Laboratorio 07: API REST

Introducción

Las **APIs (Interfaces de Programación de Aplicaciones)** son esenciales para la comunicación entre aplicaciones y servicios en el mundo digital. Una de las arquitecturas más populares para construir APIs es **REST (Representational State Transfer)**, que utiliza el protocolo HTTP y sigue principios claros para acceder y gestionar recursos de manera eficiente.

Esta guía ofrece una visión general de las APIs RESTful, incluyendo su estructura, los métodos HTTP, la gestión de errores y las mejores prácticas para su implementación. Tanto si eres un desarrollador experimentado como si estás comenzando, encontrarás en esta guía las herramientas necesarias para crear APIs RESTful efectivas que mejoren tus aplicaciones.

Que es mongoose

Mongoose es una biblioteca de modelado de objetos para **MongoDB** y **Node.js**. Proporciona una solución elegante para interactuar con bases de datos MongoDB al permitir a los desarrolladores definir modelos y esquemas que reflejan la estructura de los datos. Mongoose facilita la validación, el manejo de relaciones y la creación de consultas complejas, lo que simplifica el proceso de desarrollo de aplicaciones que requieren una base de datos NoSQL.

Que es dotenv

dotenv es una biblioteca que permite cargar variables de entorno desde un archivo `.env` en el entorno de ejecución de una aplicación Node.js. Esto es útil para mantener la configuración de la aplicación, como las credenciales de la base de datos, las claves API y otras configuraciones sensibles, fuera del código fuente. Al utilizar dotenv, los desarrolladores pueden gestionar fácilmente diferentes configuraciones para entornos de desarrollo y producción sin comprometer la seguridad.

Configurar el Proyecto

Para iniciar, utilizaremos el comando `npm init -y`, que nos permite inicializar rápidamente un proyecto en Node.js. **npm** (Node Package Manager) es el gestor de paquetes de Node.js, y el flag `-y` acepta automáticamente los valores predeterminados para la configuración del proyecto, generando el archivo `package.json` sin necesidad de responder las preguntas habituales, lo que agiliza el proceso inicial.

Además, instalaremos nuestro framework **Express**, junto con **Mongoose** para manejar la base de datos MongoDB, y **dotenv** para gestionar las variables de entorno. Estos paquetes son fundamentales para estructurar la aplicación y conectarla de manera segura a una base de datos.

```
npm init -y
npm install express mongoose dotenv
```

Estructura del proyecto

```
LABORATORIO-07/
├── node_modules/
├── src/
│   ├── config/
│   │   └── dbConnection.config.js
│   ├── controllers/
│   │   ├── author.controller.js
│   │   └── book.controller.js
│   ├── middleware/
│   │   └── error.middleware.js
│   ├── models/
│   │   ├── author.model.js
│   │   └── book.model.js
│   ├── routes/
│   │   ├── author.route.js
│   │   ├── book.route.js
│   │   └── main.route.js
│   ├── services/
│   │   ├── author.service.js
```

```
│   │   └── book.service.js
│   └── utils/errors/
│       ├── author.errorCodes.js
│       ├── book.errorCodes.js
│       └── serviceError.js
├── .env
├── .gitignore
├── app.js
├── package-lock.json
└── package.json
```

Archivo gitignore

El archivo `.gitignore` permite indicarle a Git qué archivos no deben ser incluidos en el repositorio, ya sea porque no son necesarios para la ejecución del código o porque no deben compartirse. Para ello, se crea un archivo llamado `.gitignore`, en el cual se especifican los nombres de los archivos o directorios que queremos que Git ignore.

Por ejemplo, uno de los casos más comunes es ignorar los módulos o dependencias instaladas, como los que se encuentran en la carpeta `node_modules` en proyectos de JavaScript.

También es importante ignorar el archivo `.env`, que contiene las **variables de entorno** sensibles, como las claves de acceso a la base de datos o las configuraciones secretas del servidor. Este archivo nunca debe subirse a un repositorio público para evitar que información sensible sea expuesta.

```
node_modules/
.env
```

Variables de entorno

El archivo `.env` es utilizado para almacenar variables de entorno que configuran el comportamiento de una aplicación. Este enfoque permite mantener la configuración separada del código fuente, lo que mejora la seguridad y la flexibilidad.

En el ejemplo, `MONGODB_URI` define la conexión a una base de datos MongoDB, mientras que `PORT` establece el puerto en el que la aplicación escuchará las solicitudes. Almacenar estas configuraciones en el archivo `.env` facilita el cambio de parámetros sin necesidad de modificar el código, lo que es especialmente útil cuando se trabaja en diferentes entornos, como desarrollo y producción.

El user y la pass deben ser sustituidos por los datos elegidos durante la creación de la base de datos

```
// example
MONGODB_URI=mongodb://user:pass@localhost:27017/nameDB?authSo
PORT=44000
```

Conexión a la MongoDB

`dbConnection.config.js`

Este código se encarga de establecer una conexión a una base de datos MongoDB utilizando Mongoose, una biblioteca que facilita la interacción con bases de datos MongoDB desde aplicaciones Node.js.

La función `conecctiondb` verifica si la URI de la base de datos está correctamente definida y, de ser así, intenta conectarse a MongoDB. Si la conexión es exitosa, se notifica en la consola. En caso de fallos durante la conexión, se maneja el error de manera que se informa al desarrollador sobre el problema.

```
import mongoose from 'mongoose';
import 'dotenv/config';

export const connectiondb = async()=>{
    const URI = process.env.MONGODB_URI;

    if(!URI){
        throw new Error('No se ha definido la URI de la base
    }

    try{
        await mongoose.connect(URI);
        console.log('Base de datos conectada');
```

```
    }catch(e){
        console.error(e);
        throw new Error('Error a la hora de iniciar la base d
    }
}
```

Creación de modelo

`author.model.js`

Cuando usamos **Mongoose**, una popular librería para integrar `Node.js` con **MongoDB**, el primer paso clave es definir un **modelo**. En `Mongoose`, un modelo actúa como la representación de la estructura de los datos que almacenaremos en la base de datos.

Los modelos se basan en **esquemas**, que establecen qué campos contendrá cada documento dentro de una **colección** de `MongoDB`, junto con las reglas que deben seguir esos campos (tipos de datos, valores por defecto, si son obligatorios, etc.). Además, los modelos manejan las **relaciones** entre documentos mediante el uso de `ObjectId`, que referencia documentos en otras colecciones, permitiendo modularidad y consultas eficientes.

```
import { Schema, model } from 'mongoose';


const authorSchema = new Schema({
    name:{
        type: String,
        required: true
    },
    lastName:{
        type: String,
        required: true
    },
    books:[
        {
            type: Schema.Types.ObjectId,
            ref: 'Book'
        }
```

```
    ],
    creationDate:{
        type: Date,
        default: Date.now()
    }
})

const Author = model('Author', authorSchema);

export { Author };
```

Creación de servicios

`book.service.js`

Los **servicios** en una aplicación tienen como objetivo centralizar y gestionar la **lógica de negocio**. Esto significa que se encargan de realizar las operaciones clave que involucran nuestros datos, como la creación, actualización, eliminación y consulta, utilizando los modelos que hemos definido previamente.

En lugar de tener esta lógica dispersa por todo el código, los servicios agrupan esas funcionalidades, haciendo el código más organizado y fácil de mantener. Además, se encargan de manejar validaciones, errores, y procesos más complejos, como relacionar diferentes entidades (por ejemplo, asignar autores a libros).

```
export const addAuthorToBook = async (book, authorId) => {
    try {
        const existAuthor = book.authors.find(author => aut
hor.toString() === authorId);
        if (existAuthor) throw new ServiceError('El autor y
a fue asignado al libro', BookErrorCodes.AUTHOR_ALREADY_ASS
IGNED);

        book.authors.push(authorId);
        const bookUpdated = await book.save();
        return bookUpdated;
    } catch (error) {
        throw new ServiceError('Error al asignar el autor a
```

```
l libro', BookErrorCodes.AUTHOR_ASSIGN_FAILED);
    }
}
```

👀 Este servicio, `addAuthorToBook`, asigna un autor a un libro verificando primero si el autor ya está en la lista de autores del libro. Si el autor ya está asignado, lanza un error. Si no, agrega el `authorId` al array `authors` del libro y guarda los cambios en la base de datos. Si ocurre algún error durante el proceso, lanza un error personalizado con un mensaje adecuado.

Códigos de error

`author.errorCodes.js`

En nuestros servicios, implementamos un mecanismo estructurado para manejar errores mediante códigos de error específicos. Esto nos permite identificar y tratar diferentes situaciones de manera clara y consistente, lo cual es fundamental para garantizar que los usuarios y desarrolladores puedan comprender y solucionar fácilmente cualquier problema que surja.

En este archivo, hemos definido un objeto `ErrorCodes` que contiene códigos numéricos asociados a diferentes errores relacionados con la creación, búsqueda y eliminación de autores y libros. Estos códigos facilitan una identificación precisa de los errores en nuestra aplicación.

```
export const AuthorErrorCodes = {
    AUTHOR_CREATION_FAILED: 1001,
    AUTHOR_FETCH_FAILED: 1002,
    AUTHOR_NOT_FOUND: 1003,
    AUTHOR_SEARCH_FAILED: 1004,
    BOOK_ALREADY_ASSIGNED: 1005,
    BOOK_ASSIGN_FAILED: 1006,
    AUTHOR_FETCH_BY_ID_FAILED: 1008,
    NO_AUTHORS_FOUND: 1009,
    BOOK_DELETE_FAILED: 1010,
```

```
    AUTHOR_DELETE_FAILED: 1012,
};
```

👀 Este objeto, `AuthorErrorCodes`, define un conjunto de códigos de error numéricos que están asociados con diferentes tipos de fallos o errores. Cada código de error es único y puede ser utilizado para identificar y manejar errores de manera más precisa en la aplicación.

Manejo personalizado de errores

`ServiceError.js`

Para gestionar los errores de manera más clara y estructurada dentro de nuestra aplicación, hemos creado una clase personalizada llamada `ServiceError`. Esta clase extiende la clase base `Error` de JavaScript, permitiéndonos no solo manejar mensajes de error, sino también asociar un **código numérico** a cada error.

```
export class ServiceError extends Error {
    constructor(message, code) {
        super(message);
        this.code = code;
    }
}
```

👀 La clase `ServiceError` extiende la clase estándar `Error` en JavaScript para permitir el manejo de errores personalizados con un **mensaje** y un **código**. Esto es útil para lanzar y capturar errores más específicos en la aplicación, proporcionando tanto una descripción del problema como un código de error que ayuda a identificar la causa exacta.

Creación de controladores

`author.controller.js`

En esta sección, definimos los controladores de nuestra aplicación, que actúan como intermediarios entre las solicitudes del cliente y los servicios. Su principal función es gestionar la lógica de negocio, interactuar con los servicios para realizar operaciones y enviar respuestas al cliente. Un aspecto fundamental es el manejo de errores, donde se capturan excepciones en el bloque `catch` y se envían mensajes claros basados en los códigos de error del servicio. Estos códigos permiten identificar la naturaleza del problema,

```
export const createAuthorController = async (req, res, nex
t) => {
    try {
        const author = req.body;
        const existAuthor= await findAuthorByNameAndLastNam
e(author.name, author.lastName);
        if(existAuthor) throw createError(400, 'El autor ya
existe');

        const authorCreated = await saveAuthor(author);
        res.status(201).json({ message: 'author created', d
ata: authorCreated });
    } catch (e) {
        switch(e.code)
        {
            case AuthorErrorCodes.AUTHOR_NOT_FOUND:
                next(createError(404, 'El autor no exist
e'));
                break;
            case AuthorErrorCodes.AUTHOR_SEARCH_FAILED:
                next(createError(500, 'Error al buscar el a
utor'));
                break;
            default:
                next(e);
        }
    }
}
```

👀 Este controlador, `createAuthorController`, maneja la creación de un autor, y utiliza un formato de manejo de excepciones que captura errores lanzados desde los servicios. Los códigos de error definidos previamente en `AuthorErrorCodes` se utilizan para responder con mensajes y códigos de estado HTTP apropiados.

Creación de rutas

`authore.route.js`

En esta sección, nos enfocaremos en la creación de **rutas** dentro de nuestra aplicación Express. Las rutas son esenciales para gestionar las solicitudes HTTP y definir cómo la aplicación responde a diferentes endpoints. Utilizaremos controladores para manejar la lógica de negocio asociada con cada ruta.

En esta implementación, definiremos la estructura de las rutas que permitirá acceder a nuestras funciones, especificando la extensión que debe tener cada ruta y el método HTTP correspondiente. También se pueden añadir middlewares para realizar validaciones y otras funcionalidades necesarias, en este caso solo aremos uso de nuestro controlador.

```
authorRouter.put('/:authorId/:bookId', assingBookToAuthorCont
```

👀 La ruta permite realizar una operación de actualización para asignar un libro a un autor específico. En esta ruta, `authorId` y `bookId` son **parámetros dinámicos** que representan los identificadores únicos de un autor y un libro en la base de datos, respectivamente. Cuando se recibe una solicitud a esta URL, el servidor extrae automáticamente estos **ID** y los pasa al controlador.

Creación de Rutas en el Archivo Principal

`main.route.js`

El `main route` en una aplicación Express es el archivo central donde se organizan y gestionan todas las rutas de la aplicación. Actúa como un punto de redirección, dirigiendo las solicitudes a los enrutadores específicos de cada recurso. Esto permite que cada enrutador maneje la lógica correspondiente y las respuestas adecuadas, manteniendo el código estructurado y fácil de mantener. Al centralizar la gestión de rutas, el `main route` facilita la expansión de la aplicación al permitir la incorporación de nuevas rutas de manera eficiente.

```
import {bookRouter} from './book.route.js';
import {authorRouter}  from './author.route.js';
import { Router } from 'express';

const mainRouter = Router();

mainRouter.use('/book', bookRouter);
mainRouter.use('/author', authorRouter);

export { mainRouter };
```

👀 En este fragmento de código, se importan los routers `bookRouter` y `authorRouter` desde sus respectivos archivos de ruta. Luego, se crea un **router principal** utilizando `Router` de **Express**. A continuación, se configuran las rutas base para los libros y los autores mediante `mainRouter.use('/book', bookRouter)` y `mainRouter.use('/author', authorRouter)`. Esto significa que cualquier solicitud a la ruta `/book` será manejada por `bookRouter` y las solicitudes a `/author` serán gestionadas por `authorRouter`.

Middleware para manejo de errores

`error.middleware.js`

El middleware de manejo de errores es una parte fundamental en las aplicaciones Express, diseñado para gestionar y responder a los errores que puedan surgir durante el procesamiento de solicitudes. En este código, la función `errorHandler` recibe cuatro parámetros: `err`, `req`, `res` y `next`. Cuando se produce un error en la aplicación, este middleware se activa y responde con un

código de estado HTTP adecuado (o 500 si no se especifica) y un mensaje de error en formato JSON.

Este enfoque permite centralizar el manejo de errores, proporcionando respuestas claras y consistentes ante fallos, lo que facilita la depuración y mejora la experiencia del usuario.

```
export const errorHandler = (err, req, res, next) => {

    return res.status(err.status || 500).json({ message: er
r.message || 'Something went wrong' });
}
```

Configuración de la aplicacion

`app.js`

Este código configura y ejecuta una aplicación web con Express, un framework para Node.js. Comienza importando las bibliotecas necesarias y configurando la conexión a la base de datos. Luego, inicializa la aplicación y establece middleware para procesar solicitudes en formato JSON, gestionar rutas y manejar errores de manera centralizada. Finalmente, define el puerto en el que la aplicación escuchará las solicitudes y arranca el servidor, permitiendo que la aplicación esté disponible para recibir y responder a las solicitudes de los usuarios.

```
import express from 'express';
import 'dotenv/config';
import {errorHandler} from './src/middleware/error.middleware
import {mainRouter} from './src/routes/main.route.js';
import {connectiondb} from './src/config/dbConnection.config.


const app = express();
connectiondb();

app.use(express.json());
app.use(mainRouter);
```

```
app.use(errorHandler);


const PORT = process.env.PORT || 3001;
app.listen(PORT, ()=>{
    console.log(`Server is running on port ${PORT}`);
})
```

Tareas de laboratorio

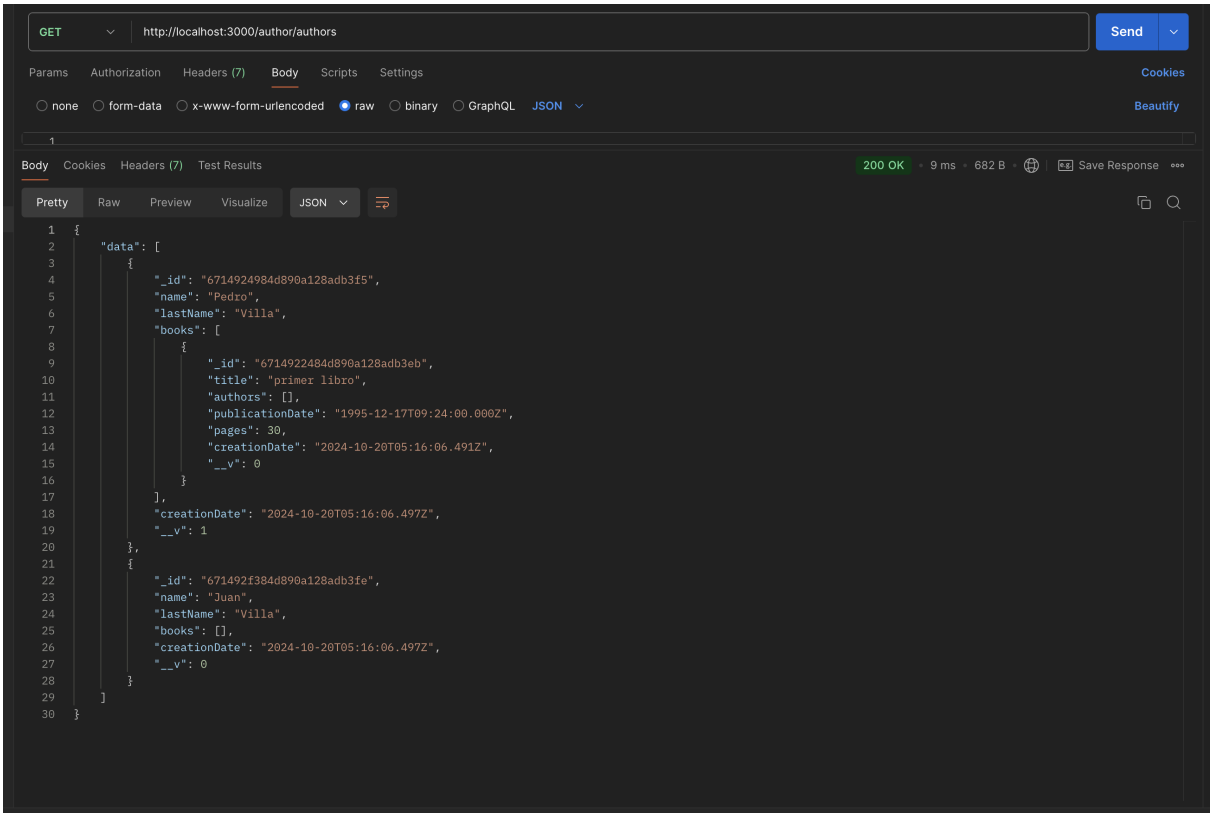
1. Obtener todos los autores

En esta tarea, el objetivo será implementar la lógica necesaria para permitir la recuperación de todos los autores desde la base de datos.

1. **Crear o Modificar el Servicio**:
    - Desarrollar un servicio que contenga una función para obtener todos los autores. Esta función debe:
        - Consultar la base de datos utilizando Mongoose para recuperar todos los documentos de la colección de autores.
        - Manejar cualquier error que pueda ocurrir durante la consulta.
2. **Generar Códigos de Errores**:
    - Establecer códigos de error específicos para la función de obtención de autores. Por ejemplo:
        - Error al obtener los autores.
3. **Desarrollar el Controlador**:
    - Implementar un controlador que se encargue de gestionar la solicitud HTTP para obtener todos los autores. Este controlador debe:
        - Llamar a la función del servicio que se creó.
        - Procesar la respuesta y enviar los autores como respuesta JSON.
        - Manejar errores llamando a un middleware de manejo de errores en caso de que se produzca una excepción.

4. **Configurar la Ruta**:
    - Definir una ruta que permita realizar la solicitud para obtener todos los autores. Esta ruta debe estar vinculada al controlador que se implementó.
    - Asegurarse de que la ruta siga el formato adecuado, por ejemplo, `GET /authors`.

> Respuesta esperada para la solución de dicha tarea



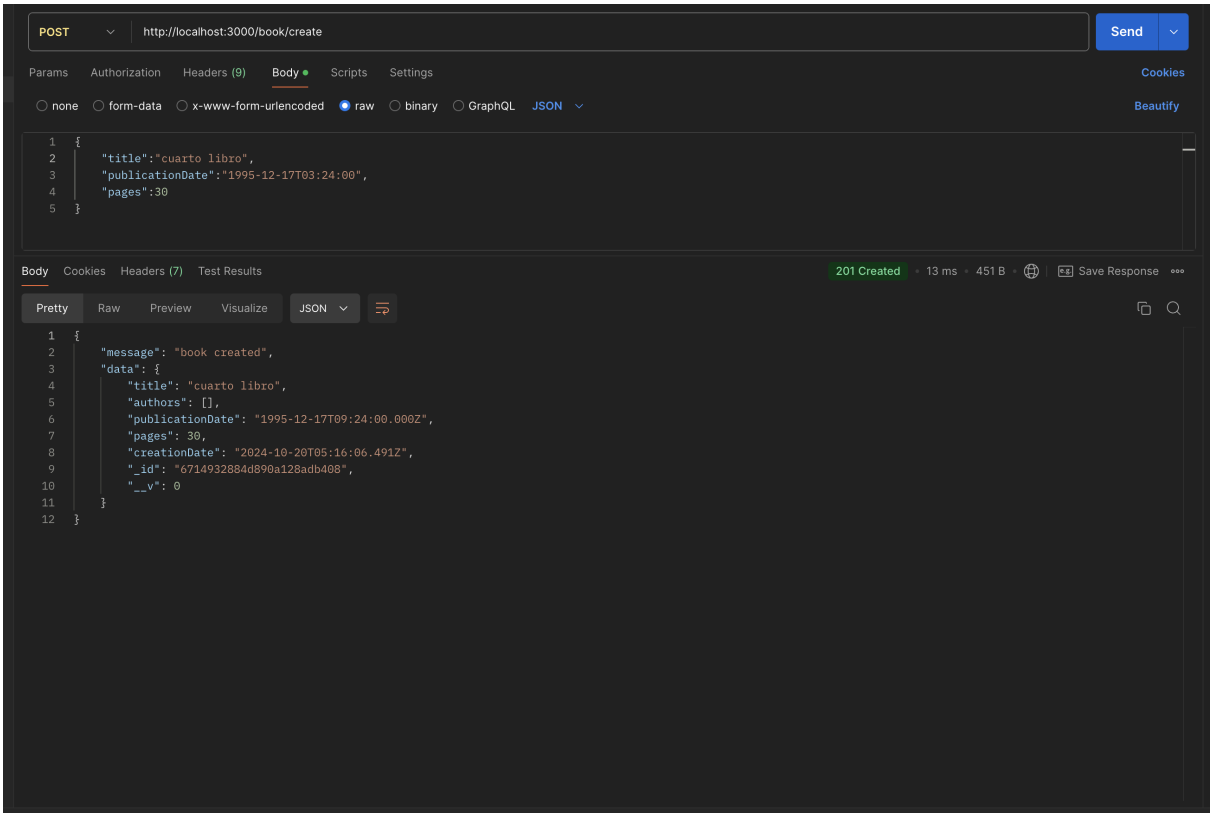
2. Creación de libros

En esta tarea, el objetivo será implementar la lógica necesaria para permitir la creación de nuevos libros en la base de datos.

Instrucciones:

1. **Crear o Modificar el Servicio**:
   - Desarrollar un servicio que contenga una función para crear un libro. Esta función debe:
     - Recibir los datos del libro como parámetro y utilizar Mongoose para guardarlo en la colección de libros.
     - Manejar cualquier error que pueda ocurrir durante la creación.
2. **Generar Códigos de Errores**:
   - Establecer códigos de error específicos para la función de creación de libros. Por ejemplo:
     - Error al crear el libro.
     - Libro ya existe (si aplica).
3. **Desarrollar el Controlador**:
   - Implementar un controlador que se encargue de gestionar la solicitud HTTP para crear un libro. Este controlador debe:
     - Llamar a la función del servicio que se creó.
     - Procesar la respuesta y enviar un mensaje de éxito o el libro creado como respuesta JSON.
     - Manejar errores llamando a un middleware de manejo de errores en caso de que se produzca una excepción.
4. **Configurar la Ruta**:
   - Definir una ruta que permita realizar la solicitud para crear un libro. Esta ruta debe estar vinculada al controlador que se implementó.
   - Asegurarse de que la ruta siga el formato adecuado, por ejemplo, `POST /create`.

Respuesta esperada para la solución de dicha tarea



Tarea para casa.

Implementación de la Entidad Editorial

Como parte de la aplicación de gestión de libros y autores, la tarea consiste en agregar una nueva entidad llamada **Editorial**. Esta entidad estará relacionada con los libros en una relación de muchos a uno (n:1), lo que significa que un libro puede pertenecer a una única editorial, mientras que una editorial puede tener múltiples libros asociados.

Instrucciones:

1. **Crear el Modelo Editorial**:
    - Definir un esquema en Mongoose para la entidad **Editorial**. Asegurarse de incluir campos como `name`, `address`, y cualquier otro dato relevante.
    - El modelo de **Libro** debe tener una referencia a la **Editorial** correspondiente. Esto se puede hacer agregando un campo de referencia en el esquema de libros.
2. **Implementar Servicios**:

- Crear funciones de servicio para manejar la lógica relacionada con las editoriales. Esto debe incluir:
    - **Creación**: Permitir la creación de una nueva editorial.
    - **Eliminación**: Permitir la eliminación de una editorial.
    - **Incluir libros a una editorial**: Funcionalidad para agregar libros existentes a una editorial.
    - **Obtener todas las editoriales**: Recuperar todas las editoriales almacenadas en la base de datos.

3. **Generar Códigos de Errores**:
    - Establecer códigos de error para manejar excepciones que puedan ocurrir en los servicios. Por ejemplo:
        - Error al crear una editorial.
        - Error al eliminar una editorial.
        - Error al incluir libros en una editorial.
        - Error al obtener editoriales.
4. **Desarrollar Controladores**:
    - Implementar controladores que se encargarán de gestionar las solicitudes HTTP y de interactuar con los servicios creados. Los controladores deben incluir:
        - Métodos para manejar las operaciones de creación, eliminación, inclusión y obtención de editoriales.
5. **Configurar Rutas**:
    - Definir las rutas necesarias para interactuar con las editoriales. Esto incluirá las rutas para crear, eliminar y obtener las editoriales, así como para incluir libros en ellas.

Objetivo:

Al finalizar esta tarea, se habrá ampliado la funcionalidad de la aplicación para incluir la gestión de editoriales, permitiendo a los usuarios crear, eliminar y consultar editoriales, así como asociar libros a ellas. Además, la implementación de códigos de error permitirá un manejo más robusto de las

excepciones y mejorará la experiencia del usuario al interactuar con la aplicación.