12

LÓGICA DE NEGOCIO AVANZADA

Hasta ahora, has logrado crear la mayor parte de casos de uso para usuarios, pero aún falta mucho trabajo. Todavía queda por hacer la autenticación del usuario, un esquema de cursos con sus respectivas rutas y, por qué no, un endpoint que permita al administrador buscar un usuario determinado.

12.1 BÚSQUEDA DE USUARIOS POR NOMBRE

Para realizar esta tarea, necesitas crear unos cuantos usuarios. Crea unos cinco o diez con algunos nombres y apellidos reales para poder filtrarlos luego. Ahora, recuerda que necesitas una ruta y un controlador.

PASO 1

Ve a **usuarios.routes.js** y crea la ruta (línea 11) e importa una función llamada **obtenerUsuarioPorNombre**, la cual harás a continuación:

router.get('/user/search', obtenerUsuarioPorNombre);

PASO 2

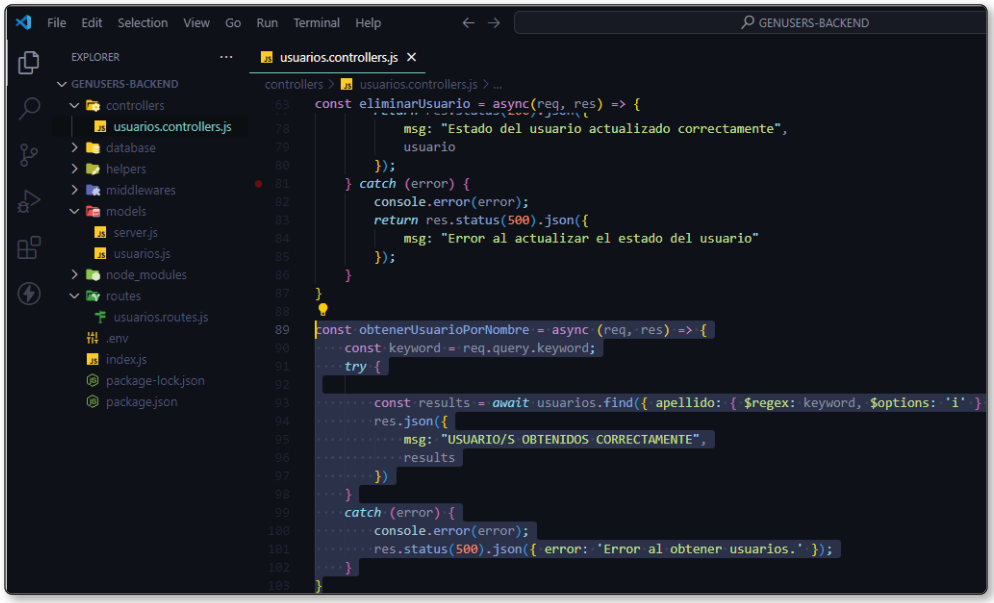
Ve al archivo **usuarios.controllers.js** y crea la función mencionada antes, de la siguiente manera, y no olvides exportarla al final:

```
const obtenerUsuarioPorNombre = async (req, res) => {
    const keyword = req.query.keyword;
    try {

        const results = await usuarios.find({ apellido: { $regex: keyword, $op-
tions: 'i' } });
        res.json({
            msg: "USUARIO/S OBTENIDOS CORRECTAMENTE",
            results
        })
    }
    catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Error al obtener usuarios.' });
    }
}
```

En primer lugar, extrae la palabra clave de la consulta utilizando **req.query.keyword** (es lo que se escribirá por teclado). Luego, realiza una búsqueda en la base de datos utilizando el método **find()** de **Mongoose**, que busca coincidencias parciales del apellido con la palabra clave, ignorando mayúsculas y minúsculas.

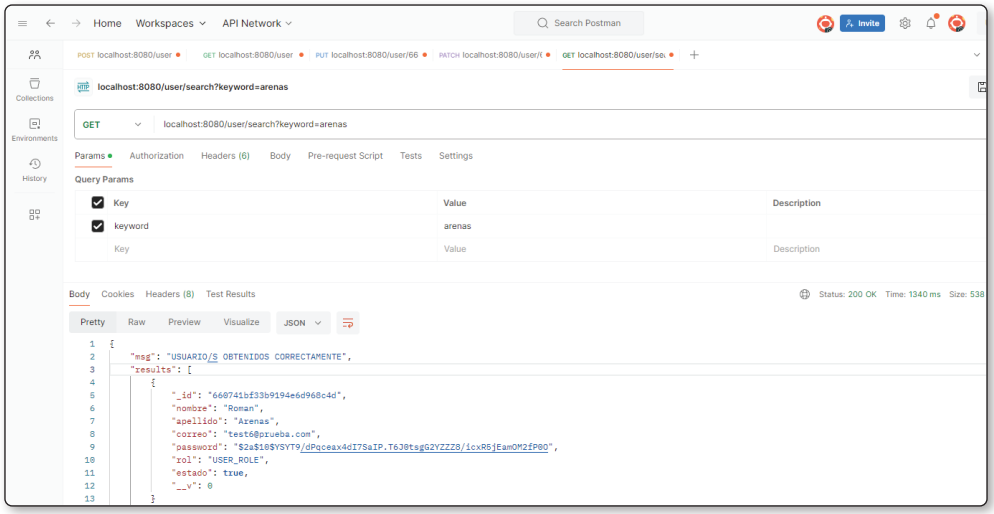
Los resultados de la búsqueda se envían como respuesta al cliente en formato JSON junto con un mensaje para indicar que los usuarios se obtuvieron correctamente. Sin embargo, si ocurre algún error durante el proceso, se captura y se responde al cliente con un código de estado HTTP 500 y un mensaje que indica que hubo un error al obtener los usuarios.

Es importante que tengas en cuenta que este tipo de búsquedas se pueden realizar sobre cualquier campo, por lo que se ha elegido para este ejemplo el apellido.



PASO 3

Probar este endpoint desde **Postman** es un tanto diferente. Debes colocar la URL base seguida de un parámetro, como por ejemplo **localhost:8080/user/search?keyword=arenas**, donde **arenas** sería el apellido que se desea buscar.



12.2 AUTENTICACIÓN DE USUARIOS (GENERACIÓN DE TOKEN)

La **autenticación** es un proceso esencial para verificar la identidad de los usuarios que intentan acceder a recursos protegidos en una aplicación web. Usualmente, este proceso implica la verificación de credenciales, como nombres de usuario y contraseñas, almacenadas en una base de datos. Una vez que se han verificado las credenciales, se proporciona al usuario un token de acceso que actúa como una credencial digital para identificar al usuario en futuras solicitudes.

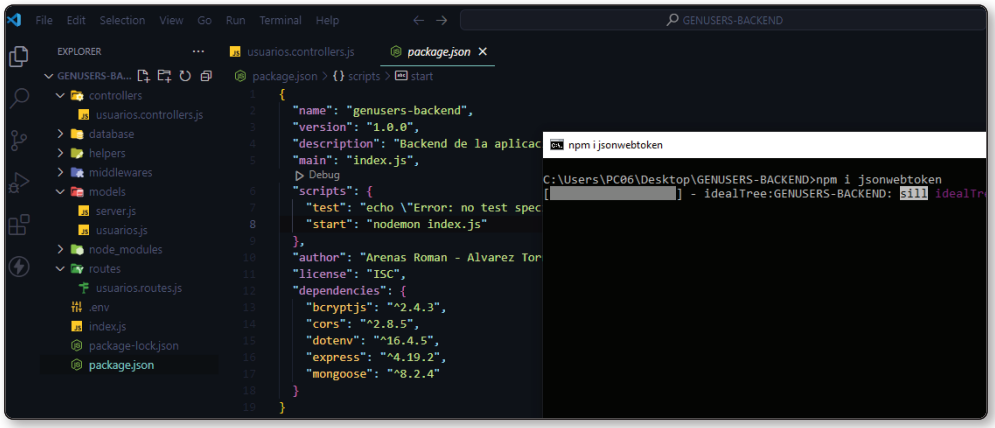
Para el manejo de las credenciales, utilizarás el paquete **jsonwebtoken**.

Una de las ventajas clave de utilizar tokens de acceso web (JSON Web Tokens, JWT) en la autenticación de usuarios en Node.js es su capacidad para mantener el estado de autenticación en el lado del cliente de manera segura y eficiente. Los **JWT** son tokens autónomos que contienen información codificada sobre el usuario y pueden ser verificados fácilmente por el servidor sin necesidad de mantener un estado de sesión en él Esto reduce la carga del servidor y mejora el rendimiento de la aplicación, ya que no es necesario consultar la base de datos en cada solicitud para verificar la sesión del usuario.

Además, los JWT son firmados digitalmente utilizando algoritmos de cifrado, lo que los hace seguros contra manipulaciones. Esto significa que la información almacenada en un JWT no puede ser alterada por usuarios malintencionados sin la clave de firma adecuada. Como resultado, los JWT son una forma segura y escalable de implementar la autenticación de usuarios en aplicaciones Node.js, proporcionando una experiencia de usuario fluida y segura.

PASO 1

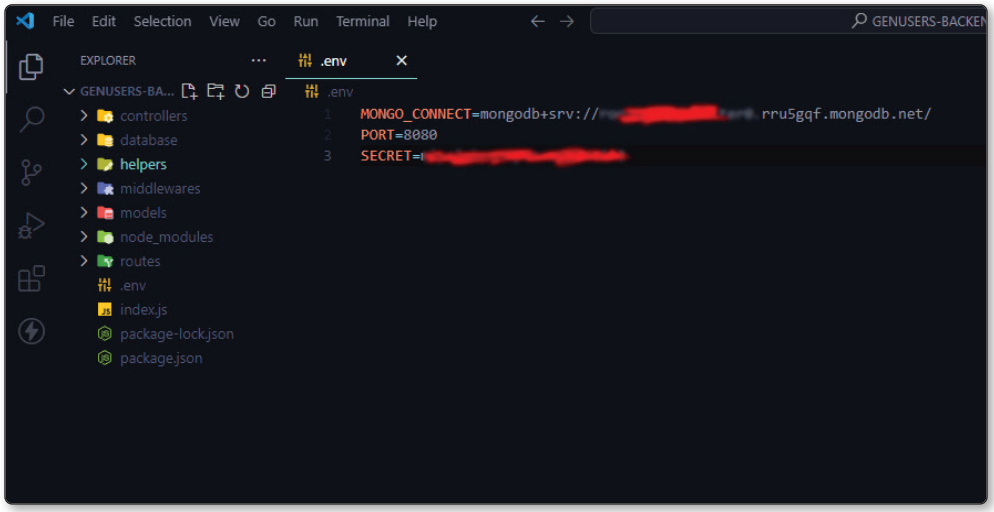
Para comenzar a implementar este estupendo sistema, ve a la terminal, detenla un momento e instala el paquete con **npm i jsonwebtoken**.



PASO 2

Además, necesitarás una clave para firmar los token. Esto puede hacerse desde las variables de entorno, solo asegúrate de que sea una palabra difícil de adivinar. Ve al archivo **.env**, crea una variable llamada **SECRET** y colócale una palabra clave.

Una vez creada la variable de entorno, reinicia el servidor para que los cambios surtan efecto.



PASO 3

Cuando tengas el paquete instalado, dirígete a la carpeta **helpers** y crea un archivo **js** que contendrá el código que generará el token. En este caso, se ha colocado **generarJWT.js**.

Dentro del archivo ubica la siguiente función:

```
const jwt = require('jsonwebtoken')
require('dotenv').config();

const generarJWT = (uid = "")=>{

    return new Promise((resolve, reject) => {
        const payload = {uid};

        jwt.sign(payload, process.env.SECRET ,{
```

```
            expiresIn: “12h”
        }, (err, token)=>{
            if(err){
                console.log(err)
                reject(‘No se pudo generar el token’)
            }
            else{
                resolve(token)
            }
        })
    })
}
module.exports = generarJWT;
```

La función toma un parámetro opcional **uid**, que representa el identificador único del usuario para el cual se está generando el token. Luego, crea un objeto payload que contiene el uid proporcionado como parte de la carga útil del token. A continuación, se utiliza el método **jwt.sign()** para firmar digitalmente el token, utilizando una clave secreta definida en el archivo **.env**. Este método toma tres argumentos: el **payload** por firmar, la clave secreta para la firma y las opciones adicionales, como el tiempo de expiración del token, el cual es de dos horas en este caso, y que podrás ajustar a tu criterio.

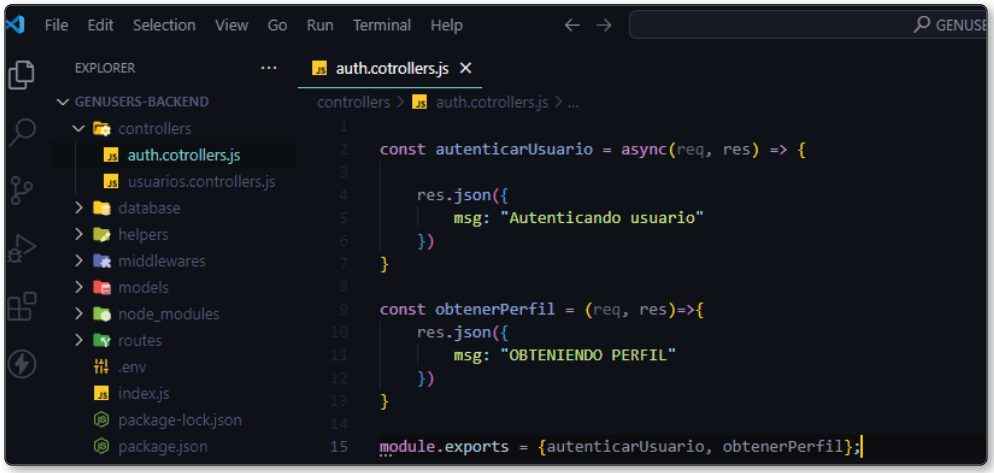
El token generado se devuelve como una promesa. Si la generación del token es exitosa, la promesa se resuelve con el token generado. En caso de que ocurra un error durante la generación, la promesa se rechaza con un mensaje que indique que no se pudo generar el token.

12.3 AUTENTICACIÓN DE USUARIOS (RUTA Y CONTROLADOR)

Si bien podrías manejar la autenticación dentro de las rutas y controladores de usuario, a fin de que el código sea más organizado y entendible, lo harás separado.

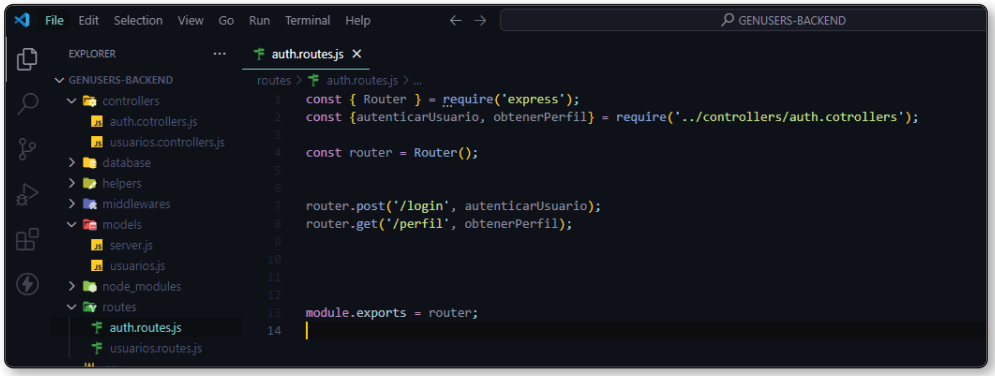
PASO 1

Ve a la carpeta **controllers** y crea un archivo nuevo llamado **auth.controllers.js**. Dentro, deberás crear dos funciones: una para autenticar el usuario y otra para obtener el perfil. Por ahora, solo crea las bases y expórtalas.



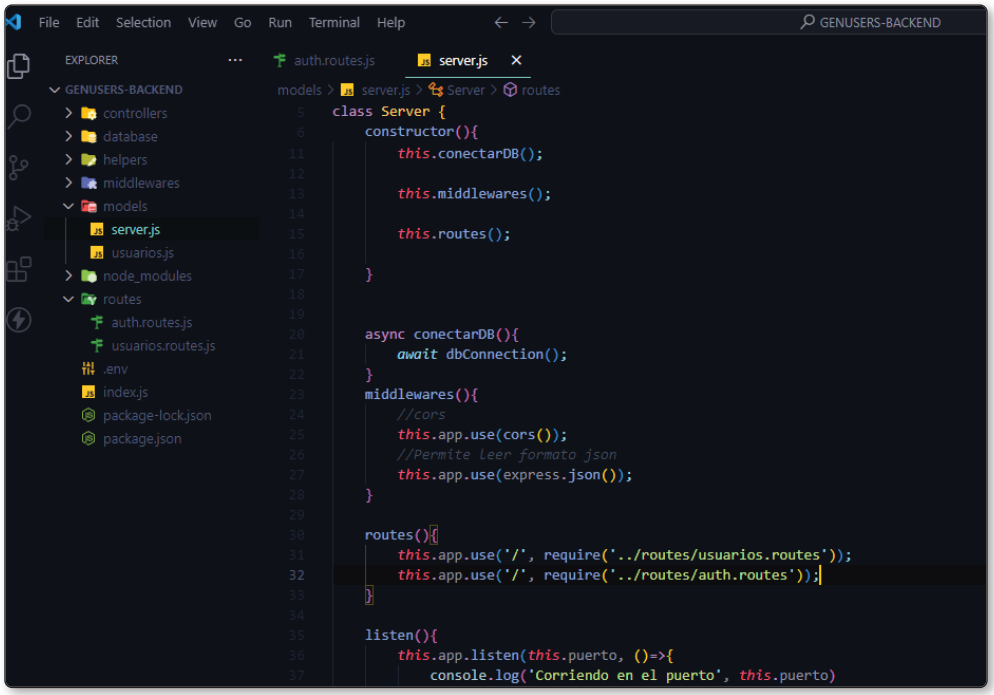
PASO 2

Ahora ve a la carpeta **routes** y crea las dos rutas correspondientes dentro de un nuevo archivo llamado **auth.routes.js**, habiendo hecho con anterioridad la importación de los paquetes necesarios.



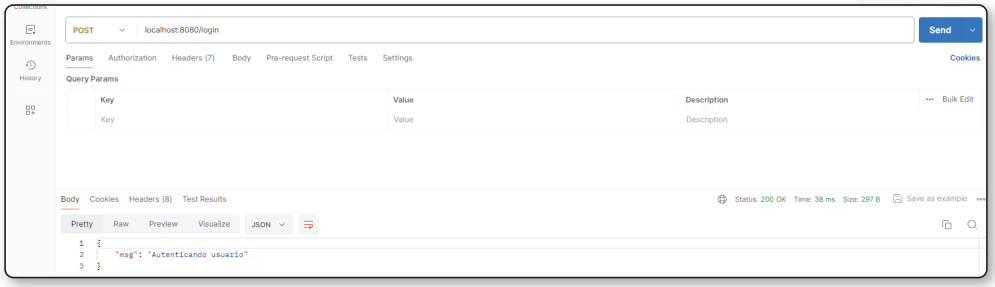
PASO 3

Este archivo debe tener una ruta definida en el servidor. Ve a **server.js** dentro de la carpeta **models** y define la ruta correspondiente (línea 32).



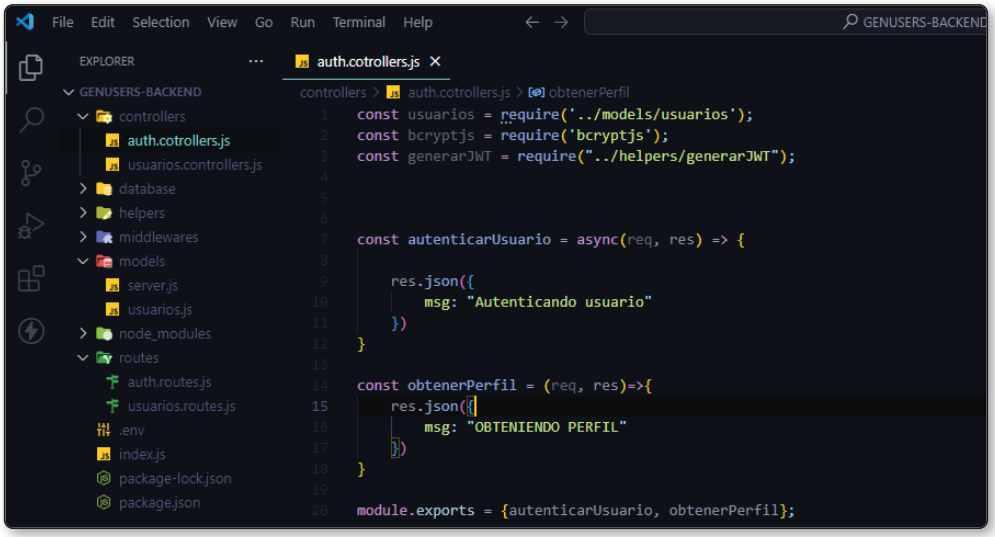
PASO 4

Ejecuta las pruebas para corroborar el buen funcionamiento de los endpoint.



PASO 5

Dentro de **auth.controllers.js**, importa el modelo de **usuarios**, el paquete **bcryptjs** y la función **generarJWT** creada en los pasos anteriores.



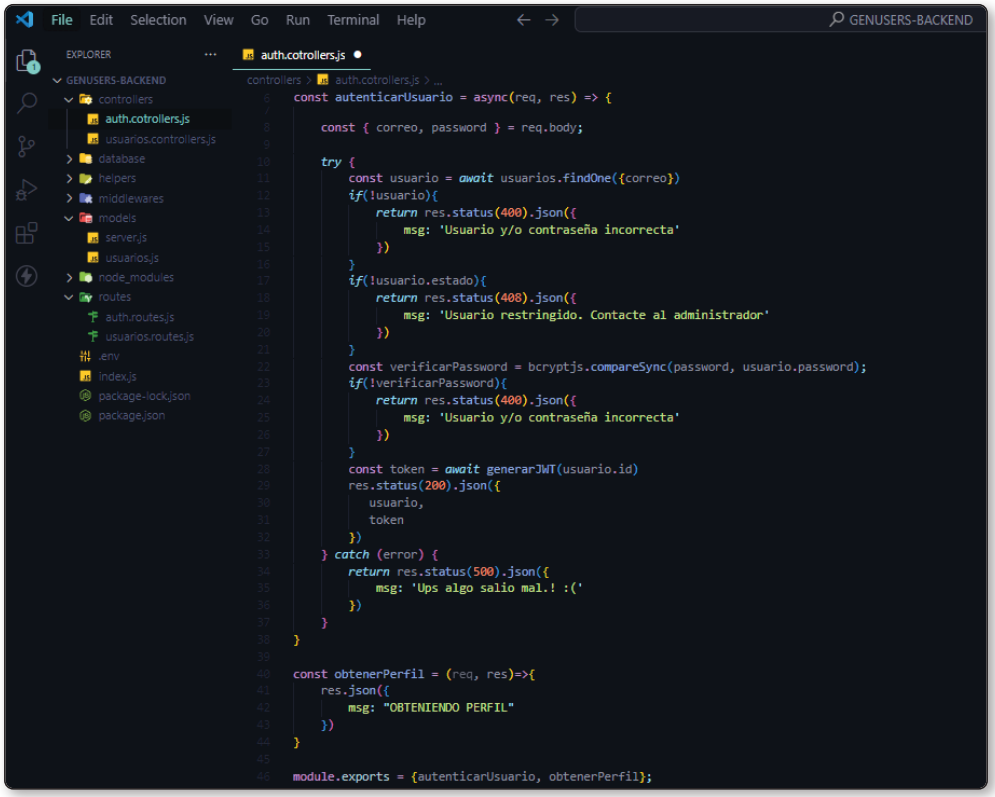
PASO 6

Ahora, dentro de la función **autenticarUsuario**, coloca el siguiente código (líneas 6 a 38):

```
    const autenticarUsuario = async(req, res) => {
    const { correo, password } = req.body;
    try {
        const usuario = await usuarios.findOne({correo})
        if(!usuario){
            return res.status(400).json({
                msg: 'Usuario y/o contraseña incorrecta'
            })
        }
        if(!usuario.estado){
            return res.status(408).json({
                msg: 'Usuario restringido. Contacte al administrador'
            })
        }
        const verificarPassword = bcryptjs.compareSync(password, usuario.pas-
sword);
        if(!verificarPassword){
            return res.status(400).json({
                msg: 'Usuario y/o contraseña incorrecta'
            })
        }
        const token = await generarJWT(usuario.id)
        res.status(200).json({
           usuario,
```

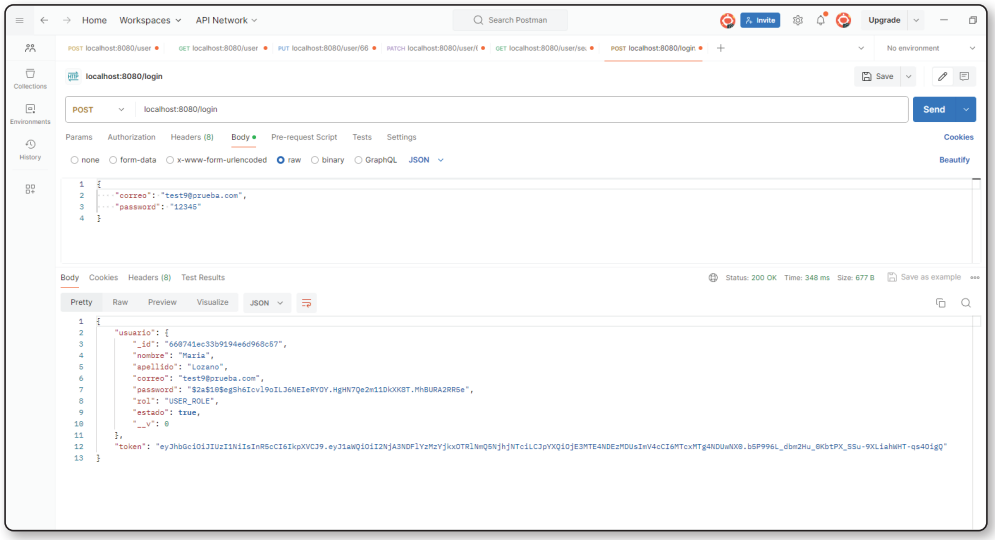
```
            token
        })
    } catch (error) {
        return res.status(500).json({
            msg: 'Ups algo salio mal.! :('
        })
    }
}
```

Este bloque de código se encarga de autenticar a un usuario en un sistema, utilizando su correo electrónico y su contraseña. Comienza extrayendo el correo electrónico y la contraseña de la solicitud HTTP. Luego, busca en la base de datos un usuario que coincida con el correo electrónico proporcionado. Si no se encuentra ningún usuario o el usuario está inactivo, se devuelve un mensaje de error correspondiente. Si se encuentra un usuario activo, se verifica la contraseña proporcionada con la contraseña almacenada en la base de datos utilizando el método **compareSync** ya que esta se encuentra encriptada. Si las contraseñas coinciden, se genera un token JWT para el usuario y se devuelve junto con sus datos en una respuesta de éxito. Este token va a ser el utilizado en el frontend para leer el usuario luego de ser almacenado en el **localStorage**. En caso de cualquier error durante el proceso, se devuelve un mensaje de error genérico



PASO 7

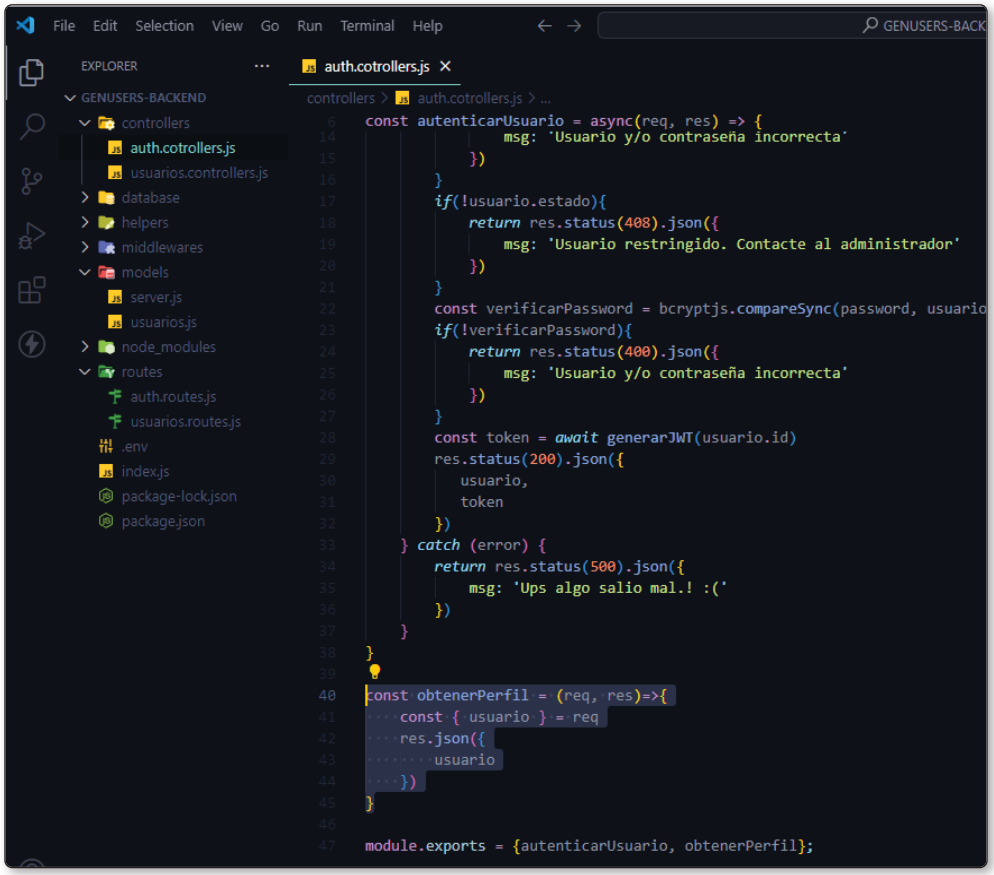
Es momento de probar el endpoint. Ve a **Postman** y haz una petición del tipo **POST** a la URL **localhost:8080/login** y envíale los datos de autenticación de un usuario existente en la base de datos. Esto te devolverá el usuario junto con el token.



PASO 8

Es momento de fabricar el endpoint que permitirá obtener el usuario a través del token de autenticación. Ve a **auth.controllers.js**, específicamente a la función **obtenerPerfil**, y colócale lo siguiente:

```
const obtenerPerfil = (req, res)=>{
    const { usuario } = req
    res.json({
        usuario
    })
}
```



PASO 9

Si haces la petición desde **Postman** a **localhost:8080/perfil**, verás que solo retorna un objeto vacío. Esto se debe a que no hay ninguna lectura del token proporcionado o mejor dicho este no está llegando a la petición por ninguna parte.

Aquí es donde entrarán en juego los middlewares. Ve a la carpeta **middlewares**, crea un archivo llamado **checkAuth.js** y coloca el siguiente código dentro:

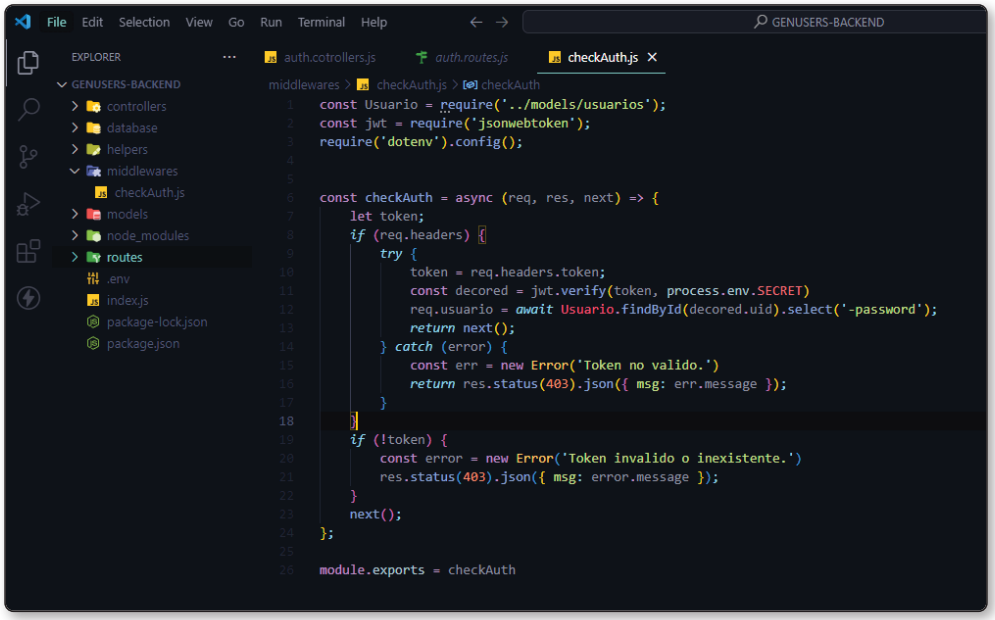
```
const Usuario = require('../models/usuarios');
const jwt = require('jsonwebtoken');
require('dotenv').config();
```

```
const checkAuth = async (req, res, next) => {
    let token;
    if (req.headers) {
        try {
            token = req.headers.token;
            const decored = jwt.verify(token, process.env.SECRET)
            req.usuario = await Usuario.findById(decored.uid).select('-pas-
sword');
            return next();
        } catch (error) {
            const err = new Error('Token no valido.')
            return res.status(403).json({ msg: err.message });
        }
    }
    if (!token) {
        const error = new Error('Token invalido o inexistente.')
        res.status(403).json({ msg: error.message });
    }
    next();
};

module.exports = checkAuth
```

El middleware comienza intentando extraer el token del encabezado de la solicitud **req.headers.token**. Si se encuentra un token, se debe verificar su validez utilizando la función **jwt.verify()**, que utiliza la clave secreta definida en el archivo **.env**. Si el token es válido, se extrae el identificador único (uid) del usuario codificado en el token y se busca en la base de datos utilizando el modelo de **usuarios** para obtener la información del usuario correspondiente. La información se adjunta al objeto de solicitud **req.usuario** para que esté disponible en los controladores posteriores.

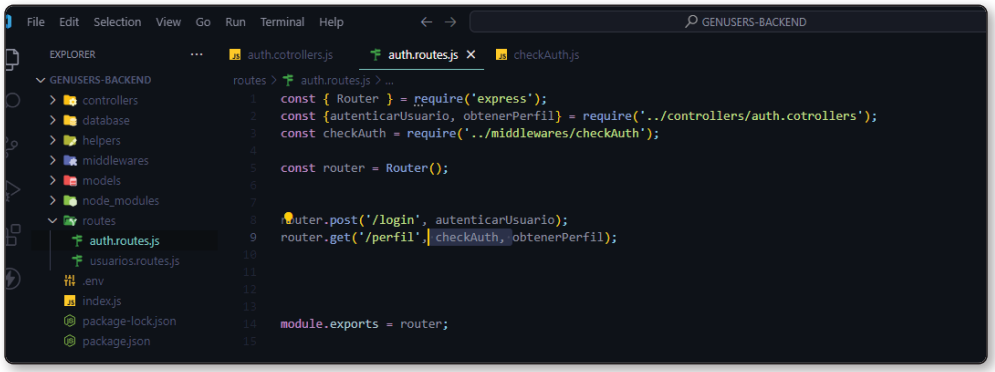
Si se produce algún error durante la verificación del token, se captura y se devuelve un mensaje de error para indicar que el token no es válido. Si no se encuentra ningún token en los encabezados de la solicitud, se devuelve un mensaje de error que indica que el token es inválido o inexistente.

Por último, si no se localiza ningún error y el token es válido, el middleware llama a la función **next()** para pasar la solicitud al siguiente middleware o controlador en la cadena de middleware.



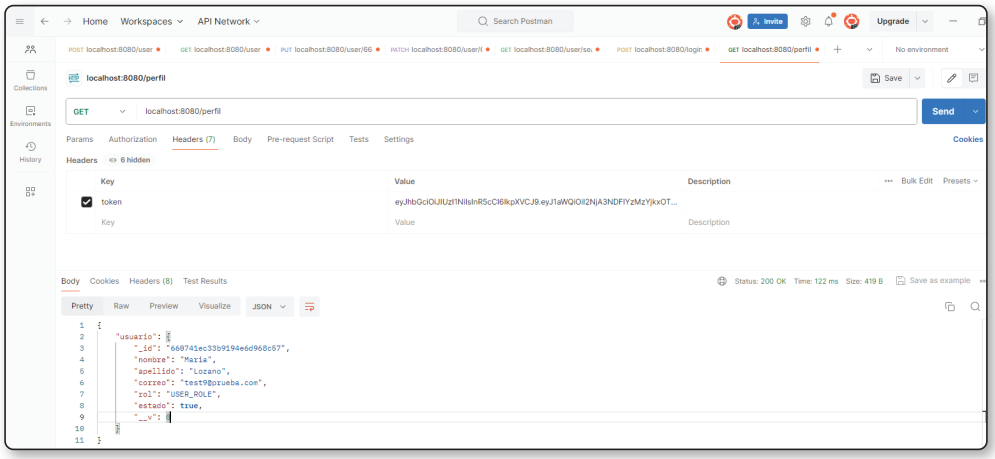
PASO 10

Ahora ve a **auth.routes.js** y coloca el middleware dentro de la petición para obtener el perfil. No te olvides de importarlo.



PASO 11

Para probar este endpoint, dirígete a **Postman**, loguéate con un usuario existente, copia el token y envíaselo a la petición del perfil a través de los **Headers** como se muestra en la imagen. Si todo es correcto, te retornará el usuario autenticado.

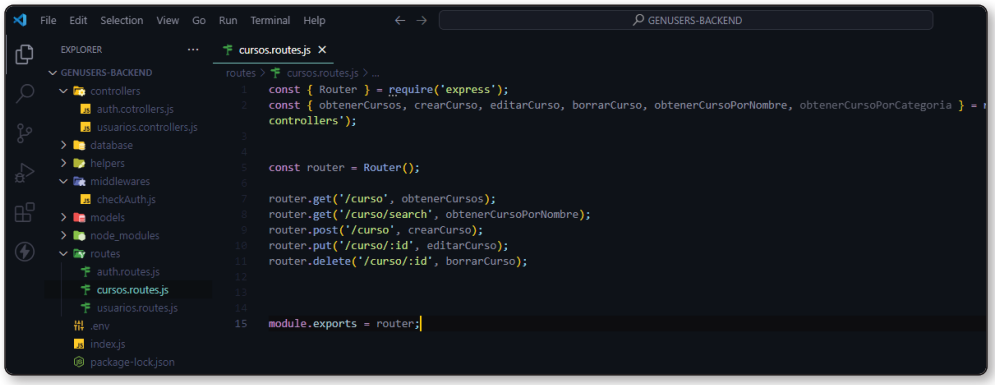


12.4 ESQUEMA DE CURSOS

Es momento de armar todo el sistema de cursos junto con sus respectivas rutas y controladores. Esto será muy similar a lo del capítulo anterior. Aquí puedes ajustar las rutas a tu conveniencia, pero, a modo didáctico, implementarás las más comunes y, luego, decidirás si necesitas alguna más.

PASO 1

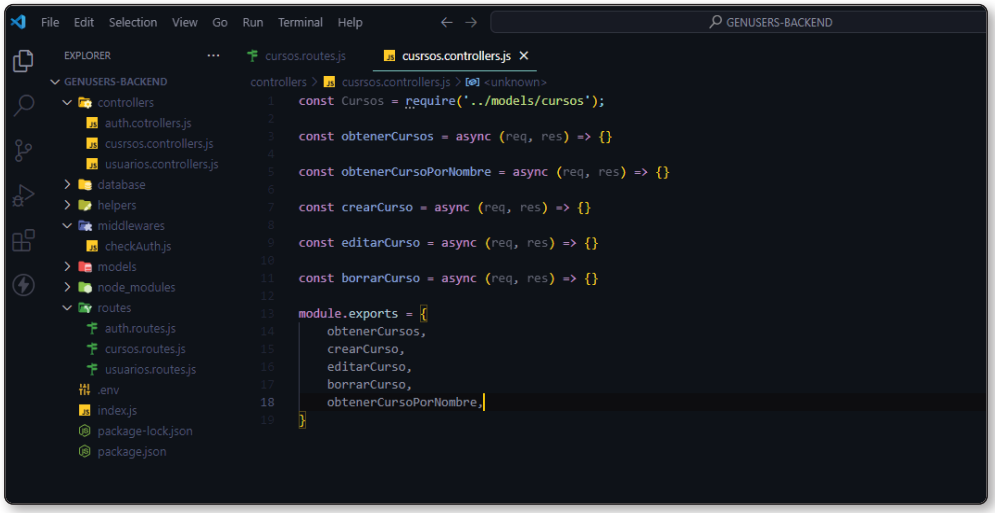
Ve a la carpeta **routes** y crea un archivo llamado **cursos.routes.js**. Allí crea la estructura correspondiente a las rutas.



PASO 2

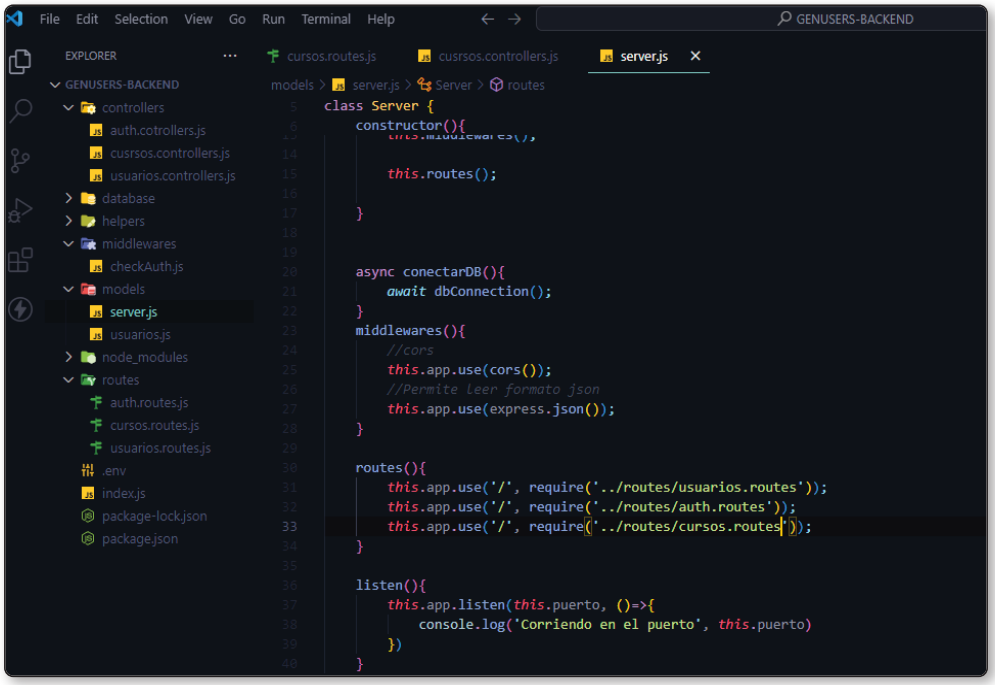
Como puedes observar, hay varias funciones que están siendo llamadas, pero sin haberse definido, incluso aún no has creado el archivo **cursos.controllers.js**.

Ve a la carpeta **controllers** y crea dicho archivo junto con las funciones que se requieren. Por ahora, solo déjalas sin contenido.



PASO 3

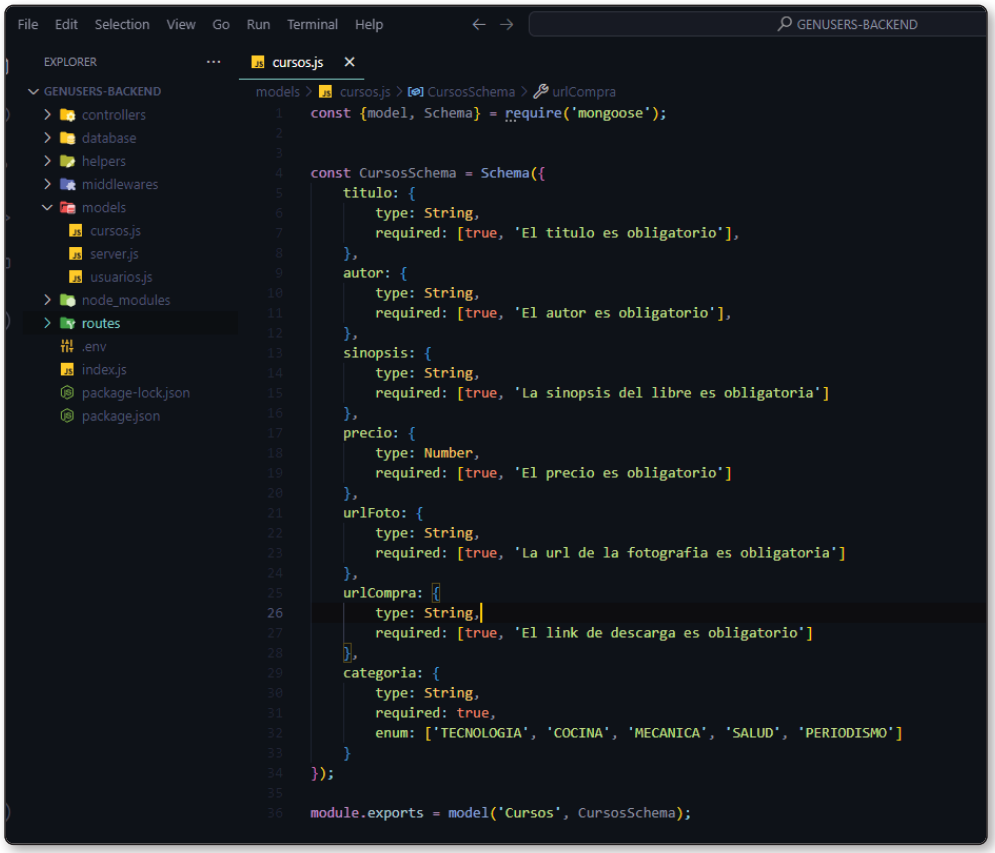
A continuación, enlaza las rutas al servidor. Ve al archivo **server.js** y crea una ruta para los cursos (línea 33).



Es momento de crear el modelo de cursos que te permitirá tener la receta para su construcción.

PASO 1

Ve a la carpeta **models** y crea el modelo en un archivo llamado **cursos.js**. Dentro, puedes moldearlo a tu gusto requiriendo los datos que creas necesarios en tu aplicación.



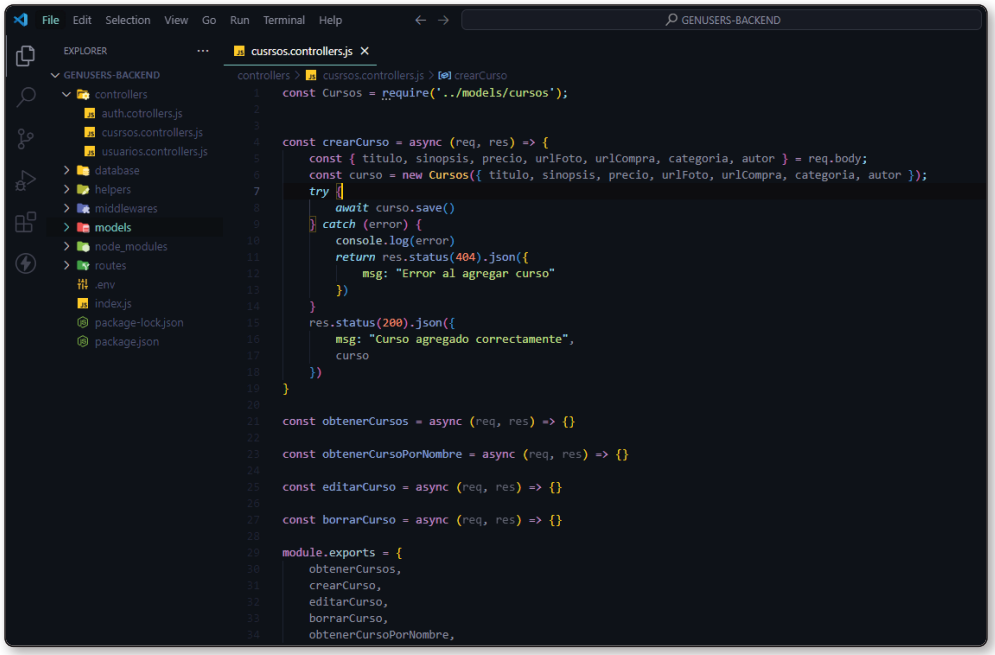
PASO 2

Ahora que el esquema está en curso, ve al archivo **cursos.controllers.js** y modifica la función **crearCurso** de la siguiente manera:

```
const crearCurso = async (req, res) => {
    const { titulo, sinopsis, precio, urlFoto, urlCompra, categoria, autor } =
req.body;
    const curso = new Cursos({ titulo, sinopsis, precio, urlFoto, urlCompra,
```

```
categoria, autor });
    try {
        await curso.save()
    } catch (error) {
        console.log(error)
        return res.status(404).json({
            msg: “Error al agregar curso”
        })
    }
    res.status(200).json({
        msg: “Curso agregado correctamente”,
        curso
    })
}
```

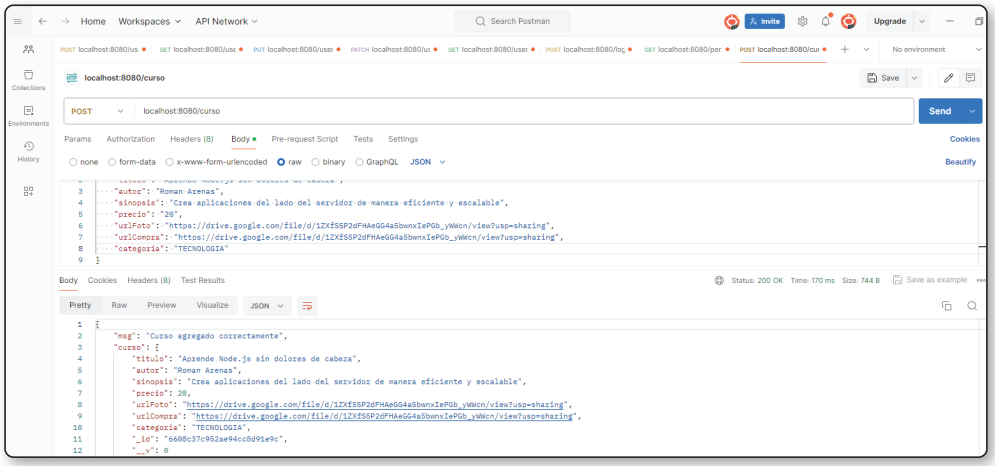
Esto es similar a lo realizado en la creación de usuarios. El controlador espera recibir los datos del curso en el cuerpo de la solicitud HTTP, incluyendo el título, la sinopsis, el precio, la URL de la foto, la URL de compra, la categoría y el autor del curso. Luego, utiliza estos datos para crear una nueva instancia del modelo **Cursos**. Intenta guardar el curso creado en la base de datos utilizando el método **save()**. Si se produce algún error durante este proceso, el controlador captura el error y devuelve una respuesta de estado 404 con un mensaje que indica que ocurrió un error al agregar el curso. En caso de éxito, el controlador devuelve una respuesta de estado 200 con un mensaje para avisar que el curso se agregó correctamente, junto con los datos del curso agregado.



PASO 3

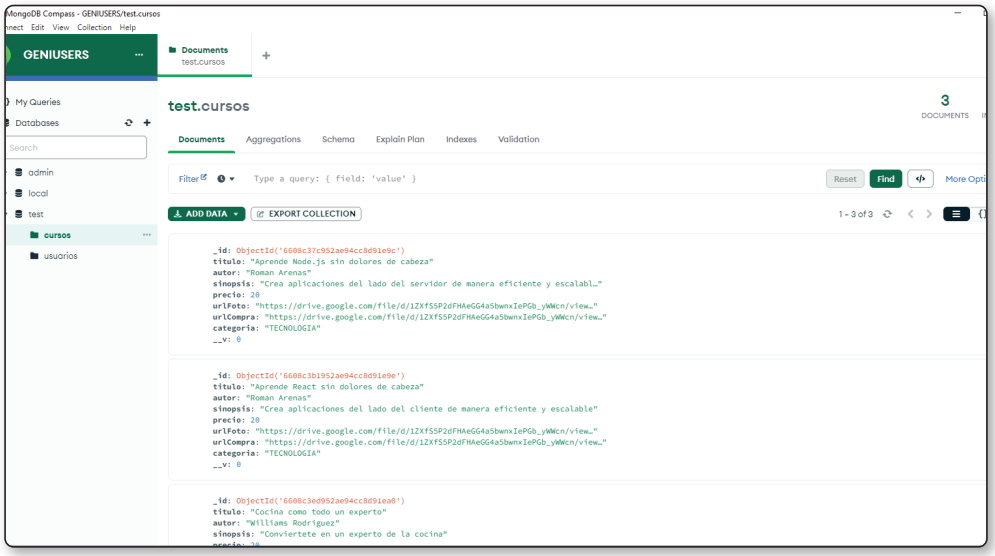
Para probar el endpoint, ve a **Postman**, crea una petición del tipo **POST** apuntando a la URL **localhost:8080/curso** y envíale en el body los datos que requiere el modelo.

Crea varios cursos para tener en la base de datos.



PASO 4

Si observas la base de datos, verás que se ha creado la nueva colección con los cursos agregados.

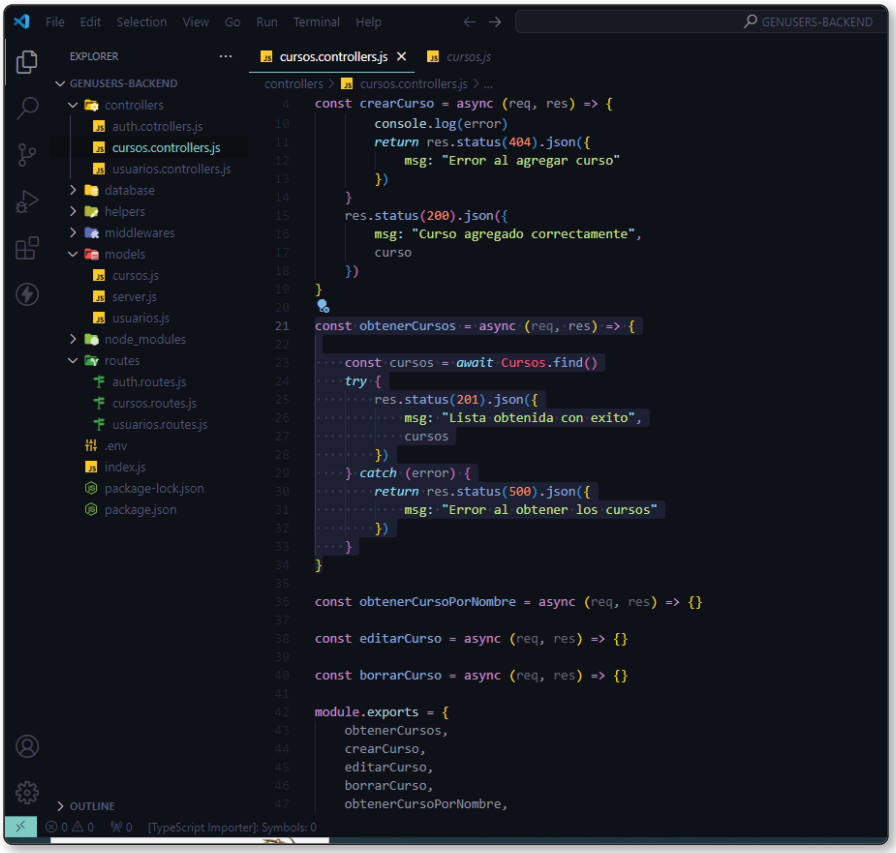


PASO 5

Ahora es momento del endpoint que obtiene los cursos. Ve a la función **obtenerCursos** y modifícala de la siguiente manera.

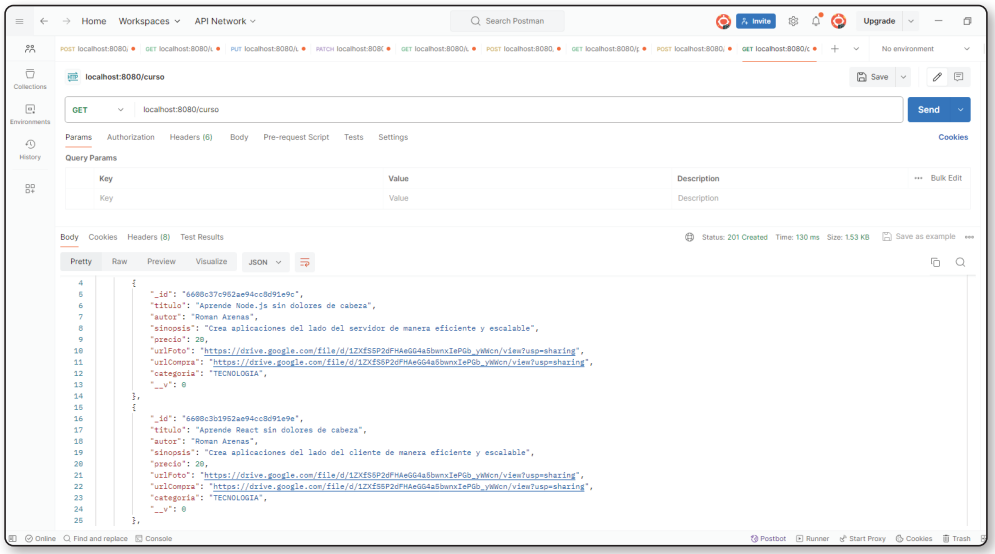
```
const obtenerCursos = async (req, res) => {

    const cursos = await Cursos.find()
    try {
        res.status(201).json({
            msg: “Lista obtenida con exito”,
            cursos
        })
    } catch (error) {
        return res.status(500).json({
            msg: “Error al obtener los cursos”
        })
    }
}
```

Aquí se utiliza el método **find()** para obtener la colección completa de cursos.



PASO 6

En **Postman**, la petición es del tipo **GET** hacia la URL de curso.

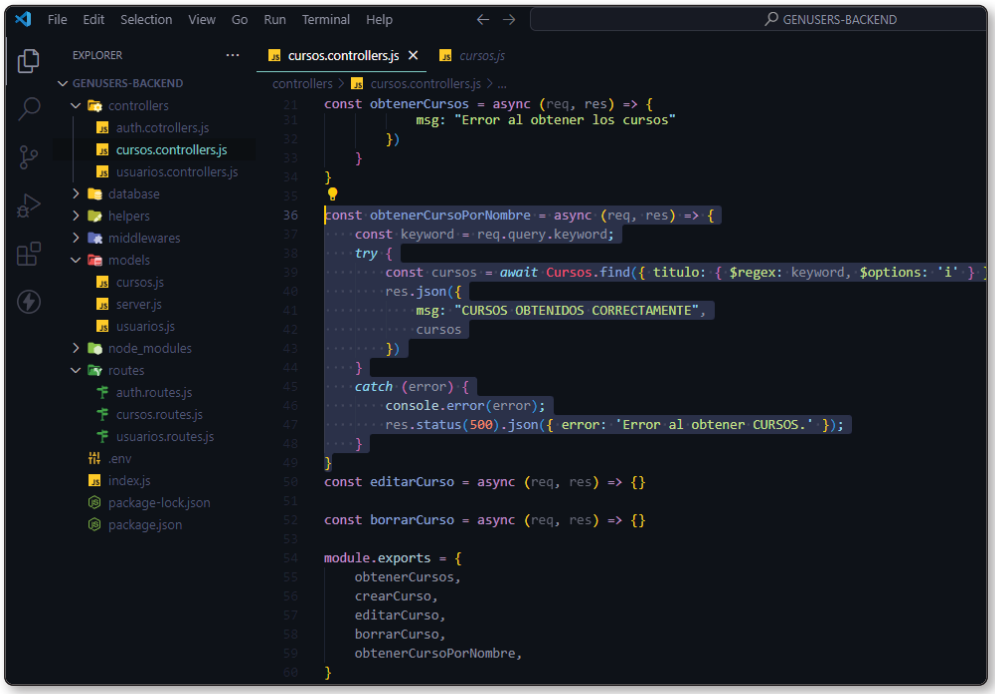


PASO 7

Es el turno del endpoint que permitirá obtener el curso por título. Ve a la función **obtenerCursoPorNombre** y modifícala de la siguiente manera:

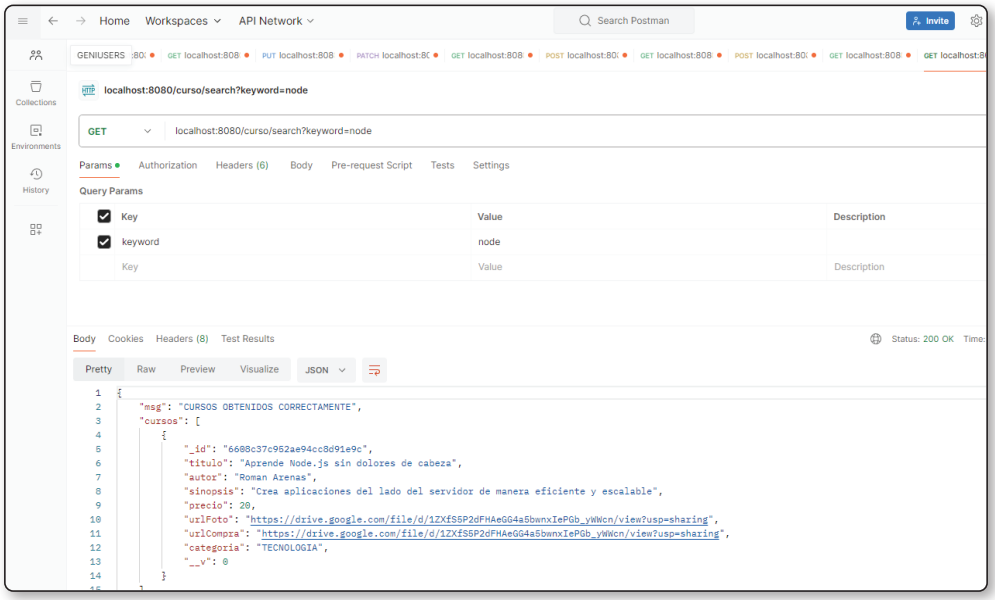
```
const obtenerCursoPorNombre = async (req, res) => {
    const keyword = req.query.keyword;
    try {
        const cursos = await Cursos.find({ titulo: { $regex: keyword, $options:
'i' } });
        res.json({
            msg: "CURSOS OBTENIDOS CORRECTAMENTE",
            cursos
        })
    }
    catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Error al obtener CURSOS.' });
    }
}
```

Esto es similar a lo realizado en los controladores de usuarios.



PASO 8

En **Postman**, deberás pasarle la palabra en el keyword y obtendrás el resultado deseado.



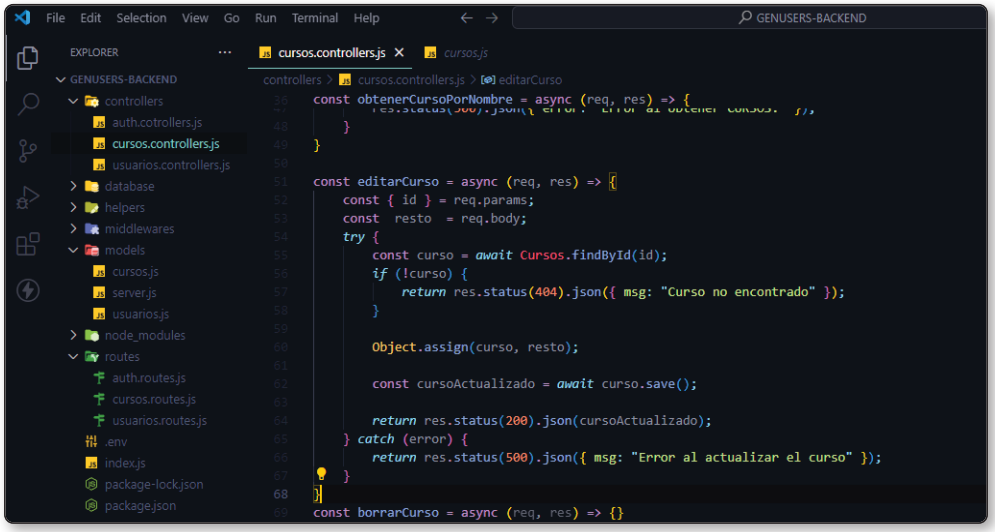
PASO 9

Es el turno de la edición. Ve a la función **editarCurso** y modifícala de la siguiente manera:

```
const editarCurso = async (req, res) => {
    const { id } = req.params;
    const  resto  = req.body;
    try {
        const curso = await Cursos.findById(id);
        if (!curso) {
            return res.status(404).json({ msg: “Curso no encontrado” });
        }

        Object.assign(curso, resto);
        const cursoActualizado = await curso.save();
        return res.status(200).json(cursoActualizado);
    } catch (error) {
        return res.status(500).json({ msg: “Error al actualizar el curso” });
    }
}
```

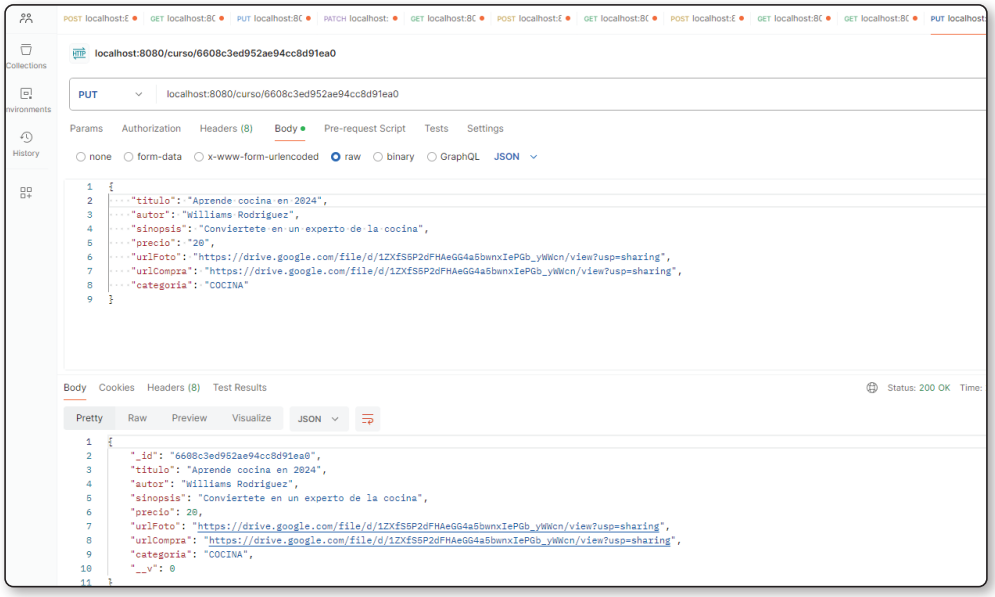
Al recibir el ID del curso a través de los parámetros de la solicitud y los nuevos datos del curso en el cuerpo de la solicitud, primero busca el curso correspondiente en la base de datos utilizando el método **findById()** de Mongoose. Si el curso existe, actualiza sus datos con los proporcionados en el cuerpo de la solicitud utilizando **Object.assign()** y, luego, guarda los cambios con el método **save()**. En caso de éxito, devuelve una respuesta con el curso actualizado; de lo contrario, muestra un mensaje de error si no se encuentra el curso o se produce un error durante el proceso de actualización.

Este proceso cumple la misma función que la edición de usuarios, pero con una sintaxis diferente.



PASO 10

Para probar el endpoint, crea una petición **PUT** en **Postman**, pásale el ID correspondiente al curso y modifica lo que requieras.



PASO 11

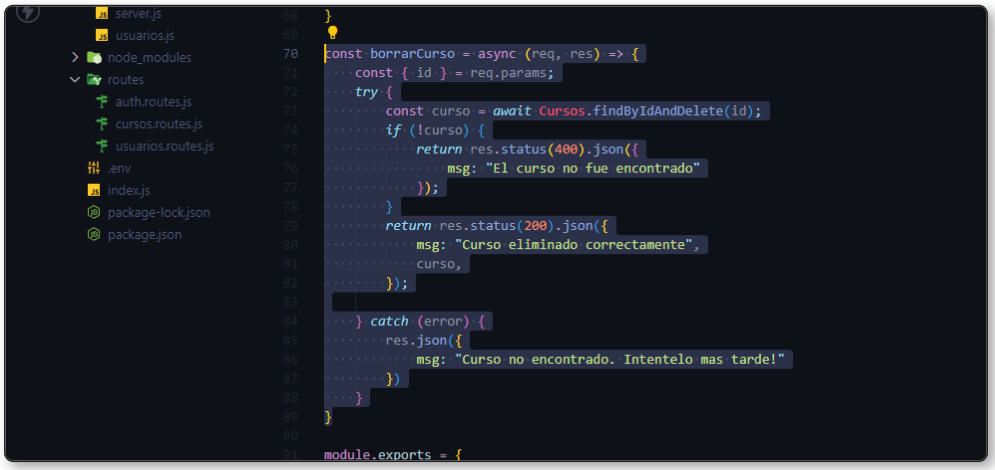
Por último, la función **eliminarCurso**, que directamente destruirá el registro a diferencia del de **usuarios** que solo cambiaba el estado a **false** provocando una especie de "baneo", aquí no será necesario.

Ve a la función **borrarCurso** y modifícala de la siguiente manera:

```
const borrarCurso = async (req, res) => {
    const { id } = req.params;
    try {
        const curso = await Cursos.findByIdAndDelete(id);
        if (!curso) {
            return res.status(400).json({
                msg: "El curso no fue encontrado"
            });
        }
        return res.status(200).json({
            msg: "Curso eliminado correctamente",
            curso,
        });

    } catch (error) {
        res.json({
```

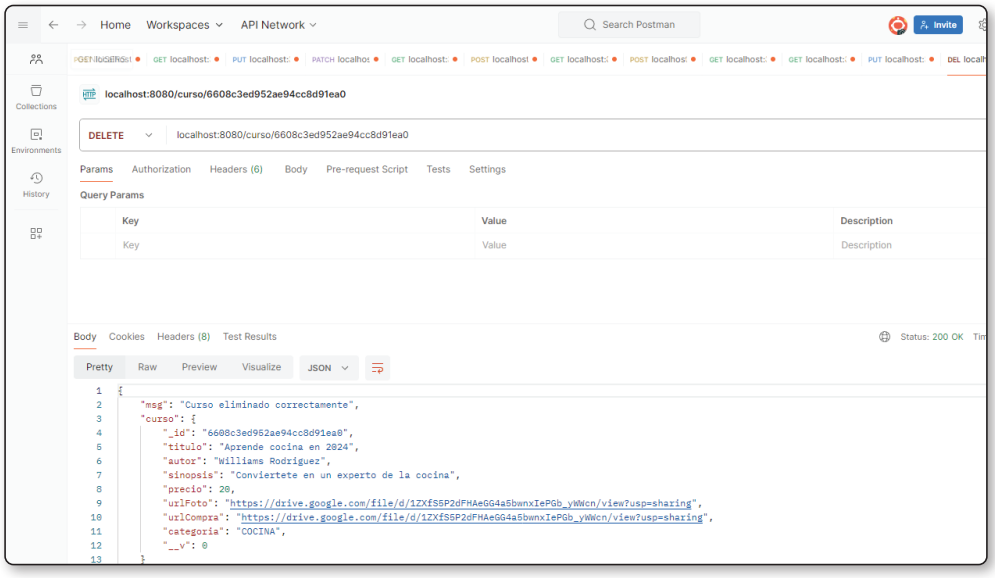
```
            msg: “Curso no encontrado. Intentelo mas tarde!”
        })
    }
}
```

El controlador extrae el ID del curso de los parámetros de la solicitud **req. params**, y luego utiliza el método **findByIdAndDelete()** de Mongoose para buscar y eliminar el curso correspondiente en la base de datos.



PASO 12

Ahora, haz la prueba en **Postman**. Crea una petición del tipo **DELETE** y pásale el ID de un curso válido.



12.5 ACTIVIDADES

A continuación se presentan las preguntas y los ejercicios que deberías saber responder y resolver para considerar aprendido el capítulo.

12.5.1 Test de autoevaluación

1. *¿Cuál es el propósito de la búsqueda de usuarios por nombre en una aplicación Node.js?*
2. *¿Qué método de Mongoose se utiliza para buscar usuarios por un campo específico en la base de datos?*
3. *¿Cuál es la diferencia entre los parámetros* **req.body** *y* **req.query** *en Express?*
4. *¿Por qué es importante capturar y manejar los errores durante la búsqueda de usuarios en una aplicación Node.js?*
5. *¿Qué ventajas ofrecen los tokens de acceso web (JWT) en la autenticación de usuarios en comparación con otros métodos tradicionales de autenticación?*
6. *¿Cuál es el propósito de la variable de entorno* **SECRET** *en la generación de tokens JWT?*
7. *¿Cómo se manejan los errores durante la generación de un token JWT en Node.js?*
8. *¿Cuál es la duración predeterminada de un token JWT generado en el ejemplo proporcionado?*
9. *¿Por qué se recomienda reiniciar el servidor después de configurar una variable de entorno en Node.js?*
10. *¿Cuál es el papel de la función* **jwt.sign()** *en la generación de tokens JWT en Node.js?*

12.5.2 Ejercicios Prácticos

1. *Modifica la función* **obtenerUsuarioPorNombre** *para que busque usuarios por su nombre en lugar de su apellido.*
2. *Implementa una nueva función en el controlador de usuarios para actualizar la información de un usuario existente en la base de datos.*

3. *Crea una ruta y un controlador adicionales para permitir la eliminación de usuarios de la base de datos.*

4. *Extiende la función* **generarJWT** *para incluir más información en el token, como el nombre de usuario y su rol.*

5. *Implementa un middleware en Express para validar la autenticidad de los tokens JWT en las solicitudes entrantes antes de permitir el acceso a recursos protegidos.*