Universidad Tecnológica Nacional

Facultad Regional Rosario

Tecnicatura Universitaria en Programación

Programación III

Unidad 3.3

Introducción a Node

Introducción

En estas unidades nos enfocaremos en el lado servidor o *backend* del desarrollo web. Hasta este momento nuestro código solo se alojaba en el lado cliente o *frontend*, que nosotros manejamos mediante la librería de React y el lenguaje Javascript.

Veremos entonces:

- A qué nos referimos con código “de lado servidor”
- Conceptos generales de *backend*.
- Introducción a la librería node JS (que ya venimos utilizando para correr vite, pero ahora usaremos específicamente para escribir nuestro código servidor)

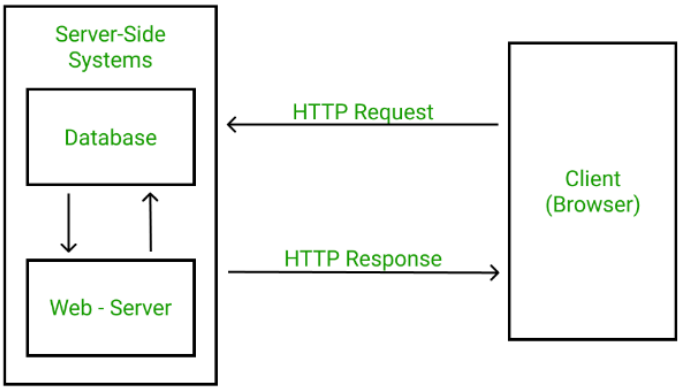
Código de lado servidor

Hasta este momento, nuestras soluciones web han sido solo pará código que se encuentra alojado en el navegador. Esto, para una gran cantidad de páginas es suficiente y necesario (pensemos en páginas web que se comportan similar a un perfil de negocios de Instagram o Facebook). Pero, actualmente, la mayoría de los sistemas web deben poseer algún tipo de lógica de servidor o *backend* funcionando, debido a diversas razones:

- Guardar y persistir *data* en una base de datos.
- Cargar esa *data* en nuestra web.
- Carga y guardado de archivos (documentos, imágenes, videos, etc)

El camino en los sistemas web

Imaginemos un caso de uso donde yo entro a Netflix y me carga las primeras 100 últimas películas disponibles de la plataforma, ¿Cómo sería el camino que realiza el sistema web?



1. En el lado cliente, en el navegador nosotros cargamos la página */home* de Netflix y eso carga el componente MoviesList.
2. El componente, a su vez, hace una llamada (denominada *request*) al servidor, pidiéndole las últimas 100 películas disponibles en la plataforma.
3. El servidor recibe el pedido y se encarga de elaborar una consulta a la base de datos para traer esa información.
4. El servidor recibe los datos, los emprolija / modifica para que los reciba el cliente (armando los conocidos como DTOs (*Data Transfer Object*)) y elaborando de esa forma una *response* (o respuesta).
5. El componente MoviesList recibe la respuesta (muy probablemente en formato JSON) y se encarga de listar todas las películas para visualización de las mismas en el navegador.

ORM

Si observamos el gráfico anterior, vemos que el web server debe comunicarse con la base de datos. Esto se puede hacer de manera directa mediante queries de SQL del tipo:

*SELECT * FROM BOOKS;*

O se puede utilizar lo que se conoce como un **ORM (Object relational mapping)**. ORM es una técnica de programación en donde abstraemos o creamos un puente entre la capa de la base de datos y la capa servidor, de manera que relacionemos el paradigma orientado a objetos del servidor con el modelo relacional de tablas de SQL. La anterior query en el ORM que vamos a utilizar durante el cursado (llamado *sequelize*) se vería así:

const books = await Book.findAll();

Donde Book es la entidad que trabajamos en React, ahora con propiedades que se correspondan con las columnas de la tabla Book en SQL.

Node.js

Para poder montar nuestro servidor, utilizaremos node.js (que ya lo habíamos instalado previamente para poder correr vite en el entorno de React) junto con express como framework de *backend*. A su vez, utilizaremos SQLite como motor de base de datos junto a *sequelize* como ORM conector.

Para comenzar, debemos iniciar un nuevo proyecto de Node y agregarle todos los *packages* necesarios. Abrimos una nueva carpeta llamada book-champions-api y allí dentro, en una terminal, correremos la siguientes línea de código:

Primero:
npm init -y

(-y es para evitar que nos haga preguntas de setup)

Luego:
npm i express morgan sequelize sqlite3

Morgan es una librería que actua de middleware en node, para registrar en consola los pedidos que se realicen a nuestro servidor.

Es importante recordar la diferencia entre Node y express. Node es una **herramienta** que nos permite **correr código javascript fuera del navegador**, mientras que express es un ***framework* ligero que nos ayuda a escribir código servidor**, abstrayendo y simplificando muchas de las funcionalidades que habría que armar si no lo usamos.

Dentro del *package.json*,vamos a agregar dos scripts:

Los dos scripts agregados son:

start: permite correr el proyecto sin necesidad de escribir el comando de node. Debemos correrlo con *npm start*.
dev: permite correr el proyecto y estar pendiente de los cambios (mediante el método conocido como *hot reload*). Esto hará que si hacemos algún cambio en el código servidor, el mismo se reinicie. Lo corremos con *npm run dev*.

Debemos también agregar el atributo *"type": "module"* de manera que podamos utilizar los *import* / *export* de JS moderno.

Creamos a su vez la carpeta llamada *src* y dentro, agregamos un archivo *index.js,* que se encargará de manejar el arranque de la app.

Ahora si, en index.js haremos que el servidor escuche en el puerto 3000 de nuestra computadora.

```
1   import express from 'express';
2
3   const app = express();
4
5   const port = 3000;
6
7   app.listen(port);
8   console.log(`Server listening on port ${port}`)
9
```

Luego, en una terminal corremos:

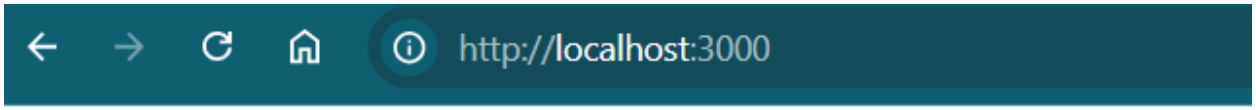
npm run dev

Deberíamos ver algo así:

```
> book-champions-api@1.0.0 dev
> node --watch src/index.js

(node:24444) ExperimentalWarning: Watch mode is an experimental feature and might change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
Server listening on port 3000
```

Si nos dirigimos a http://localhost:3000 (http, no https) veremos el siguiente mensaje:




```
Cannot GET /
```

Eso si bien es un error, es positivo debido a que nos indica que el servidor esta corriendo.

Mejoras

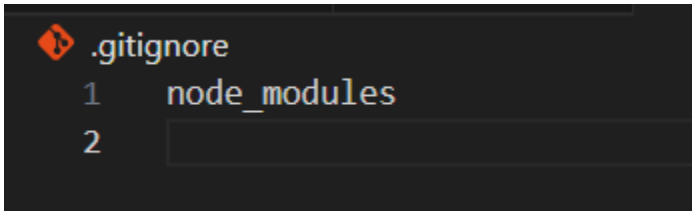
El valor del puerto (en nuestro caso, 3000) podría estar dentro de un archivo *config.js*, al mismo nivel que *index*

```
src > JS config.js > ...
1   export const PORT = 3000;
```

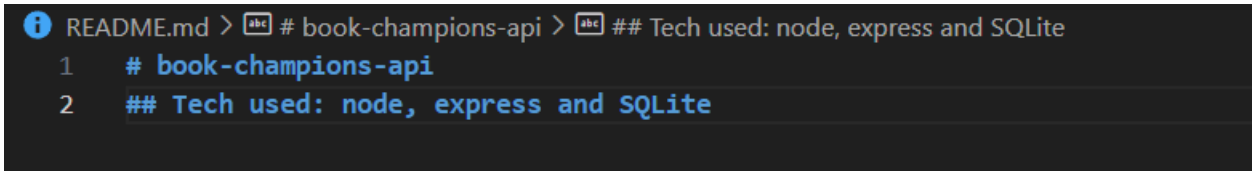
Y luego en index modificamos:

```
src > JS index.js > ...
1   import express from 'express';
2
3   import { PORT } from './config.js';
4
5   const app = express();
6
7   app.listen(PORT);
8   console.log(`Server listening on port ${PORT}`)
```

Por otro lado, para evitar subir los node_modules a git (y otros archivos innecesarios), vamos a crear un .gitignore:



y un README.md con el nombre del proyecto:



Rutas y controladores

La comunicación entre el lado cliente y el lado servidor comienza en las **rutas** (también conocidos como controladores). Estas funciones se encargarán de recibir los pedidos al servidor (*request*) y elaborar una respuesta al cliente (*response*).

Dentro del protocolo REST, nos vamos a concentrar en 4 verbos específicos

- **GET**: se utiliza para obtener datos del servidor (ejemplo, traerme todos los libros disponibles de la app)
- **POST**: se usa para generar nuevas entidades en la base de datos (por ejemplo, agregar un libro)
- **PUT**: nos sirve para actualizar datos de una entidad (ejemplo, actualizar el título de un libro)
- **DELETE**: se utiliza para eliminar una entidad (por ejemplo, eliminar un libro)

Estos verbos se relacionan con el concepto de CRUD en software (Create, Read, Update, Delete), donde el camino básico para cualquier entidad en la base de datos es

- Creación
- Lectura
- Actualización
- Eliminación

Creando las rutas

Vamos entonces a armar nuestras rutas básicas para interactuar con la entidad Libro. En una nueva carpeta llamada routes, agregamos un archivo llamado *books.routes.js* con el siguiente código:

```
1   import { Router } from "express";
2
3   const router = Router();
4
5   router.get("/books", (req, res) => {
6       res.send("Obteniendo libros")
7   });
8
9   export default router;
```

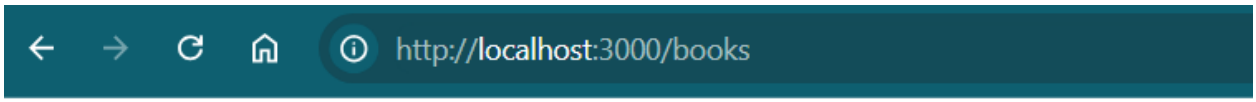
Allí inicializamos el Router y le agregamos la primera ruta (un get en la url http://localhost:3000/books). El objeto router tiene acceso a los distintos verbos que vimos más arriba y acepta dos parámetros:

1. Un *string* con el path donde se va a alojar esa ruta.
2. Una callback que recibe como parámetros la *request* y la *response*, en donde nosotros invocando la response, enviamos un mensaje al usuario.

Luego, tendremos que hacer que *express* reconozca ese router.

```
1   import express from 'express';
2
3   import { PORT } from './config.js';
4   import bookRoutes from "./routes/books.routes.js"
5
6   const app = express();
7
8   app.listen(PORT);
9   app.use(bookRoutes);
10  console.log(`Server listening on port ${PORT}`)
11
```

Si ahora nos dirigimo al navegador, a la url http://localhost:3000/books, deberíamos ver el siguiente mensaje:



Obteniendo libros

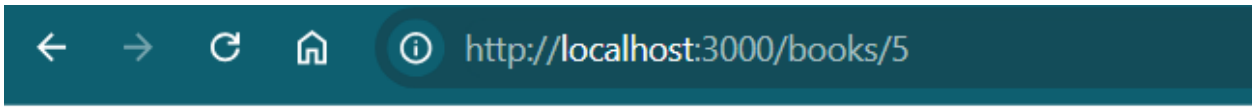
Parámetros dinámicos

Si bien esa ruta nos sirve para acceder a todos los libros de nuestra app ¿Qué hacemos cuando necesitamos la información de uno solo? Todos los libros en nuestra futura base de datos relacional van a tener un id autoincremental que los identifica. Eso se lo podríamos pasar en el pedido al servidor de la siguiente manera:

```
9   router.get("/books/:id", (req, res) => {
10      const { id } = req.params;
11      res.send(`Obteniendo libro con id: ${id}`);
12  });
```

Ahí el :id, indica que debemos pasarle una id (por ejemplo, 5) para identificar el libro correspondiente. Luego, destructuramos esa información del parámetro de la *request*.

Si nos dirigimos a http://localhost:3000/books/5



Obteniendo libro con id: 5

O 1213

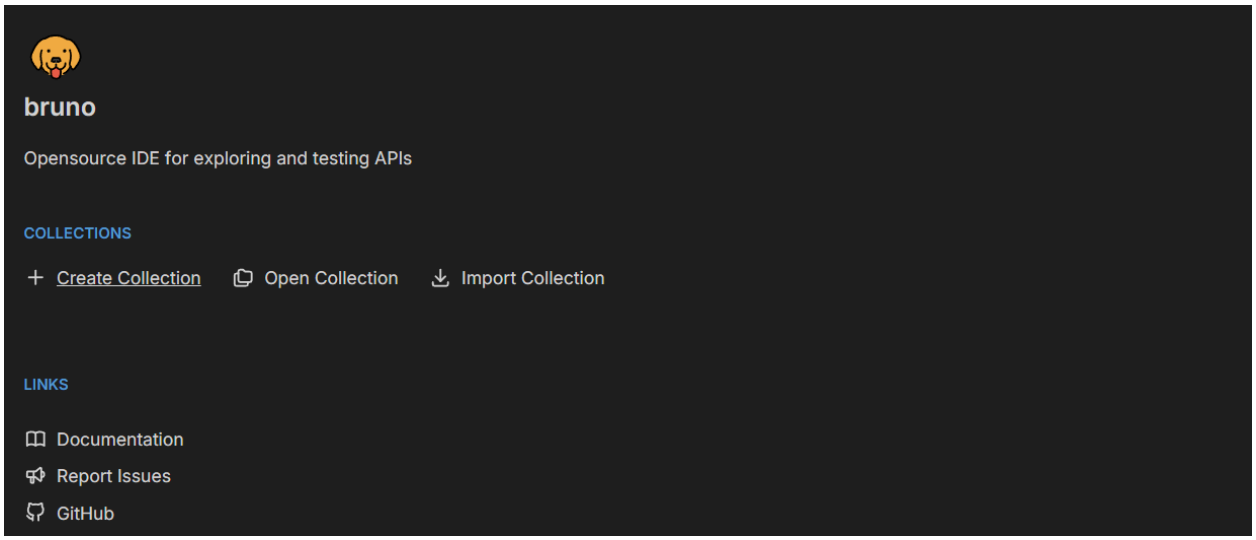
Obteniendo libro con id: 1213

Agregando las rutas faltantes

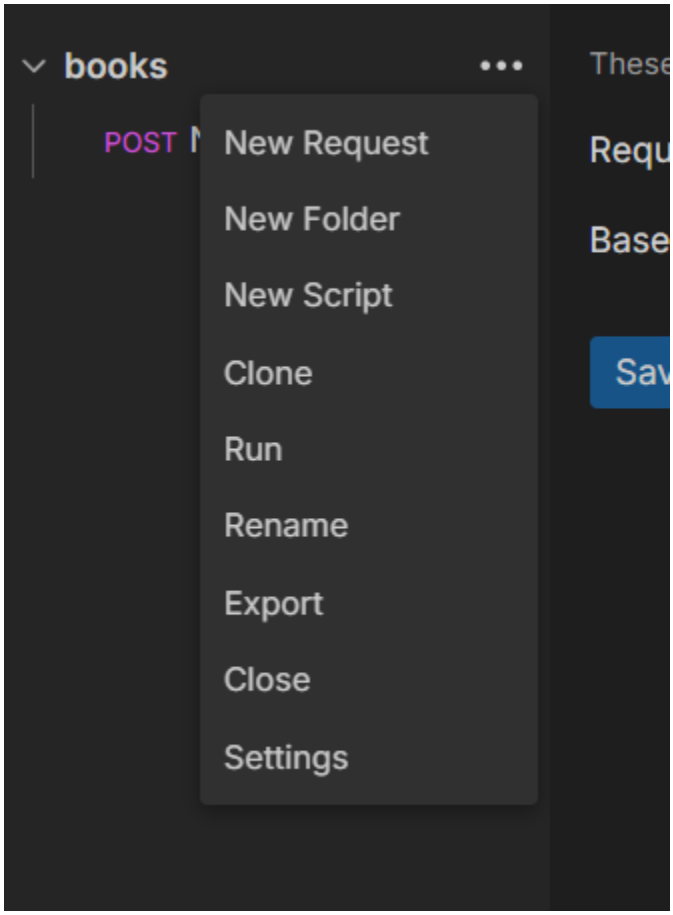
```
14  router.post("/books", (req, res) => {
15      res.send("Creando libro")
16  });
17
18  router.put("/books/:id", (req, res) => {
19      const { id } = req.params;
20      res.send(`Actualizando libro con id: ${id}`);
21  });
22
23  router.delete("/books/:id", (req, res) => {
24      const { id } = req.params;
25      res.send(`Borrando libro con id: ${id}`);
26  });
```

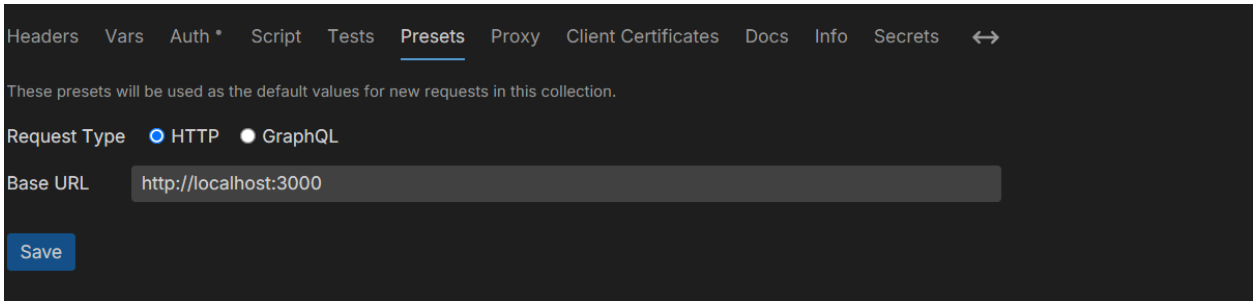
Agregamos entonces rutas para *post*, *put* y *delete*. Pero al ser verbos distintos del *get*, va a ser necesario especificar el verbo mediante un cliente. Postman es uno de los más utilizados, pero nosotros optamos por una variante *open source* llamada Bruno.

Para poder utilizarlo, procedemos a descargarlo e instalarlo. Al abrirnos, se nos presentará un botón llamado “Create Collection”y le pondremos el nombre de Books-Dev.



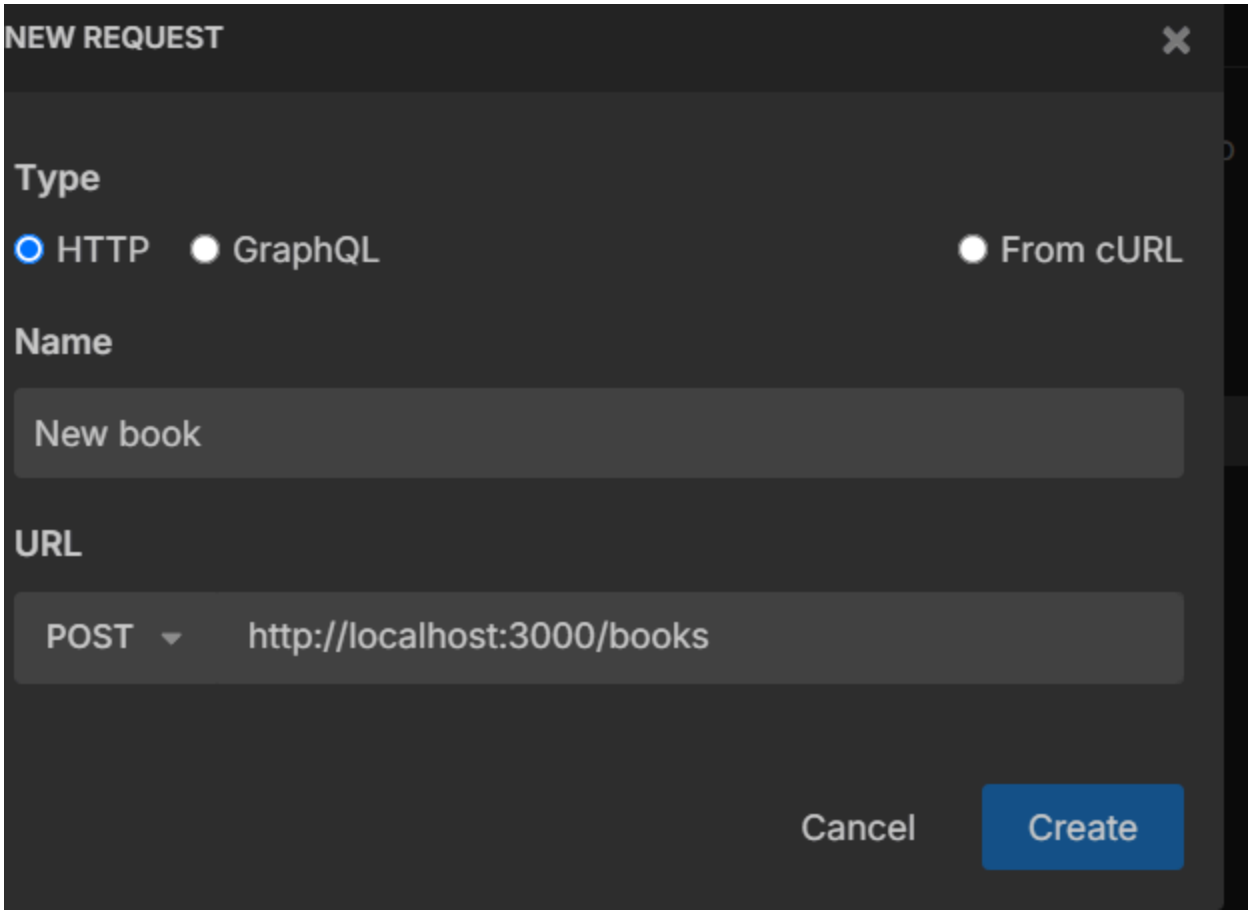
Primero, en la opción de *settings* que se abre cuando seleccionamos los puntos suspensivos en la colección, nos dirigiremos a la sección de Presets



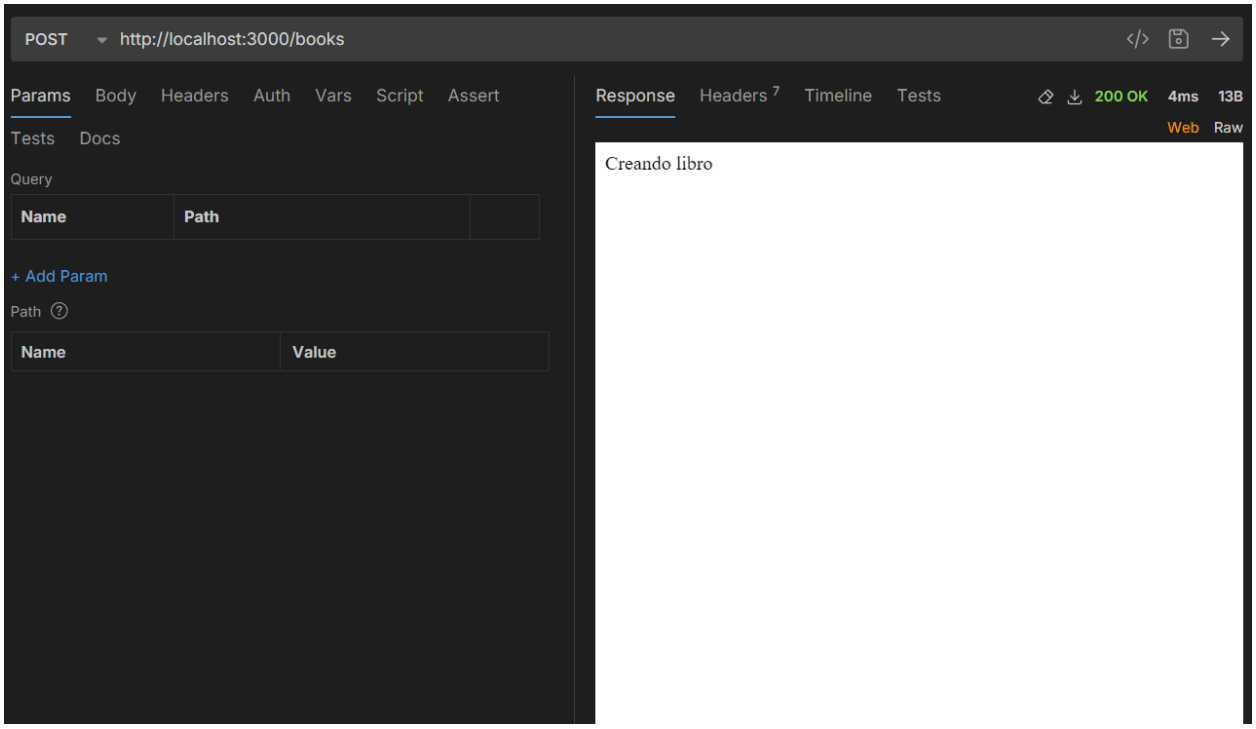


Allí agregaremos la URL base para todas nuestras peticiones.

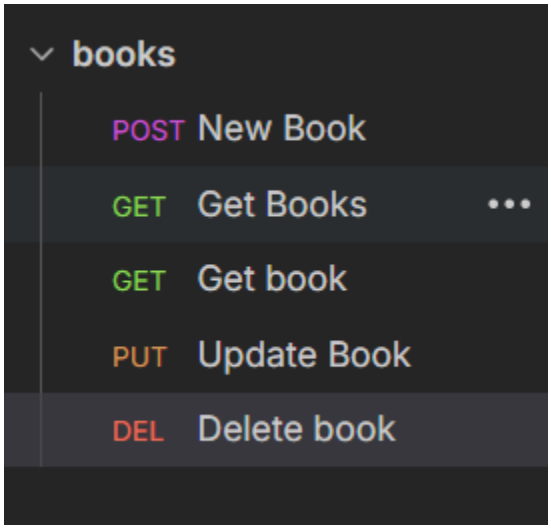
A continuación, vamos a crear nuestra primera *request* en la opción *new request*.



Luego de apretar en Create, podremos testear de la siguiente forma:



Y allí observamos que nos devolvió la respuesta correcta. Realizar lo mismo con las 5 rutas que creamos.



Base de datos

Una **base de datos** es un sistema organizado para recopilar, almacenar y gestionar información de manera estructurada.

Utiliza **tablas** para organizar datos relacionados con filas que representan registros individuales y columnas que definen atributos específicos. Se emplean sistemas de gestión de base de datos (SGBD) para administrar estas estructuras, permitiendo la entrada, búsqueda y manipulación eficiente de datos.

En los apuntes de la cátedra utilizaremos SQLite3 como motor de base de datos y sequelize como el ORM asociado.

Instanciando nuestra base de datos

Dentro del archivo *db.js*, instanciamos nuestra base de datos indicando el “dialecto” (el motor de base de datos a utilizar) y donde guardaremos el archivo de base de datos en el proyecto:

```
1  import { Sequelize } from "sequelize";
2
3  export const sequelize = new Sequelize({
4      dialect: 'sqlite',
5      storage: './books.db'
6  });
```

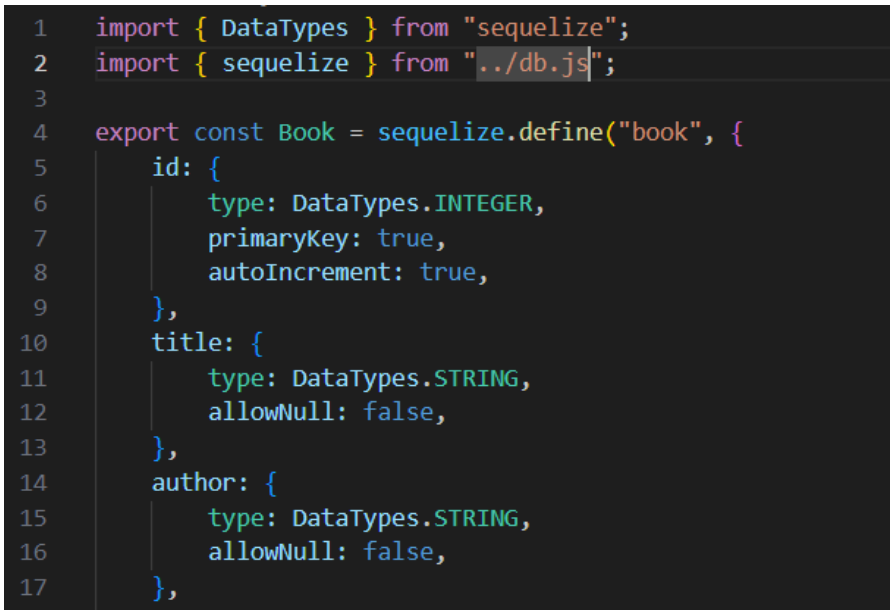
Luego, en index.js probaremos que todo funcione utilizando un *try-catch*:

```
7   const app = express();
8
9   try {
10      app.listen(PORT);
11      app.use(bookRoutes);
12
13      await sequelize.authenticate();
14
15      console.log(`Server listening on port ${PORT}`);
16
17  } catch (error) {
18      console.log(`There was an error on initialization`);
19  }
20
```

Modelos

Los modelos van a ser la representación de las tablas de SQL a crear. Estas representaciones van a estar escritas en Node y serán traducidas gracias a Sequelize para que el motor de base de datos la pueda interpretar correctamente.

Por ahora, creamos a nivel de *routes* una carpeta llamada *models* con el archivo Book.js:



```
import { DataTypes } from "sequelize";
import { sequelize } from "../db.js";

export const Book = sequelize.define("book", {
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true,
    },
    title: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    author: {
        type: DataTypes.STRING,
        allowNull: false,
    },
```

Vamos a definir las siguientes propiedades para nuestra entidad Book:

- id
  - Tipo: INTEGER
  - Es *primary key*
  - Es autoincremental
- title
  - Tipo: STRING
  - No permite nulo
- author
  - Tipo: STRING
  - No permite nulo
- rating
  - Tipo: INTEGER
- pageCount:
  - Tipo: INTEGER
- summary
  - Tipo: TEXT
- imageUrl
  - Tipo: STRING
- available
  - Tipo: BOOLEAN
  - Valor default: false

Como tercer parámetro, pondremos *timestamps: false*, para que no nos agregue los *createdAt* y los *updatedAt.*

Entonces, vamos a cambiar el *authenticate* de la base de datos por un *sync* (que se encargará de, como dice su nombre, sincronizar la base de datos con lo especificado en el código) e importamos el modelo en index.

```
1   import express from 'express';
2
3   import { PORT } from './config.js';
4   import { sequelize } from './db.js';
5
6   import "./models/Book.js"
7
8   import bookRoutes from "./routes/books.routes.js"
9
10  const app = express();
11
12  try {
13      app.listen(PORT);
14      app.use(bookRoutes);
15
16      await sequelize.sync();
17
18      console.log(`Server listening on port ${PORT}`);
19
20  } catch (error) {
21      console.log(`There was an error on initialization`);
22  }
23
```

Una vez que levantemos el servidor, deberíamos ver el siguiente mensaje:

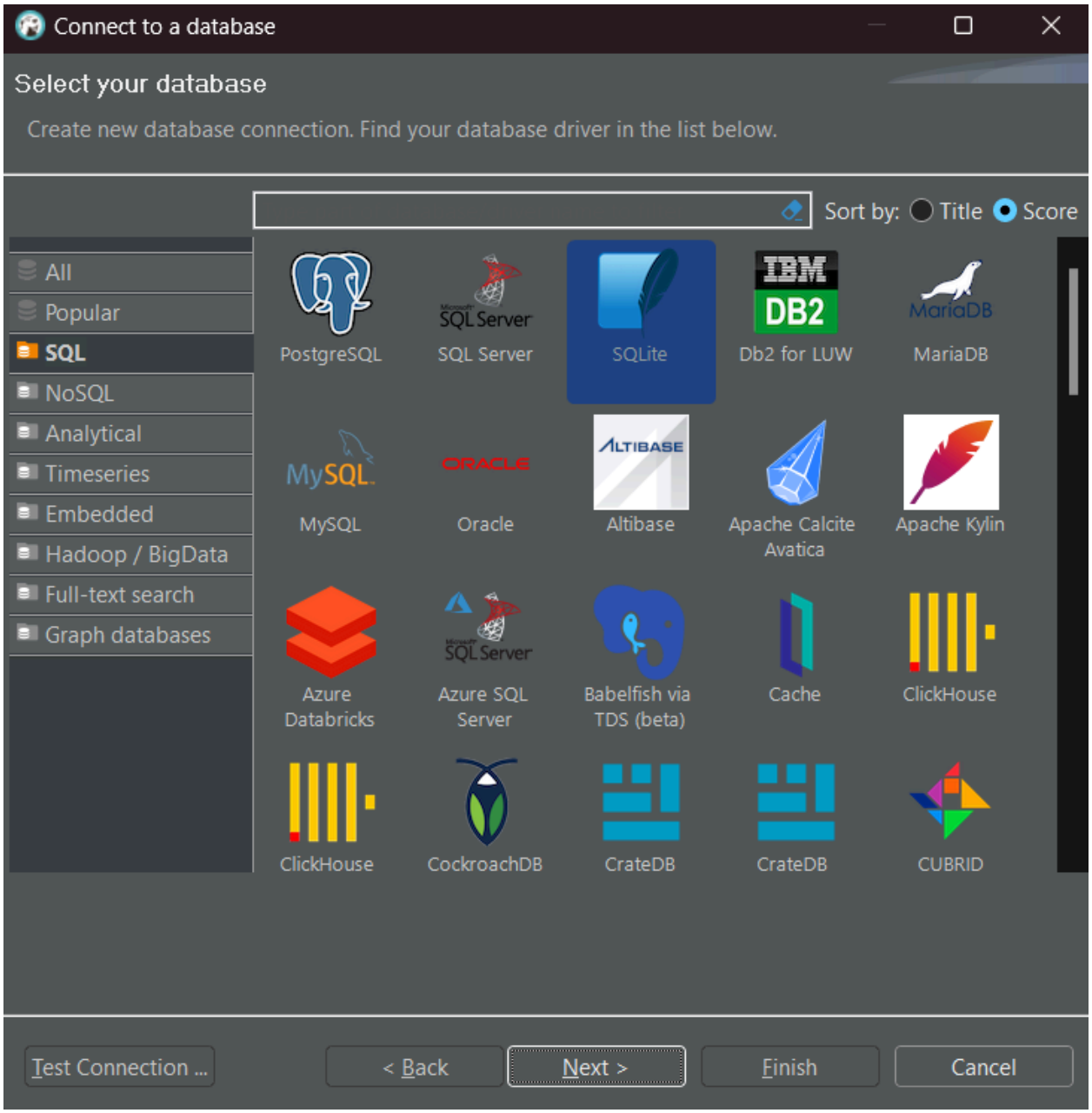
```
Executing (default): SELECT name FROM sqlite_master WHERE type='table' AND name='books';
Executing (default): CREATE TABLE IF NOT EXISTS `books` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `title` VARCHAR(255) NOT NULL, `author` VARCHAR(255) NOT NULL,
 `rating` INTEGER, `pageCount` INTEGER, `summary` TEXT, `imageUrl` VARCHAR(255), `available` TINYINT(1) DEFAULT 1, `createdAt` DATETIME NOT NULL, `updatedAt` D
ATETIME NOT NULL);
Executing (default): PRAGMA INDEX_LIST(`books`)
Server listening on port 3000
```

Que nos indica que se ha creado la tabla.

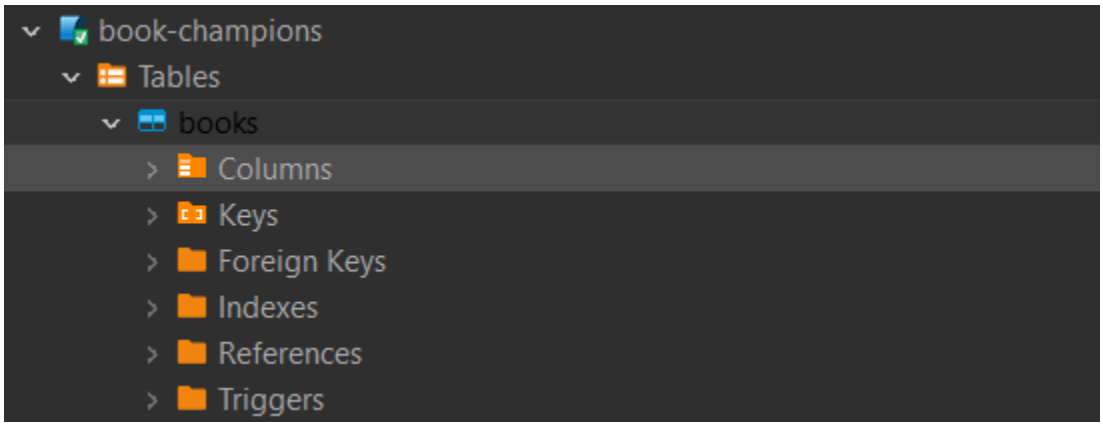
Visualización de base de datos

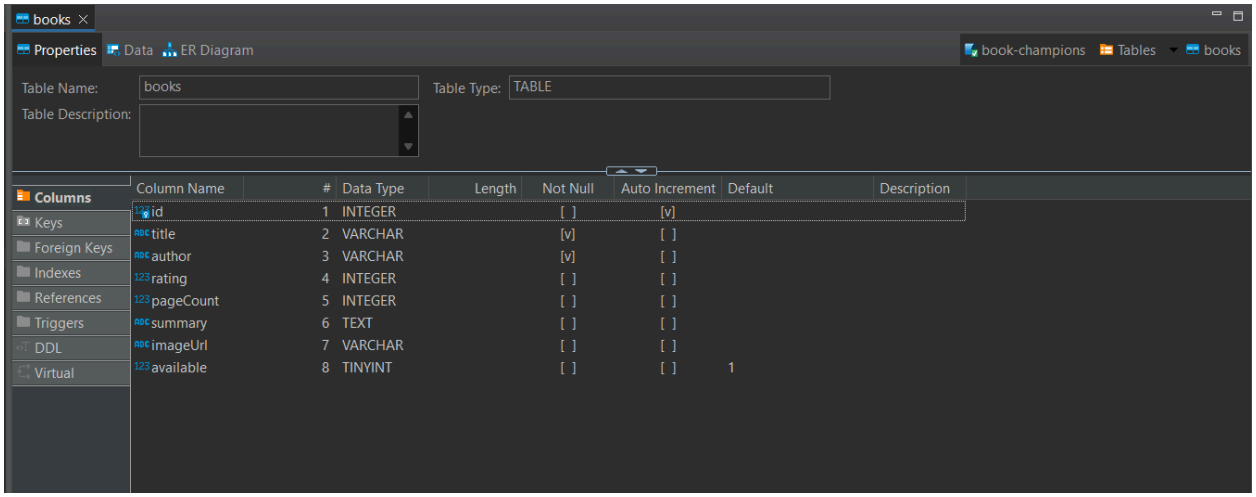
Para poder no solo gestionar las bases de datos que creamos, sino también poder visualizarlas, vamos a utilizar el programa DBeaver en su versión Community (gratuita).

Luego de la instalación, agregaremos una conexión del tipo SQLite:



Luego, con el explorador de archivos que nos provee buscamos el archivo books.db de nuestro proyecto. Una vez que se encuentra cargado, verificaremos que la tabla *book* este presente (sin embargo, aún no tendrá datos)



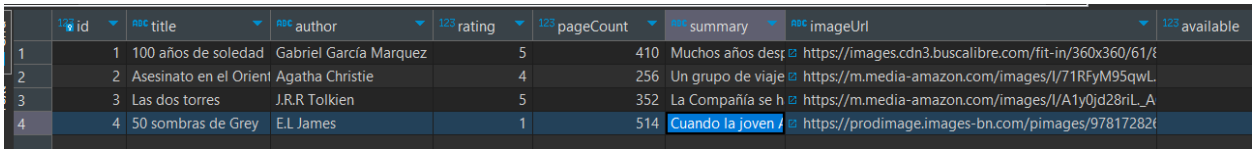


Observemos que algunos *data types* fueron cambiados ¿Por qué es esto? Bueno, *sequelize* debe comunicarse y aceptar la mayor cantidad de tipos de datos que existan. Si esos tipos de datos no existen en el motor los reemplaza. Por ejemplo, el *boolean* de *available* lo cambió por TINYINT, es decir, lo va a tratar como 1 o 0.

Métodos CRUD

Read

Para poder buscar o “leer” los datos de la tabla libros, primero la llenaremos manualmente con los datos que poseemos en nuestro lado cliente. Dbeaver nos permite directamente en la pestaña “Data” agregar los datos:

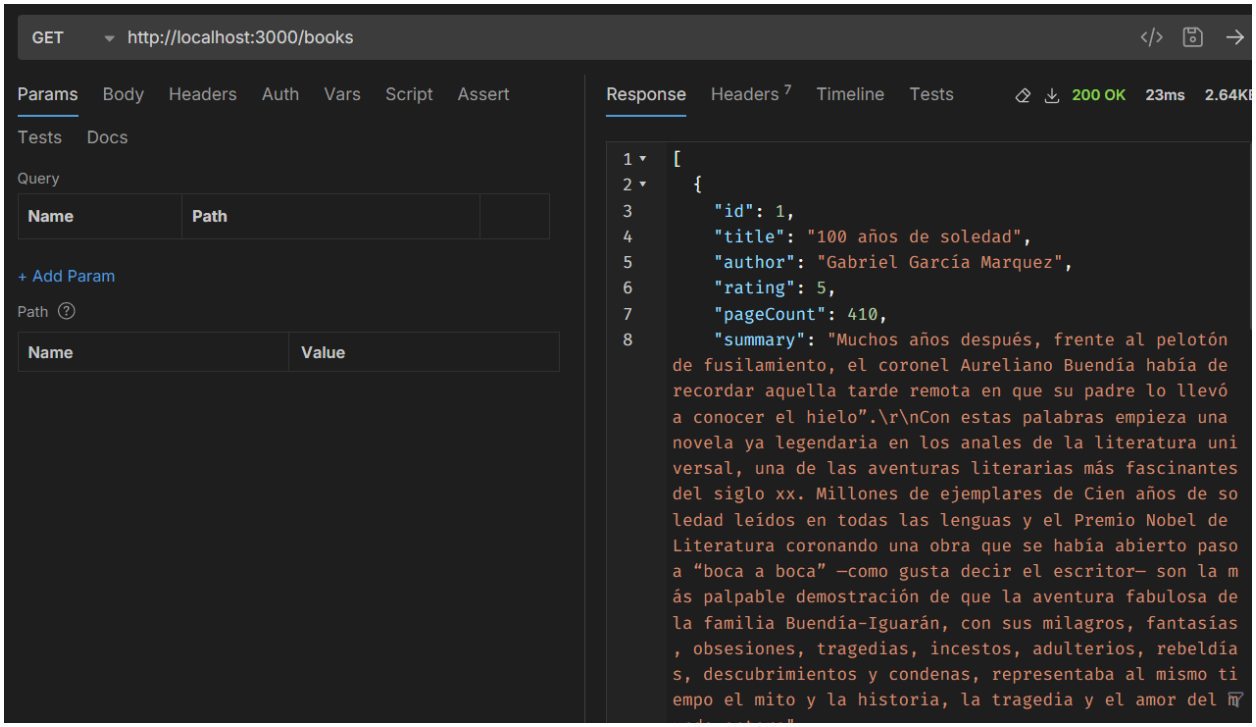


	id	title	author	rating	pageCount	summary	imageUrl	available
1	1	100 años de soledad	Gabriel García Marquez	5	410	Muchos años desp	https://images.cdn3.buscalibre.com/fit-in/360x360/61/8	
2	2	Asesinato en el Orient	Agatha Christie	4	256	Un grupo de viaje	https://m.media-amazon.com/images/I/71RFyM95qwL.	
3	3	Las dos torres	J.R.R Tolkien	5	352	La Compañía se h	https://m.media-amazon.com/images/I/A1y0jd28riL._A	
4	4	50 sombras de Grey	E.L James	1	514	Cuando la joven A	https://prodimage.images-bn.com/pimages/97817282	

Luego, debemos especificar el método de ORM en nuestro código que permite traer todos los libros. Sequelize posee una variedad de funciones que engloban sentencias de SQL para poder trabajarlas de forma más directa o sencilla.

```
7   router.get("/books", async (req, res) => {
8       const books = await Book.findAll();
9       res.json(books);
10  });
```

Agregamos async a nuestro segundo parámetro (la *callback*) y luego en vez de devolver un texto, devolvemos el JSON que nos provee sequelize. Verifiquemos su funcionamiento en Bruno:



También podemos modificar el método de búsqueda de un solo libro, utilizando la función *findByPk*:

```
12  router.get("/books/:id", async (req, res) => {
13      const { id } = req.params;
14      const book = await Book.findByPk(id);
15      res.json(book);
16  });
17
```

También podríamos usar el *findOne*, que funciona con un *where* que compara atributos (así podremos buscar por título o por autor, por ejemplo):

const book = await Book.findOne({ where: { title: bookTitle } });

Create

Para la creación de un nuevo libro, y basándonos en el modelo que creamos, debemos de alguna manera recibir **al menos** el título y el autor del libro, ya que estos campos no pueden ser nulos en nuestra tabla. Veamos cómo lo podemos armar.

Los pedidos que se realizan al servidor no sólo pueden traer información en los parámetros, sino que pueden traer información extra en lo que se conoce como **cuerpo(*body*)** de la *request*. **En el protocolo REST, esta es la forma más común de enviar información del lado cliente**.

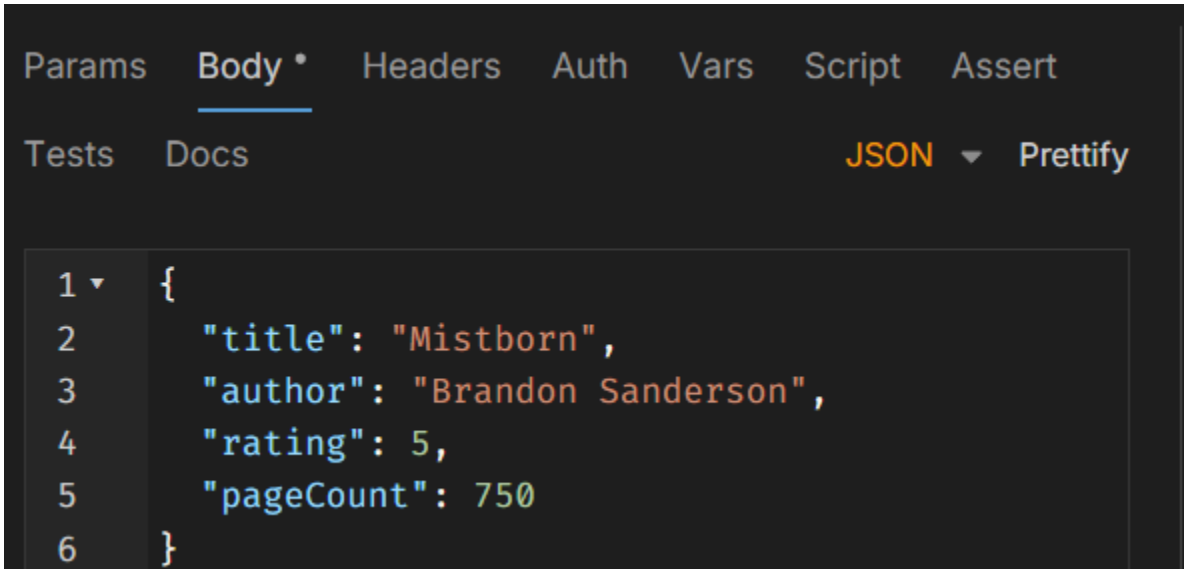
Modificamos nuestro código para que entonces tenga en cuenta esa supuesta información que va a venir en el cuerpo de la solicitud:

```
const { title, author, rating, pageCount, summary, imageUrl,available} = req.body;
```

Después, debemos especificarle a sequelize que debe crear una nueva fila en la tabla *books* con esa información.

```
const newBook = await Book.create({
    title,
    author,
    rating,
    pageCount,
    summary,
    imageUrl,
    available
})
res.json(newBook)
```

Dentro de Bruno, armaremos el JSON correspondiente:



Luego, nos falta agregar una configuración para que express pueda parsear correctamente el cuerpo de la *request*:

```
try {
    app.use(express.json())
    app.listen(PORT);
    app.use(bookRoutes);

    await sequelize.sync();

    console.log(`Server listening on port ${PORT}`);

} catch (error) {
```

Comprobamos entonces que podemos agregar libros mediante la solicitud POST a /books.

Update

Para la modificación de una entidad, debemos realizar dos operaciones:

1. La búsqueda de la entidad a actualizar.
2. La actualización en sí misma.

```
37  export const updateBook = async (req, res) => {
38      const { id } = req.params;
39      const { title, author, rating, pageCount, summary, imageUrl, available } = req.body;
40
41      // Find the book
42      const book = await Book.findByPk(id);
43
44      // Update it
45      await book.update({
46          title,
47          author,
48          rating,
49          pageCount,
50          summary,
51          imageUrl,
52          available
53      });
54
55      await book.save();
56
57      res.json(book);
58
59  };
```

Delete

Efectivamente, sequelize posee un método para la eliminación de la entidad llamado *destroy*.

```
63  router.delete("/books/:id", async (req, res) => {
64      const { id } = req.params;
65      const book = await Book.findByPk(id);
66
67      await book.destroy();
68
69      res.send(`Book with id: ${id} deleted`);
70  });
71
72  export default router;
```

Validaciones

Notemos que tanto la creación de una nueva entidad Book como su modificación, en ambas sino envíamos el campo *title* o el campo *author*, nos dará un error ya que estas propiedades son obligatorias. Podemos agregar una validación super simple de manera de que informemos correctamente al lado cliente **cuál** fue el error y **por qué** se produjo.

Para ello, devolveremos un código de error específico junto a un mensaje informativo. Hay una amplia variedad de códigos de error pero nosotros nos concentraremos en los siguientes:

- **400** (*bad request*): hay un error en la sintaxis de la request y el servidor no puede interpretar el pedido correctamente.
- **401** (*unauthorized*): una persona no logueada está solicitando recursos para los que necesita estar autenticado.
- **403** (*forbidden*): el solicitante no tiene los permisos requeridos para realizar la solicitud al servidor (relacionado con el rol de usuario)
- **404** (*not found*): el recurso solicitado no pudo encontrarse.
- **500** (*internal server error*): el servidor ha encontrado una situación que no sabe cómo manejarla.

Observando la lista anterior, el código de error más adecuado sería el **400**.

```
    // Title and author are required
    if (!title || !author)
        return res.status(400).send({ message: "Title and author fields are required" });
```

De esta manera, cortamos el método antes que siga su ejecución. Luego el cliente puede captar ese error y mostrar el mensaje aclaratorio.

Otra validación posible es aquella que informa al usuario que el libro solicitado no se encuentra. Para ello, utilizamos un código 404 luego de las búsquedas:

```
    if (!book)
        return res.status(404).send({ message: "Book not found" });
```

Establecimiento de *services*

Para emprolijar y organizar nuestro código, moveremos todo lo que sea lógica de negocio a un archivo llamado *book.services.js*, dentro de la carpeta *services* y las routes la dejaremos solo para el establecimiento de rutas que se comunican con el lado cliente.

src > services > book.service.js > deleteBook

```
import { Book } from "../models/Book.js";

export const findBooks = async (req, res) => {
    const books = await Book.findAll();
    res.json(books);
};

export const findBook = async (req, res) => {
    const { id } = req.params;
    const book = await Book.findOne({ where: { id: id } });

    if (!book)
        res.status(404).send({ message: "Book not found" });

    res.json(book);
}

export const createBook = async (req, res) => {
    const { title, author, rating, pageCount, summary, imageUrl, available } = req.body;

    // Title and author are required
    if (!title || !author)
        res.status(400).send({ message: "Title and author fields are required" });

    const newBook = await Book.create({
```

Luego, nuestro archivo de rutas / controladores nos quedará así:

src > routes > books.routes.js > ...

```
1   import { Router } from "express";
2   import { createBook, deleteBook, findBook, findBooks, updateBook } from "../services/book.service";
3
4   const router = Router();
5
6   router.get("/books", findBooks);
7
8   router.get("/books/:id", findBook);
9
10  router.post("/books", createBook);
11
12  router.put("/books/:id", updateBook);
13
14  router.delete("/books/:id", deleteBook);
15
16  export default router;
```

Finalmente, nos queda un arquitectura donde:

1. **Controladores**: se encargan de ser el punto de comunicación inicial y final con el lado cliente. Establecen la ruta.
2. **Modelos**: son una representación ORM del lado servidor para poder trabajar las entidades de la base de datos.
3. **Servicios**: van a contener la lógica de negocio que realiza la función requerida por el lado cliente.

Fecha	Versionado actual	Autor	Observaciones
10/01/2025	1.0.0	Gabriel Golzman	Primera versión