Creacion de un API con node, express y mongo CRUD y validaciones

lección	apuntes
Etiquetas	resúmenes
materia	Libre configuración
Miguel Ángel	Miguel Ángel

Inicializa tu proyecto Node.js usando `npm init` y sigue las instrucciones para configurar el archivo `package.json`.

A continuacion ponemos el siguiente comando en la terminal

```
sudo npm init -y
```

Y se creara el archivo package.json, posteirmente instalaermos nuestras dependencias necesarias: Express, Mongoose (para MongoDB), dotenv (para variables de entorno), y cors (para manejar CORS).

```
sudo npm install express mongoose dotenv cors
```

Ejecutar el siguiente comando en la terminal en la carpeta de tu proyecto para instalar 'express-validator':

```
sudo npm install express-validator
```

Tras esto se instalara todo lo necesario para realizar un peticion.

Configuración de Variables de Entorno (.env)

Crea un archivo `.env` en la raíz del proyecto y define las variables necesarias.

```
MONGO_CNN = "mongodb+srv://mianreme:LYtte_1325S@cluster0.qrbwtmu.mongodb.net/repaso?retryWrites=true&w=majority"
PORT = 3001
```

Es importante que en la variable MONGO_CNN especifiques el nombre de la bbdd donde se iran almacenado los datos de la bbdd. En el siguiente ejemplo se especifica.

```
MONGO_CNN = "mongodb+srv://mianreme:LYtte_1325S@cluster0.qrbwtmu.mongodb.net/nuevo_proyecto?retryWrites=true&w=majority"
```

A continuacion en el archivo app.js donde generalmente se encarga de configurar y arrancar el servidor. Veamos cada sección del archivo:

Importación de Dependencias:

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

Aquí se están importando los módulos necesarios. `express` se utiliza para crear y configurar la aplicación web, `mongoose` es un ODM (Object Data Modeling) para MongoDB, y `cors` permite gestionar las solicitudes de recursos de origen cruzado (CORS).

Creación de la Aplicación:

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

Aquí se crea una instancia de la aplicación Express.

Configuración del Entorno y Middleware:

```
require('dotenv').config();
mongoose.set("strictQuery", false);
app.use(cors());
app.use(express.json());
```

- `dotenv` se utiliza para cargar las variables de entorno desde un archivo `.env`.
- `mongoose.set("strictQuery", false)` desactiva la comprobación estricta de consultas en Mongoose.
- `app.use(cors())` habilita el middleware CORS para permitir solicitudes desde otros dominios.
- `app.use(express.json())` permite que la aplicación utilice JSON como formato de datos en las solicitudes y respuestas.

Conexión a la Base de Datos:

```
async function main() {
    try {
        await mongoose.connect(process.env.MONGO_CNN);
        console.log(`El servidor está escuchando en el puerto ${process.env.PORT}`);
    } catch (err) {
        console.log(err);
    }
}
main().catch((err) => console.log(err));
```

1. Aquí se define una función `main` que establece la conexión a la base de datos MongoDB utilizando la URL proporcionada en la variable de entorno `MONGO_CNN, que se encuentra en el archivo .env`

Inicio del Servidor:

```
app.listen(process.env.PORT, () => {
    console.log(`El servidor en funcionamiento en el puerto ${process.env.PORT}`);
});
```

1. Esta parte inicia el servidor en el puerto especificado en la variable de entorno `PORT`.

En resumen, el archivo `app.js` establece la configuración básica para una aplicación Express, conecta a la base de datos, y luego inicia el servidor en el puerto especificado.

A continuacion se muestra el codigo completo de app.js

```
// Importamos dependencias
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const app = express();

require('dotenv').config();
mongoose.set("strictQuery", false);
// Habilitar CORS para manejar las solicitudes
app.use(cors());
app.use(express.json());

// Función que conecta a la base de datos
async function main() {
    try {
        await mongoose.connect(process.env.MONGO_CNN);
        console.log(`El servidor está escuchando en el puerto ${process.env.PORT}`);
    } catch (err) {
        console.log(err);
    }
}

// Llamamos a la función
main().catch((err) => console.log(err));

app.listen(process.env.PORT, () => {
    console.log(`El servidor en funcionamiento en el puerto ${process.env.PORT}`);
});
```

A continuacion, procedemos a instalar nodemon en el caso de que no la tengamos instalada.

```
sudo npm install -g nodemon
```

Si la tenemos instalada procedemos a escribir el siguiente comando

```
nodemon app.js
```

La aplicación seguirá ejecutándose y escuchando en el puerto que previamente indicamos. Si deseas detener la aplicación, puedes hacerlo presionando `Ctrl + C` en la terminal donde se está ejecutando nodemon

El siguiente paso sera la creacion de un modelo, para ello crearemos la carpeta models y dentro de ella creamos nuestro primero modelo

Models

Importar Módulos de Mongoose:

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
```

Importamos las dependencias necesarias. `mongoose` es una biblioteca de modelado de objetos MongoDB para Node.js, y `Schema` nos permite definir la estructura del documento.

Definir el Esquema del modelo:

En este ejemplo estamos creando un modelo de coche

```
const CocheSchema = new Schema({
    marca: {
        type: String,
        required: true,
    },
    modelo: {
        type: String,
        required: true,
    },
    anio: {
        type: Number,
        required: true,
    },
    color: {
        type: String,
        required: true,
    },
});
```

- Creamos un nuevo esquema (`CocheSchema`) utilizando el constructor `Schema`. Este esquema define la estructura de nuestros documentos de coche en la base de datos.
- El coche tiene propiedades como `marca`, `modelo`, `año`, y `color`.

Exportar el Modelo:

```
module.exports = mongoose.model('Coche', CocheSchema);
```

Exportamos el modelo creado a partir del esquema. Esto nos permitirá interactuar con la colección de coches en la base de datos.

A continuacion procedemos a crear la carpeta routes y crear dentro de ellar el archivo **cocheRoutes.js**

Routes

En `cocheRoutes.js`, se definirán las rutas específicas relacionadas con la entidad "coche". Cada ruta estará asociada a un controlador que manejará las operaciones correspondientes (por ejemplo, crear, leer, actualizar, eliminar).

```
const express = require('express');
const router = express.Router();
const { check } = require('express-validator');
const { validateCocheData } = require('../middleware/cocheMiddleware');
const cocheController = require('../controllers/cocheController');

// Obtener todos los coches
router.get('/', cocheController.getAllCoches);
```

```
// Obtener un coche por ID
router.get('/:id', cocheController.getCocheById);

// Crear un nuevo coche (aplicamos el middleware de validación antes de llamar a la función del controlador)
router.post('/', [
    check("marca", 'Marca is required').not().isEmpty(),
    check("modelo", 'Modelo is required').not().isEmpty(),
    check("anio", 'Año is required').not().isEmpty(),
    check("color", 'Color is required').not().isEmpty(),
    validateCocheData
], cocheController.createCoche);

// Actualizar un coche por ID (aplicamos el middleware de validación antes de llamar a la función del controlador)
router.put('/:id', [
    check("marca", 'Marca is required').not().isEmpty(),
    check("modelo", 'Modelo is required').not().isEmpty(),
    check("anio", 'Año is required').not().isEmpty(),
    check("color", 'Color is required').not().isEmpty(),
    validateCocheData
], cocheController.updateCoche);

// Eliminar un coche por ID
router.delete('/:id', cocheController.deleteCoche);

module.exports = router;
```

Aquí, `cocheController` se refiere al controlador que contendrá las funciones para manejar cada una de estas rutas. Cada ruta corresponde a una operación CRUD específica:

- `getAllCoches`: Obtiene todos los coches.
- `getCocheById`: Obtiene un coche por su ID.
- `createCoche`: Crea un nuevo coche.
- `updateCoche`: Actualiza un coche por su ID.
- `deleteCoche`: Elimina un coche por su ID.

En resumen aqui en routes se define las rutas que tu aplicación Express responderá y las vinculas a funciones específicas en un controlador.

Cada ruta se asocia a una función en el controlador que manejará la lógica de negocio específica para esa ruta. Esto ayuda a mantener el código organizado y seguir el principio de separación de preocupaciones.

El siguinete paso sera instalar el controlador

Crontrollers

En este archivo, definirás las funciones que manejarán las operaciones CRUD (Crear, Leer, Actualizar y Eliminar) para los coches. Cada función se asocia con una ruta específica en el archivo de rutas.

Puedes crear el controlador y definir las funciones básicas para comenzar. Por ejemplo:

```
// cocheController.js

const Coche = require('../models/coche');

// Obtener todos los coches
exports.getAllCoches = async (req, res) => {
  try {
    const coches = await Coche.find();
    res.status(200).json(coches);
```

```
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};

// Obtener un coche por su ID
exports.getCocheById = async (req, res) => {
  try {
    const coche = await Coche.findById(req.params.id);
    if (!coche) {
      return res.status(404).json({ error: 'Coche no encontrado' });
    }
    res.status(200).json(coche);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};

// Crear un nuevo coche
exports.createCoche = async (req, res) => {
  try {
    const nuevoCoche = await Coche.create(req.body);
    res.status(201).json(nuevoCoche);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};

// Actualizar un coche por su ID
exports.updateCoche = async (req, res) => {
  try {
    const coche = await Coche.findByIdAndUpdate(req.params.id, req.body, {
      new: true,
      runValidators: true,
    });
    if (!coche) {
      return res.status(404).json({ error: 'Coche no encontrado' });
    }
    res.status(200).json(coche);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};

// Eliminar un coche por su ID
exports.deleteCoche = async (req, res) => {
  try {
    const coche = await Coche.findByIdAndDelete(req.params.id);
    if (!coche) {
      return res.status(404).json({ error: 'Coche no encontrado' });
    }
    res.status(200).json({ mensaje: 'Coche eliminado correctamente' });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

A continuacion se explicara el primer metodo de obtener todos lo coches getAllCoches

```
// Obtener todos los coches
exports.getAllCoches = async (req, res) => {
  try {
    // Utilizamos el modelo Coche para realizar una consulta a la base de datos.
    // El método find() sin argumentos devuelve todos los documentos en la colección Coche.
    const coches = await Coche.find();
```

```
    // Respondemos con un código de estado 200 (OK) y la lista de coches en formato JSON.
    res.status(200).json(coches);
  } catch (error) {
    // Si ocurre un error durante la operación, lo capturamos aquí.
    // Imprimimos el error en la consola para propósitos de depuración.
    console.error(error);

    // Respondemos con un código de estado 500 (Internal Server Error) y un mensaje de error.
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

Desglosemos el código línea por línea:

1. `exports.getAllCoches = async (req, res) => {`: Esta línea exporta la función `getAllCoches` para que pueda ser utilizada en otros archivos. La función toma dos parámetros: `req` (solicitud) y `res` (respuesta). Además, se define como una función asíncrona usando `async`.
2. `try {`: Inicia un bloque `try`, que es utilizado para envolver el código donde se pueden producir errores.
3. `const coches = await Coche.find();`: Utiliza el modelo `Coche` (previamente importado) para realizar una consulta a la base de datos. El método `find()` sin argumentos devuelve todos los documentos en la colección `Coche`. Utilizamos `await` para esperar la respuesta de esta operación, ya que las operaciones de base de datos son asíncronas.
4. `res.status(200).json(coches);`: Si la consulta a la base de datos es exitosa, respondemos con un código de estado `200` (OK) y enviamos la lista de coches como respuesta en formato JSON.
5. `} catch (error) {`: Inicia un bloque `catch` que se ejecuta si ocurre un error dentro del bloque `try`.
6. `console.error(error);`: Imprime el error en la consola. Esto es útil para depuración y seguimiento de errores.
7. `res.status(500).json({ error: 'Error del servidor' });`: Si se produce un error, respondemos con un código de estado `500` (Internal Server Error) y un mensaje de error en formato JSON.
8. `}`: Cierra el bloque `try-catch`.

Esta función se encarga de obtener todos los coches de la base de datos y responder con la lista de coches o manejar cualquier error que pueda ocurrir durante este proceso.

A continuacion se explicara como obtenerlo por id

```
// Obtener un coche por su ID
exports.getCocheById = async (req, res) => {
  try {
    // Utilizamos el método findById proporcionado por Mongoose para buscar un coche por su ID
    const coche = await Coche.findById(req.params.id);

    // Verificamos si el coche no fue encontrado
    if (!coche) {
      // Si no se encuentra, respondemos con un código de estado 404 y un mensaje de error
      return res.status(404).json({ error: 'Coche no encontrado' });
    }

    // Si se encuentra, respondemos con un código de estado 200 y enviamos el coche en formato JSON
    res.status(200).json(coche);
  } catch (error) {
    // Si hay un error, lo manejamos y respondemos con un código de estado 500 y un mensaje de error
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

Método `getCocheById`:

- `exports.getCocheById = async (req, res) => { ... }`: Exportamos esta función como `getCocheById`. Es una función asíncrona que toma los objetos `req` (solicitud) y `res` (respuesta).
- `try { ... }`: Iniciamos un bloque `try-catch` para manejar posibles errores.
- `const coche = await Coche.findById(req.params.id);`: Utilizamos el método `findById` proporcionado por Mongoose para buscar un coche por su ID, que se obtiene de `req.params.id`.
- `if (!coche) { ... }`: Verificamos si no se encontró un coche con ese ID.
  - `return res.status(404).json({ error: 'Coche no encontrado' });`: Si no se encuentra, respondemos con un código de estado 404 y un mensaje de error.
- `res.status(200).json(coche);`: Si se encuentra, respondemos con un código de estado 200 y enviamos el coche en formato JSON.
- `} catch (error) { ... }`: En caso de error, lo capturamos y respondemos con un código de estado 500 (error interno del servidor) junto con un mensaje de error.

A continuacion se explicara como crear un coche

```
// Crear un nuevo coche
exports.createCoche = async (req, res) => {
  try {
    // Utilizamos el método create proporcionado por Mongoose
    // para crear un nuevo coche con los datos proporcionados en req.body
    const nuevoCoche = await Coche.create(req.body);

    // Respondemos con el nuevo coche creado en formato JSON y un código de estado 201 (creado)
    res.status(201).json(nuevoCoche);
  } catch (error) {
    // Si hay un error, lo manejamos y respondemos con un código de estado 500 y un mensaje de error
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

1. **Método `createCoche`:**
   - `exports.createCoche = async (req, res) => { ... }`: Exportamos esta función como `createCoche`. Es una función asíncrona que toma los objetos `req` (solicitud) y `res` (respuesta).
   - `try { ... }`: Iniciamos un bloque `try-catch` para manejar posibles errores.
   - `const nuevoCoche = await Coche.create(req.body);`: Utilizamos el método `create` proporcionado por Mongoose para crear un nuevo coche con los datos proporcionados en `req.body`.
     - `req.body`: Los datos del nuevo coche, obtenidos del cuerpo de la solicitud.
   - `res.status(201).json(nuevoCoche);`: Respondemos con el nuevo coche creado en formato JSON y un código de estado 201 (creado).
   - `} catch (error) { ... }`: En caso de error, lo capturamos y respondemos con un código de estado 500 (error interno del servidor) junto con un mensaje de error.

A continuacion se explicara como actualizarlo por id

```
// Actualizar un coche por su ID
exports.updateCoche = async (req, res) => {
  try {
    // Utilizamos el método findByIdAndUpdate proporcionado por Mongoose
    // para buscar un coche por su ID y actualizarlo con los nuevos datos proporcionados en req.body
    const coche = await Coche.findByIdAndUpdate(req.params.id, req.body, {
```

```
      new: true, // Esto asegura que el método devuelva el coche actualizado
      runValidators: true, // Esto ejecuta las validaciones definidas en el esquema del modelo
    });

    // Verificamos si el coche no fue encontrado
    if (!coche) {
      // Si no se encuentra, respondemos con un código de estado 404 y un mensaje de error
      return res.status(404).json({ error: 'Coche no encontrado' });
    }

    // Si se encuentra y se actualiza correctamente, respondemos con el coche actualizado en formato JSON
    res.status(200).json(coche);
  } catch (error) {
    // Si hay un error, lo manejamos y respondemos con un código de estado 500 y un mensaje de error
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

Método `updateCoche`:

- `exports.updateCoche = async (req, res) => { ... }`: Exportamos esta función como `updateCoche`. Es una función asíncrona que toma los objetos `req` (solicitud) y `res` (respuesta).
- `try { ... }`: Iniciamos un bloque `try-catch` para manejar posibles errores.
- `const coche = await Coche.findByIdAndUpdate(req.params.id, req.body, { ... });`: Utilizamos el método `findByIdAndUpdate` proporcionado por Mongoose para buscar un coche por su ID y actualizarlo con los nuevos datos proporcionados en `req.body`.
  - `req.params.id`: El ID del coche que queremos actualizar, obtenido de los parámetros de la solicitud.
  - `req.body`: Los nuevos datos que se utilizarán para actualizar el coche, obtenidos del cuerpo de la solicitud.
  - `{ new: true, runValidators: true }`: Opciones que indican que queremos que el método devuelva el coche actualizado (`new: true`) y que ejecute las validaciones definidas en el esquema del modelo (`runValidators: true`).
- `if (!coche) { ... }`: Verificamos si no se encontró un coche con ese ID.
  - `return res.status(404).json({ error: 'Coche no encontrado' });`: Si no se encuentra, respondemos con un código de estado 404 y un mensaje de error.
- `res.status(200).json(coche);`: Si se encuentra y se actualiza correctamente, respondemos con el coche actualizado en formato JSON.
- `} catch (error) { ... }`: En caso de error, lo capturamos y respondemos con un código de estado 500 (error interno del servidor) junto con un mensaje de error.

Por ultimo eliminar coche por su id

```
// Eliminar un coche por su ID
exports.deleteCoche = async (req, res) => {
  try {
    // Utilizamos el método findByIdAndDelete proporcionado por Mongoose
    // para encontrar y eliminar un coche por su ID
    const cocheEliminado = await Coche.findByIdAndDelete(req.params.id);

    // Verificamos si el coche se encontró y eliminó correctamente
    if (!cocheEliminado) {
      return res.status(404).json({ error: 'Coche no encontrado' });
    }

    // Respondemos con el coche eliminado en formato JSON
```

```
    res.status(200).json(cocheEliminado);
  } catch (error) {
    // Si hay un error, lo manejamos y respondemos con un código de estado 500 y un mensaje de error
    console.error(error);
    res.status(500).json({ error: 'Error del servidor' });
  }
};
```

Método `deleteCoche`:

- `exports.deleteCoche = async (req, res) => { ... }`: Exportamos esta función como `deleteCoche`. Es una función asíncrona que toma los objetos `req` (solicitud) y `res` (respuesta).
- `try { ... }`: Iniciamos un bloque `try-catch` para manejar posibles errores.
- `const cocheEliminado = await Coche.findByIdAndDelete(req.params.id);`: Utilizamos el método `findByIdAndDelete` proporcionado por Mongoose para encontrar y eliminar un coche por su ID.
  - `req.params.id`: El ID del coche que se va a eliminar, obtenido de los parámetros de la solicitud.
- `if (!cocheEliminado) { ... }`: Verificamos si el coche se encontró y eliminó correctamente. Si no se encuentra, respondemos con un código de estado 404 (no encontrado) y un mensaje de error.
- `res.status(200).json(cocheEliminado);`: Respondemos con el coche eliminado en formato JSON y un código de estado 200 (éxito).
- `} catch (error) { ... }`: En caso de error, lo capturamos y respondemos con un código de estado 500 (error interno del servidor) junto con un mensaje de error.

El siguinete paso es conectar los metodos creeados en el archivo perteneciente a la carpeta de controllers

Conectar las rutas con la aplicación principal (`app.js`):

En el archivo principal (app.js), necesitas importar las rutas de coches y usarlas como middleware. Puedes hacerlo de la siguiente manera:

```
const cocheRoutes = require('./routes/cocheRoutes');

// ...

// Usar las rutas de coches
app.use('/coches', cocheRoutes);
```

1. Esto significa que todas las rutas definidas en `cocheRoutes` estarán precedidas por '/coches'. Por ejemplo, para obtener todos los coches, la ruta completa sería '/coches/getAll'.
2. **Probar las rutas en Thunder Client o Postman:**
   Ahora que tus rutas y controladores están configurados, puedes probarlas utilizando Thunder Client o Postman. Aquí hay algunos ejemplos de las rutas que has creado:
   - `GET /coches/getAll`: Obtener todos los coches.
   - `GET /coches/getById/:id`: Obtener un coche por su ID.
   - `POST /coches/create`: Crear un nuevo coche.
   - `PUT /coches/update/:id`: Actualizar un coche por su ID.
   - `DELETE /coches/delete/:id`: Eliminar un coche por su ID.
3. **Añadir más funcionalidades si es necesario:**
   Según los requisitos de tu aplicación, puedes expandir y mejorar las funcionalidades. Por ejemplo, podrías

añadir validaciones adicionales, paginación, filtrado, etc.

4. **Manejar la conexión a la base de datos y configuraciones:**
   Asegúrate de que tu conexión a la base de datos esté configurada correctamente en el archivo `app.js`.
   Puedes ajustar configuraciones como el puerto, la URL de conexión a la base de datos, etc.

Por ultimo tambien debemos de conectar estas funciones de controllers con las rutas (cocheRoutes.js)

```
const cocheController = require('../controllers/cocheController');
```

Siguiente paso realizar Middleware

Middleware

Crea un nuevo archivo llamado `cocheMiddleware.js` en tu carpeta de middleware.

Un middleware en Express es una función que tiene acceso a los objetos de solicitud (`req`), respuesta (`res`), y a la siguiente función de middleware en el ciclo de solicitud-respuesta de la aplicación, comúnmente denotada por `next()`. Los middlewares pueden realizar tareas como modificar la solicitud o respuesta, finalizar el ciclo de solicitud-respuesta, llamar al siguiente middleware en la pila o finalizar la pila de middlewares.

En el contexto de una API, los middlewares son especialmente útiles para realizar tareas de validación, autenticación, autorización, manipulación de datos y otras funciones antes de llegar a las funciones del controlador que manejan las rutas.

En tu caso, el middleware `validateCocheData` está diseñado para validar los datos de un coche antes de que se llegue al controlador que maneja la creación o actualización de un coche. Aquí hay una explicación detallada:

```
// cocheMiddleware.js

// Middleware para validar datos específicos de coches antes de llegar al controlador
exports.validateCocheData = (req, res, next) => {
    // Ejemplo de validación: asegurémonos de que el cuerpo de la solicitud contenga un campo 'modelo'
    if (!req.body.modelo) {
        return res.status(400).json({ error: 'El campo "modelo" es obligatorio para un coche' });
    }

    // Puedes agregar más validaciones según tus necesidades

    // Si pasa la validación, llamamos a la siguiente función de middleware o al controlador
    next();
};
```

- El middleware `validateCocheData` recibe la solicitud (`req`), la respuesta (`res`), y la función `next`.
- En este ejemplo, se realiza una validación simple para asegurarse de que el cuerpo de la solicitud (`req.body`) contenga un campo llamado 'modelo'. Si no está presente, responde con un código de estado 400 y un mensaje de error.
- Puedes agregar más validaciones según tus necesidades específicas.
- Si pasa la validación, llamamos a `next()` para pasar al siguiente middleware o al controlador

Podemos poner mas validaciones por ejemplo

```
// Verificar que el campo "anio" sea un número positivo
if (anio < 0)) {
  return res.status(400).json({ error: 'El campo "anio" debe ser un número positivo' });
}

//Verificar la longitud del campo "color"
if (color.length > 20)) {
   return res.status(400).json({ error: 'El campo "color" debe tener menos de 20 caracteres' });
}
```

Es importante que debemos de importar el middlwares en routes

```
const { validateCocheData } = require('../middleware/cocheMiddleware');
```

```
// Crear un nuevo coche (aplicamos el middleware de validación antes de llamar a la función del controlador)
router.post('/', validateCocheData, (req, res) => {
    res.send('Crear un nuevo coche');
  });

// Actualizar un coche por ID (aplicamos el middleware de validación antes de llamar a la función del controlador)
router.put('/:id', validateCocheData, (req, res) => {
  res.send('Actualizar un coche por ID');
});
```

La inclusión del middleware en las rutas `POST` y `PUT` tiene como objetivo validar los datos antes de llegar a la función del controlador correspondiente. Esta práctica es común en aplicaciones web para asegurarse de que los datos proporcionados en las solicitudes cumplan con ciertos criterios antes de procesarlos.

En tu caso, estás utilizando el middleware `validateCocheData` para realizar la validación específica de los datos de coches. El middleware verifica si el cuerpo de la solicitud (`req.body`) contiene el campo requerido, que en este caso es el campo "modelo". Si el campo "modelo" no está presente en la solicitud, se responde con un error indicando que este campo es obligatorio. Si la validación pasa, el control se pasa a la siguiente función, que es la función del controlador correspondiente (`(req, res) => {...}`).

En resumen, el flujo es el siguiente:

1. Cuando se recibe una solicitud `POST` en la ruta `/coches`, primero pasa a través del middleware `validateCocheData`.
2. El middleware verifica que el campo "modelo" esté presente en el cuerpo de la solicitud.
3. Si la validación es exitosa, la solicitud se pasa a la función del controlador, que responde con el mensaje "Crear un nuevo coche".
4. Si la validación falla, se responde con un mensaje de error antes de llegar a la función del controlador.

Este enfoque ayuda a mantener la consistencia y la integridad de los datos al garantizar que solo se procesen datos válidos.

Por ultimo debemos de crear una carpeta denomina Helpers donde almacenaremos un archivo denominado **db-validators.js**

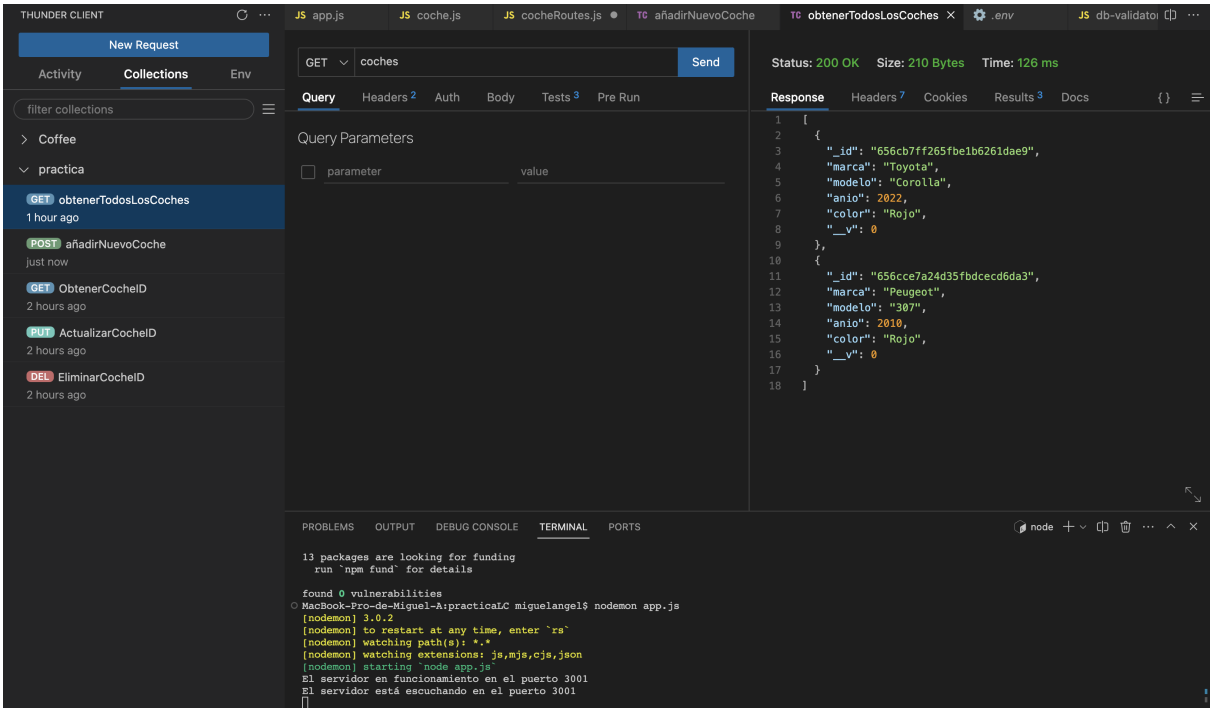
```
const validatePositiveYear = (year) => {
    if (year < 0) {
```

```
        throw new Error('El año del coche debe ser un número positivo');
    }
};

module.exports = { validatePositiveYear };
```

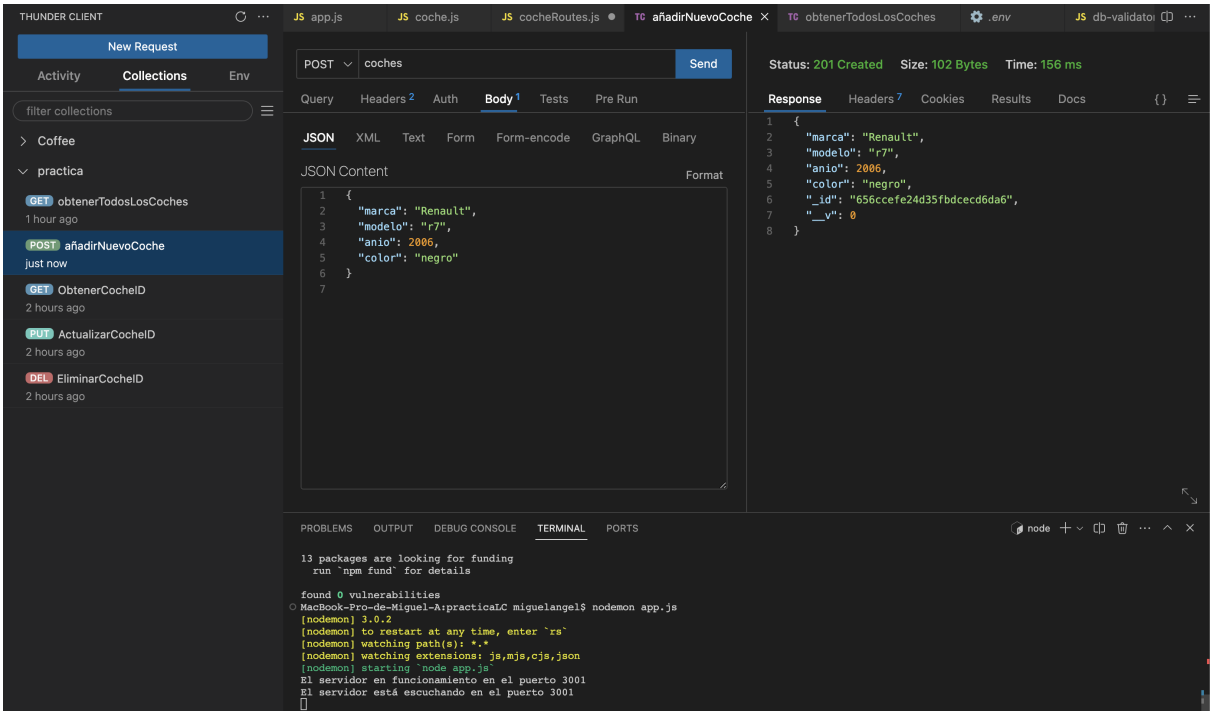
Ahora puedes usar esta función de validación en tu middleware o en cualquier parte de tu código donde necesites asegurarte de que el año del coche sea un número positivo. Por ejemplo, podrías usarlo en tu middleware de validación `validateCocheData` en `cocheMiddleware.js`. Puedes agregar esta validación después de verificar la existencia del campo `anio`. Si la validación falla, puedes enviar una respuesta de error al cliente.

```
const { validatePositiveYear } = require('../helpers/db-validators');

exports.validateCocheData = (req, res, next) => {
    // Ejemplo de validación: asegurémonos de que el cuerpo de la solicitud contenga un campo 'modelo'
    if (!req.body.modelo) {
        return res.status(400).json({ error: 'El campo "modelo" es obligatorio para un coche' });
    }

    // Validación del año
    if (req.body.anio) {
        try {
            validatePositiveYear(req.body.anio);
        } catch (error) {
            return res.status(400).json({ error: error.message });
        }
    }

    // Puedes agregar más validaciones según tus necesidades

    // Si pasa la validación, llamamos a la siguiente función de middleware o al controlador
    next();
};
```

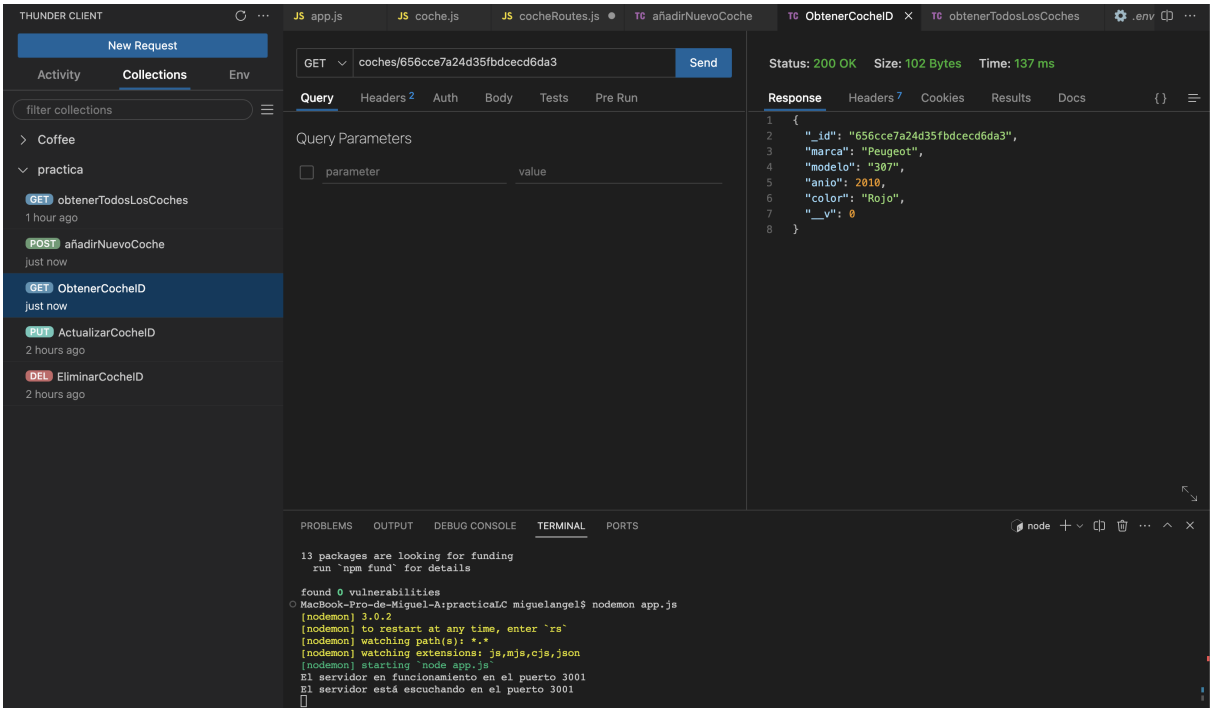
Obtener todos los coches



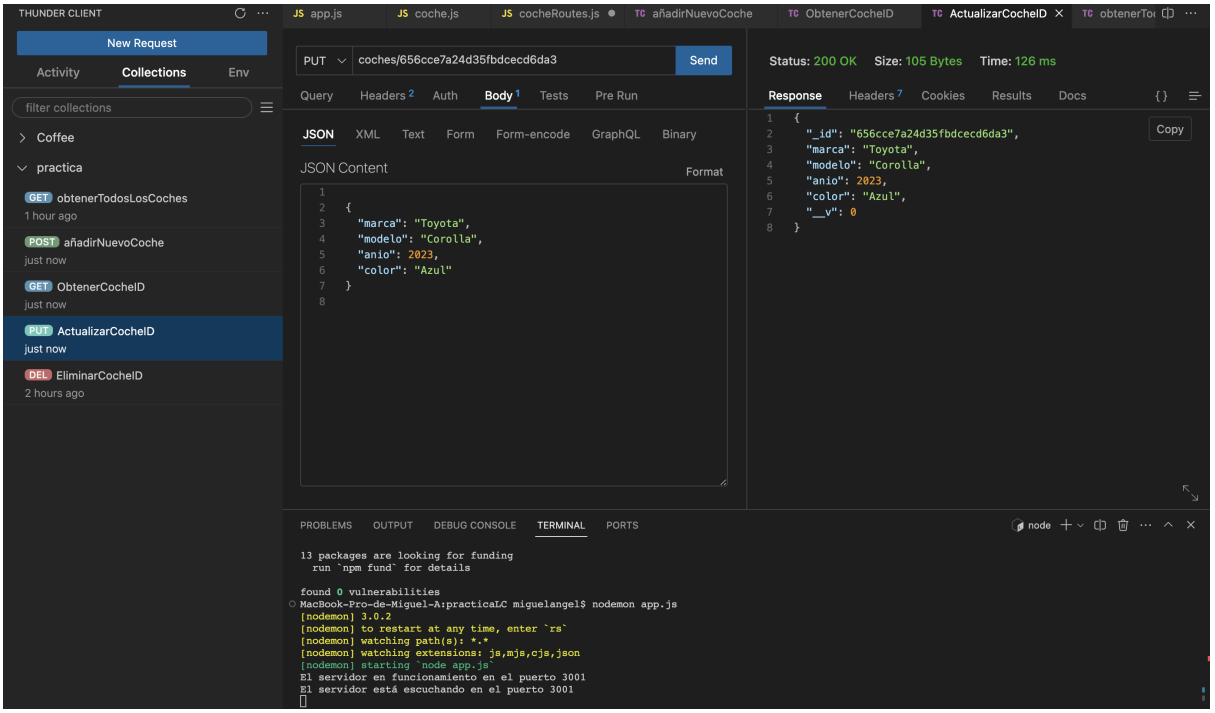
Añadir un nuevo coche



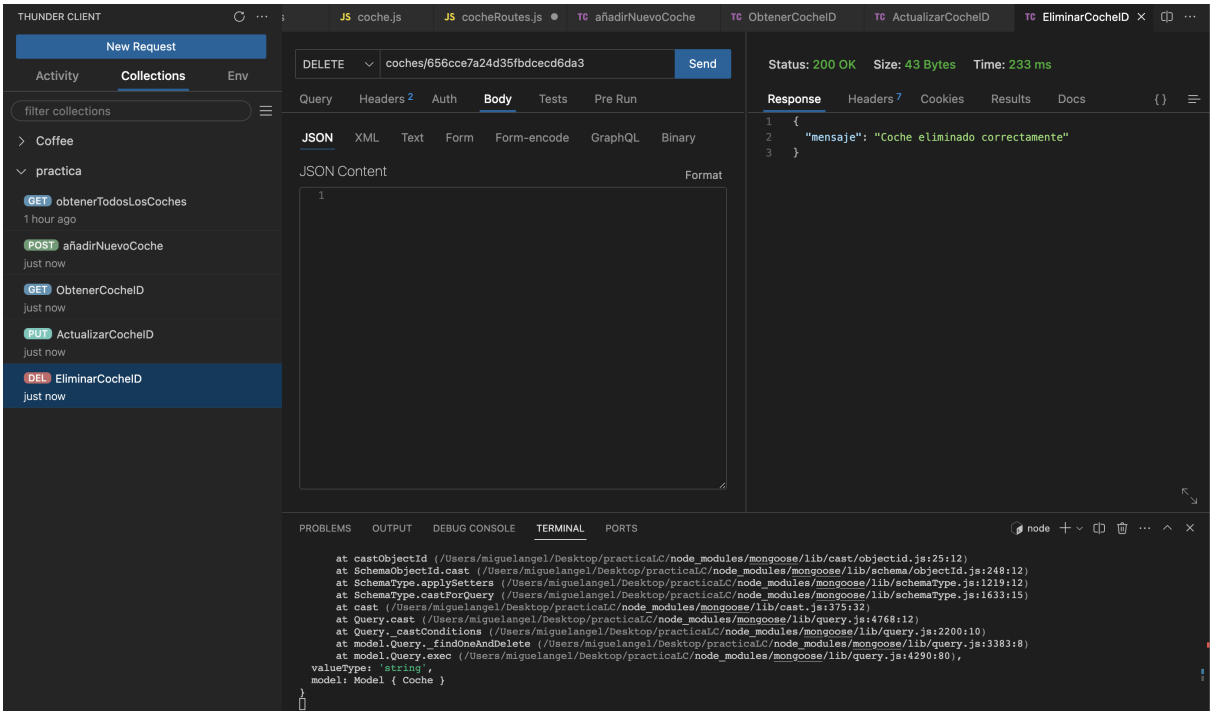
Obtener coche id



Actualizar coche por id



Eliminar coche por id


THUNDER CLIENT
New Request
Activity
Collections
Env
filter collections
Coffee
practica
GET obtenerTodosLosCoches
1 hour ago
POST añadirNuevoCoche
just now
GET ObtenerCocheID
just now
PUT ActualizarCocheID
just now
DEL EliminarCocheID
just now
coche.js
cocheRoutes.js
añadirNuevoCoche
ObtenerCocheID
ActualizarCocheID
EliminarCocheID
DELETE
coches/656cce7a24d35fbdcecd6da3
Send
Query
Headers 2
Auth
Body
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
JSON Content
Format
Status: 200 OK
Size: 43 Bytes
Time: 233 ms
Response
Headers 7
Cookies
Results
Docs
{
"mensaje": "Coche eliminado correctamente"
}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
node
at castObjectId (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/cast/objectid.js:25:12)
at SchemaObjectId.cast (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/schema/objectId.js:248:12)
at SchemaType.applySetters (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/schemaType.js:1219:12)
at SchemaType.castForQuery (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/schemaType.js:1633:15)
at cast (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/cast.js:375:32)
at Query.cast (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/query.js:4768:12)
at Query._castConditions (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/query.js:2200:10)
at model.Query._findOneAndDelete (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/query.js:3383:8)
at model.Query.exec (/Users/miguelangel/Desktop/practicaLC/node_modules/mongoose/lib/query.js:4290:80),
valueType: 'string',
model: Model { Coche }
}