parámetro de la URL.

const [rows] = await pool.query(...): Se ejecuta una consulta SQL que usa una cláusula JOIN para combinar datos de dos tablas: cursos (c) y profesores (p).

JOIN profesores p ON c.id_profesor = p.id_profesor: Esta línea une las filas de la tabla cursos con las de la tabla profesores donde el id_profesor en ambas tablas coincide.

WHERE c.id_profesor = ?: Esta condición filtra el resultado para incluir solo los cursos que pertenecen al profesor con el id proporcionado en la URL.

SELECT c.id_curso, c.nombre_curso, p.nombres AS nombre_profesor, p.apellidos AS apellido_profesor: Se seleccionan campos específicos de ambas tablas. Se utilizan alias (AS) para renombrar las columnas del profesor (nombres, apellidos) y evitar conflictos con la columna del curso del mismo nombre.

if (rows.length === 0) { ... }: Esta validación verifica si la consulta arrojó algún resultado. Si no se encuentran cursos, el servidor devuelve un código de estado 404 (Not Found) con un mensaje indicando que el profesor no tiene cursos asignados o que no existe.

res.json(rows);: Si se encuentran cursos, se envía una respuesta JSON que contiene un array con todos los cursos y el nombre completo del profesor.

```
// Obtener cursos impartidos por un profesor
app.get('/profesores/:id_profesor/cursos', async (req, res) => {
    try {
        const { id_profesor } = req.params;
        const [rows] = await pool.query(
            `SELECT c.id_curso, c.nombre_curso, p.nombres AS nombre_profesor, p.apellidos AS apellido_profesor
             FROM cursos c
             JOIN profesores p ON c.id_profesor = p.id_profesor
             WHERE c.id_profesor = ?`,
            [id_profesor]
        );

        if (rows.length === 0) {
            return res.status(404).json({ error: "Este profesor no tiene cursos asignados o no existe" });
        }

        res.json(rows);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.post('/publicaciones/crearPublicacion', async (req, res) => { ... });: Esta ruta escucha peticiones POST para la creación de publicaciones. La función es async para poder usar await al interactuar con la base de datos.

const { id_usuario, id_publicacion, id_profesor, mensaje } = req.body;: Se desestructuran los datos enviados en el cuerpo de la petición.

if (!id_usuario || !mensaje) { ... }: Esta es una validación de entrada crucial. Se verifica que los campos id_usuario y mensaje no estén vacíos. Si alguno de estos datos obligatorios falta, el servidor responde con un código de estado 400 (Bad Request) y un mensaje de error, evitando que se guarden datos incompletos en la base de datos.

await pool.query(...): Esta línea ejecuta la sentencia SQL de inserción (INSERT INTO).

id_publicacion || null: Esta es una técnica para manejar campos opcionales. Si el cliente envía un id_publicacion, se utiliza; si no, se inserta null en ese campo de la base de datos.

id_profesor || null: Lo mismo aplica para el campo id_profesor. Esto permite que una publicación pueda ser una crítica a un profesor (id_profesor definido) o una publicación general sin un profesor específico (id_profesor nulo).

[id_usuario, id_publicacion || null, id_profesor || null, mensaje]: Se utiliza una consulta parametrizada para pasar los valores de forma segura y prevenir inyecciones SQL.

res.status(201).json(...): Si la inserción se realiza con éxito, el servidor responde con un código 201 (Created) y un mensaje de éxito, indicando que la publicación ha sido creada.

res.status(500).json({ error: err.message });: Como en los bloques anteriores, este bloque catch maneja cualquier error que pueda ocurrir durante la operación, como un fallo de la base de datos, y envía una respuesta de error al cliente.

```
// ---------- Publicaciones ----------
// Crear Publicación
app.post('/publicaciones/crearPublicacion', async (req, res) => {
    try {
        const { id_usuario, id_publicacion, id_profesor, mensaje } = req.body;

        if (!id_usuario || !mensaje) {
            return res.status(400).json({ error: "Faltan datos obligatorios" });
        }


        await pool.query(
            "INSERT INTO publicaciones (id_usuario, id_publicacion, id_profesor, mensaje) VALUES (?, ?, ?, ?)",
            [id_usuario, id_publicacion || null, id_profesor || null, mensaje]
        );

        res.status(201).json({ message: "Publicación creada exitosamente" });
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.get('/publicaciones/obtenerPublicaciones', async (req, res) => { ... });: Esta ruta responde a peticiones GET y no requiere ningún parámetro, ya que su objetivo es listar todas las publicaciones disponibles.

const [rows] = await pool.query(...): Se ejecuta una consulta SQL que utiliza múltiples cláusulas JOIN para enriquecer los datos de las publicaciones.

JOIN usuarios u ON p.id_usuario = u.id_usuario: Este es un JOIN estándar que asegura que cada publicación esté asociada a un usuario. Esto es esencial para mostrar el nombre del autor de la publicación.

LEFT JOIN cursos c ON p.id_publicacion = c.id_curso: Este es un LEFT JOIN. A diferencia de un JOIN regular, un LEFT JOIN devuelve todas las filas de la tabla de la izquierda (publicaciones), incluso si no hay una coincidencia en la tabla de la derecha (cursos). Esto es crucial porque una publicación puede estar relacionada o no con un curso.

LEFT JOIN profesores pr ON p.id_profesor = pr.id_profesor: Similar al anterior, este LEFT JOIN une la publicación con un profesor si existe una relación, pero no excluye las publicaciones que no están asociadas a un profesor.

ORDER BY p.fecha_creacion DESC: La cláusula ORDER BY ordena los resultados por fecha de creación de forma descendente, lo que significa que las publicaciones más recientes aparecen primero. Esta es una funcionalidad básica y esperada en cualquier lista de publicaciones.

res.json(rows);: Finalmente, el servidor envía un array de objetos JSON, donde cada objeto contiene la información completa de una publicación, incluyendo el mensaje, la fecha de creación, el nombre del estudiante que la creó, y el nombre del curso y profesor si están asociados.

```
// Obtener todas las publicaciones
app.get('/publicaciones/obtenerPublicaciones', async (req, res) => {
    try {
        const [rows] = await pool.query(`
            SELECT p.id_publicacion, p.mensaje, p.fecha_creacion,
            u.nombres as estudiante, c.nombre_curso, pr.nombres AS profesor
            from publicaciones p
            JOIN usuarios u ON p.id_usuario = u.id_usuario
            LEFT JOIN cursos c ON p.id_publicacion = c.id_curso
            LEFT JOIN profesores pr ON p.id_profesor = pr.id_profesor
            ORDER BY p.fecha_creacion DESC; `, [req.params.id]);
        res.json(rows);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.get('/publicaciones/curso/:id_publicacion', async (req, res) => { ... });: Esta ruta escucha peticiones GET y utiliza un parámetro de ruta (:id_publicacion) para identificar la publicación que se desea recuperar. El nombre curso en la ruta parece ser un remanente o un intento de indicar que las publicaciones pueden estar relacionadas con cursos, aunque la consulta se enfoca en el id_publicacion.

const [rows] = await pool.query(...): La consulta SQL es casi idéntica a la del endpoint anterior, pero con una condición WHERE clave:

WHERE p.id_publicacion = ?: Aquí es donde se filtra la información. La consulta solo devolverá la fila de la tabla publicaciones (y sus registros unidos de usuarios, cursos, y profesores) que coincida con el id_publicacion proporcionado en la URL.

Los JOINs (usuarios, cursos, profesores) y el ORDER BY siguen funcionando de la misma manera que en el endpoint anterior, asegurando que se recuperen todos los datos relacionados y se mantenga el orden (aunque en este caso, al ser una sola publicación, el ORDER BY tiene un efecto mínimo).

res.json(rows);: Si se encuentra la publicación, el servidor responde con un array JSON que contiene un solo objeto (o un array vacío si no se encuentra). Dado que la consulta busca por un ID único, se espera que devuelva un solo resultado.

Manejo de errores: El bloque try...catch sigue siendo fundamental para capturar cualquier error que pueda ocurrir durante la consulta a la base de datos y responder con un código de estado 500 (Internal Server Error).

```
// Obtener una publicación por curso
app.get('/publicaciones/curso/:id_publicacion', async (req, res) => {
    try {
        const [rows] = await pool.query(`
            SELECT p.id_publicacion, p.mensaje, p.fecha_creacion,
                u.nombres as estudiante, c.nombre_curso, pr.nombres AS profesor
                from publicaciones p
            JOIN usuarios u ON p.id_usuario = u.id_usuario
            LEFT JOIN cursos c ON p.id_publicacion = c.id_curso
            LEFT JOIN profesores pr ON p.id_profesor = pr.id_profesor
                WHERE p.id_publicacion = ?
            ORDER BY p.fecha_creacion DESC; `, [req.params.id_publicacion]);
        res.json(rows);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

Endpoint para Obtener Publicaciones por Curso

app.get('/publicaciones/curso/:id_publicacion', async (req, res) => { ... });: Esta ruta utiliza el id_publicacion como un parámetro de ruta. Aunque el nombre del parámetro sugiere que es el id de la publicación, la consulta SQL lo usa como un id_curso. Una mejor práctica sería nombrar la ruta publicaciones/curso/:id_curso para mayor claridad.

const [rows] = await pool.query(... WHERE p.id_publicacion = ? ..., [req.params.id_publicacion]);: Esta consulta es idéntica a la que se utiliza para

obtener todas las publicaciones (con múltiples JOIN), pero añade una cláusula WHERE que filtra los resultados para mostrar solo las publicaciones cuyo id_publicacion coincide con el valor del parámetro de la URL. Como se mencionó, esto puede generar confusión, ya que el campo de filtro debería ser un id_curso para que la ruta sea coherente.

res.json(rows);: La respuesta JSON es un array de publicaciones que cumplen con la condición de filtro.

Endpoint para Obtener Publicaciones por Profesor

app.get('/publicaciones/profesor/:id_profesor', async (req, res) => { ... });: Esta ruta, bien nombrada, utiliza el id_profesor como parámetro de ruta para filtrar las publicaciones.

const [rows] = await pool.query(... WHERE p.id_profesor = ? ..., [req.params.id_profesor]);: La consulta SQL, similar a la anterior, utiliza un WHERE p.id_profesor = ? para filtrar las publicaciones por el id del profesor. Este enfoque es lógico y coherente con el propósito de la ruta.

res.json(rows);: La respuesta JSON es un array de publicaciones relacionadas con el profesor especificado.

app.get('/publicaciones/usuario/:id_usuario', async (req, res) => { ... });: Esta ruta utiliza el id_usuario como un parámetro de ruta, lo que hace que la API sea flexible y permita buscar publicaciones de cualquier usuario.

const [rows] = await pool.query(... WHERE p.id_usuario = ? ..., [req.params.id_usuario]);: La consulta SQL es la misma que la utilizada para obtener todas las publicaciones (con JOIN para obtener información de usuarios, cursos y profesores), pero se le añade una cláusula WHERE para filtrar los resultados. Esta cláusula asegura que solo se devuelvan las publicaciones donde el id_usuario en la tabla de publicaciones coincida con el id proporcionado en el parámetro de la URL.

res.json(rows);: Si se encuentran publicaciones, el servidor responde con un array de objetos JSON, donde cada objeto representa una publicación del usuario.

app.post('/comentarios/crearComentario', async (req, res) => { ... });: Esta ruta escucha las peticiones POST a la URL /comentarios/crearComentario. La función es asíncrona para poder interactuar con la base de datos de manera no bloqueante.

const { id_usuario, id_publicacion, mensaje } = req.body;: Se extraen del cuerpo de la petición los datos esenciales para un comentario: el id del usuario que lo crea, el id de la publicación a la que pertenece y el texto del mensaje.

if (!id_usuario || !id_publicacion || !mensaje) { ... }: Esta es una validación de entrada crucial. Se verifica que todos los datos obligatorios estén presentes. Si falta alguno, el servidor responde con un código de estado 400 (Bad Request) y un mensaje de error, impidiendo que se intente crear un comentario incompleto.

await pool.query("INSERT INTO comentarios (id_usuario, id_publicacion, mensaje) VALUES (?, ?, ?)", [id_usuario, id_publicacion, mensaje]);: Se ejecuta la sentencia SQL de inserción (INSERT INTO) para guardar el comentario en la tabla comentarios. Se utiliza una consulta parametrizada para prevenir ataques de inyección SQL.

res.status(201).json({ message: "Comentario creado exitosamente" });: Si la inserción es exitosa, el servidor responde con un código de estado 201 (Created) y un mensaje de éxito, confirmando que el comentario ha sido creado.

res.status(500).json({ error: err.message });: En caso de cualquier error (por ejemplo, un problema de conexión a la base de datos), el bloque catch maneja la excepción y envía una respuesta de error al cliente.

app.get('/comentarios/obtenerComentarios', async (req, res) => { ... });: Esta ruta responde a peticiones GET y no requiere ningún parámetro, ya que su objetivo es listar todos los comentarios disponibles.

const [rows] = await pool.query(...): Se ejecuta una consulta SQL que utiliza múltiples cláusulas JOIN para enriquecer los datos de los comentarios.

JOIN usuarios u ON c.id_usuario = u.id_usuario: Este JOIN une los comentarios con los usuarios para obtener el nombre del estudiante que realizó el comentario.

JOIN publicaciones p ON c.id_publicacion = p.id_publicacion: Este JOIN une los comentarios con las publicaciones para obtener el mensaje de la publicación a la que pertenece el comentario.

ORDER BY c.fecha_creacion DESC: La cláusula ORDER BY ordena los resultados por fecha de creación de forma descendente, lo que significa que los comentarios más recientes aparecen primero.

res.json(rows);: Finalmente, el servidor envía un array de objetos JSON, donde cada objeto contiene la información completa de un comentario, incluyendo el mensaje,

la fecha de creación, el nombre del estudiante que lo creó y el mensaje de la publicación a la que pertenece.

app.get('/comentarios/:id_comentario', async (req, res) => { ... });: Esta ruta utiliza el id_comentario como un parámetro de ruta para identificar de forma precisa el comentario que se desea obtener.

const [rows] = await pool.query(...): La consulta SQL es la misma que la utilizada para listar todos los comentarios, pero añade una cláusula WHERE para filtrar los resultados. Esta cláusula WHERE c.id_comentario = ? asegura que solo se devuelva el comentario que coincida con el id proporcionado en el parámetro de la URL.

res.json(rows);: Si se encuentra el comentario, el servidor responde con un array que contiene un único objeto JSON (el comentario). Es importante notar que si el comentario no existe, el array estará vacío, y aunque no hay una validación explícita para el 404, la respuesta del servidor seguirá siendo un array vacío, lo que el cliente puede interpretar.

app.get('/comentarios/publicacion/:id_publicacion', async (req, res) => { ... });: Esta ruta utiliza el id_publicacion como un parámetro de ruta. Esto permite que la API sea flexible y pueda ser consultada para obtener los comentarios de cualquier publicación.

const [rows] = await pool.query(...): La consulta SQL es la misma que se utilizó para obtener todos los comentarios, pero se le añade una cláusula WHERE para filtrar los resultados. Esta cláusula WHERE c.id_publicacion = ? asegura que solo se devuelvan los comentarios que estén asociados a la publicación con el id proporcionado en la URL.

res.json(rows);: La respuesta del servidor es un array de objetos JSON que contienen la información completa de cada comentario, incluyendo los detalles del usuario y de la publicación a la que pertenecen.

```
// Obtener todos los comentarios de una publicación
app.get('/comentarios/publicacion/:id_publicacion', async (req, res) => {
    try {
        const [rows] = await pool.query(`
            SELECT c.id_comentario, c.mensaje, c.fecha_creacion,
                u.nombres as estudiante, p.mensaje as publicacion
                from comentarios c
            JOIN usuarios u ON c.id_usuario = u.id_usuario
            JOIN publicaciones p ON c.id_publicacion = p.id_publicacion
                WHERE c.id_publicacion = ?
            ORDER BY c.fecha_creacion DESC; `, [req.params.id_publicacion]);
        res.json(rows);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.post('/usuarios/:id_usuario/cursos_aprobados', async (req, res) => { ... });: Esta ruta utiliza el id_usuario como un parámetro de ruta, lo que permite asociar el curso aprobado al usuario correcto.

const { id_usuario } = req.params; y const { id_curso } = req.body;: Se extraen los datos necesarios de la URL y del cuerpo de la petición.

if (!id_curso) { ... }: Se realiza una validación de entrada para asegurar que se proporcione el id_curso en la petición. Si falta, se envía una respuesta de error con un código de estado 400 (Bad Request).

await pool.query("INSERT INTO cursos_aprobador (id_usuario, id_curso) VALUES (?, ?)", [id_usuario, id_curso]);: Esta línea ejecuta una sentencia SQL de inserción para crear una nueva fila en la tabla intermedia. Esta tabla solo contiene los id de usuario y los id de curso, lo que establece la relación entre ambos. Se utiliza una consulta parametrizada para evitar la inyección de SQL.

Manejo de errores específicos:

if (err.code === "ER_DUP_ENTRY") { ... }: Este bloque if es muy importante. La base de datos, en caso de intentar insertar una fila que ya existe (lo que ocurre cuando un usuario ya tiene ese curso aprobado), devolverá un error con el código ER_DUP_ENTRY. El código captura este error específico y envía una respuesta amigable con un código de estado 400 y un mensaje que indica que el curso ya ha sido registrado.

res.status(500).json({ error: err.message });: Si el error no es de entrada duplicada, se envía una respuesta de error genérica del servidor.

```
// ---------------- Cursos aprobados ----------------
// Agregar curso aprobado a un usuario
app.post('/usuarios/:id_usuario/cursos_aprobados', async (req, res) => {
    try {
        const { id_usuario } = req.params;
        const { id_curso } = req.body;

        if (!id_curso) {
            return res.status(400).json({ error: "Debe enviar el id_curso" });
        }

        // Insertar relación en la tabla intermedia
        await pool.query(
            "INSERT INTO cursos_aprobador (id_usuario, id_curso) VALUES (?, ?)",
            [id_usuario, id_curso]
        );

        res.status(201).json({ message: "Curso aprobado agregado exitosamente" });
    } catch (err) {
        // Evitar error duplicado de PK (usuario ya tiene ese curso aprobado)
        if (err.code === "ER_DUP_ENTRY") {
            return res.status(400).json({ error: "Este curso ya fue aprobado por el usuario" });
        }
        res.status(500).json({ error: err.message });
    }
});
```

app.get('/usuarios/:id_usuario/cursos_aprobados', async (req, res) => { ... });: Esta ruta utiliza el id_usuario como un parámetro de ruta, permitiendo consultar los cursos aprobados para cualquier usuario específico.

const { id_usuario } = req.params;: Se extrae el id_usuario de los parámetros de la URL.

const [rows] = await pool.query(...): Se ejecuta una consulta SQL que utiliza un INNER JOIN para combinar información de dos tablas:

cursos_aprobador (ca): La tabla intermedia que registra qué usuario aprobó qué curso.

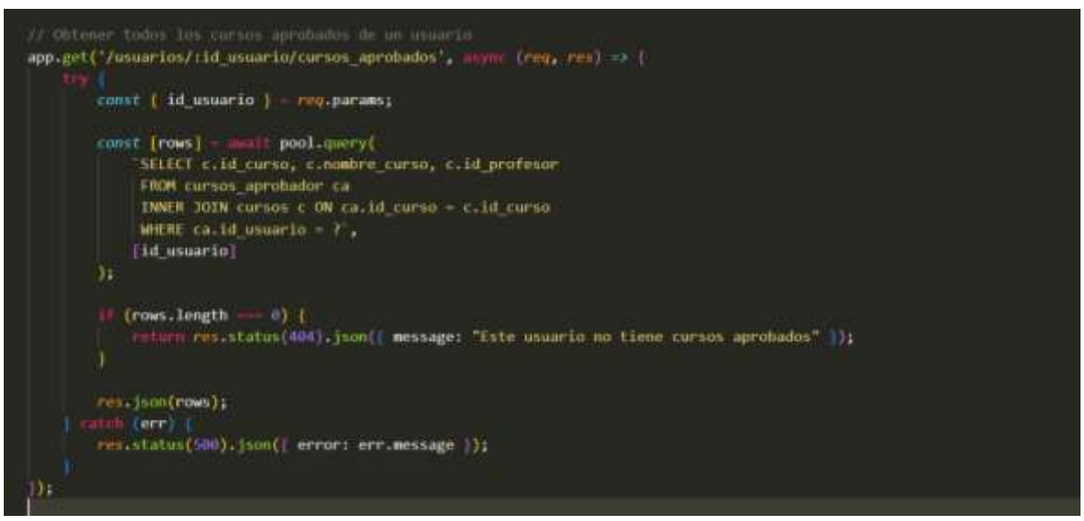
cursos (c): La tabla principal de cursos, de la cual se obtienen detalles como el nombre del curso.

INNER JOIN cursos c ON ca.id_curso = c.id_curso: Esta línea une las tablas basándose en el id_curso, asegurando que solo se recuperen los cursos que existen y que están registrados como aprobados.

WHERE ca.id_usuario = ?: Esta cláusula filtra los resultados para mostrar únicamente los cursos asociados al id_usuario proporcionado.

if (rows.length === 0) { ... }: Si la consulta no devuelve ningún resultado, significa que el usuario no tiene cursos aprobados registrados. En este caso, el servidor responde con un código de estado 404 (Not Found) y un mensaje indicando que el usuario no tiene cursos aprobados.

res.json(rows);: Si se encuentran cursos aprobados, el servidor responde con un array de objetos JSON. Cada objeto representa un curso aprobado y contiene detalles como id_curso, nombre_curso, y id_profesor (que podría ser útil para mostrar quién impartió el curso).



```
// Obtener todos los cursos aprobados de un usuario
app.get('/usuarios/:id_usuario/cursos_aprobados', async (req, res) => {
    try {
        const { id_usuario } = req.params;

        const [rows] = await pool.query(
            `SELECT c.id_curso, c.nombre_curso, c.id_profesor
             FROM cursos_aprobador ca
             INNER JOIN cursos c ON ca.id_curso = c.id_curso
             WHERE ca.id_usuario = ?`,
            [id_usuario]
        );

        if (rows.length === 0) {
            return res.status(404).json({ message: "Este usuario no tiene cursos aprobados" });
        }

        res.json(rows);
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

app.get('/usuarios/:id_usuario/cursos_aprobados', async (req, res) => { ... });: Esta ruta utiliza el id_usuario como un parámetro de ruta para obtener los cursos aprobados de un usuario específico.

const [rows] = await pool.query(...): La consulta SQL utiliza un INNER JOIN entre las tablas cursos_aprobador (la tabla intermedia) y cursos.

INNER JOIN cursos c ON ca.id_curso = c.id_curso: Este JOIN asegura que solo se obtengan los cursos que realmente tienen una relación en la tabla intermedia.

WHERE ca.id_usuario = ?: Esta cláusula filtra los resultados para que solo se incluyan los cursos aprobados por el usuario cuyo id se encuentra en la URL.

if (rows.length === 0) { ... }: Si la consulta no devuelve resultados, se envía una respuesta con un código de estado 404 (Not Found) y un mensaje que indica que el usuario no tiene cursos aprobados.

res.json(rows);: Si se encuentran cursos, se envía una respuesta JSON con un array de objetos que contienen los detalles de los cursos.

Endpoint para Eliminar un Curso Aprobado

app.delete('/usuarios/:id_usuario/cursos_aprobados/:id_curso', async (req, res) => { ... });: Esta ruta DELETE es específica para la eliminación. Utiliza dos parámetros de ruta: el id_usuario para identificar al usuario y el id_curso para identificar el curso que se desea eliminar.

const [result] = await pool.query("DELETE FROM cursos_aprobador WHERE id_usuario = ? AND id_curso = ?", [id_usuario, id_curso]);: Se ejecuta una sentencia SQL DELETE para eliminar la fila de la tabla intermedia que corresponde a la relación entre el usuario y el curso. Se utiliza una cláusula WHERE compuesta para garantizar que la fila correcta sea eliminada.

if (result.affectedRows === 0) { ... }: La propiedad affectedRows en el resultado de la consulta indica cuántas filas fueron eliminadas. Si es 0, significa que no se encontró ninguna relación que coincidiera con los id proporcionados, por lo que el servidor responde con un código de estado 404 (Not Found).

res.json({ message: "Curso aprobado eliminado exitosamente" });: Si la eliminación fue exitosa, se envía un mensaje de éxito al cliente.

```
// Eliminar un curso aprobado de un usuario
app.delete('/usuarios/:id_usuario/cursos_aprobados/:id_curso', async (req, res) => {
    try {
        const { id_usuario, id_curso } = req.params;

        const [result] = await pool.query(
            "DELETE FROM cursos_aprobador WHERE id_usuario = ? AND id_curso = ?",
            [id_usuario, id_curso]
        );

        if (result.affectedRows === 0) {
            return res.status(404).json({ message: "No se encontró el curso aprobado para este usuario" });
        }

        res.json({ message: "Curso aprobado eliminado exitosamente" });
    } catch (err) {
        res.status(500).json({ error: err.message });
    }
});
```

const PORT = process.env.PORT || 3000;: Esta línea define el puerto en el que se ejecutará el servidor. Se usa una práctica común y recomendada: primero, intenta obtener el número de puerto de una variable de entorno (process.env.PORT). Esto es crucial para la puesta en producción de la aplicación, ya que los servicios de hosting (como Heroku, AWS, o Vercel) a menudo asignan un puerto dinámicamente. Si la variable de entorno no está definida (por ejemplo, durante el desarrollo local), se utiliza el valor predeterminado 3000.

app.listen(PORT, () => { ... });: Aquí es donde el servidor realmente se inicia. El método app.listen() de Express le indica a la aplicación que empiece a aceptar peticiones en el puerto especificado. El segundo argumento es una función de callback que se ejecuta una vez que el servidor se ha iniciado exitosamente.

console.log(Servidor corriendo en http://localhost:${PORT});: Esta línea, dentro del callback, muestra un mensaje en la consola. Es una simple confirmación visual para el desarrollador de que el servidor está en funcionamiento y en qué dirección puede ser accedido. Es muy útil para la fase de desarrollo y depuración.

```
// ---------------- INICIO DEL SERVIDOR ----------------
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```