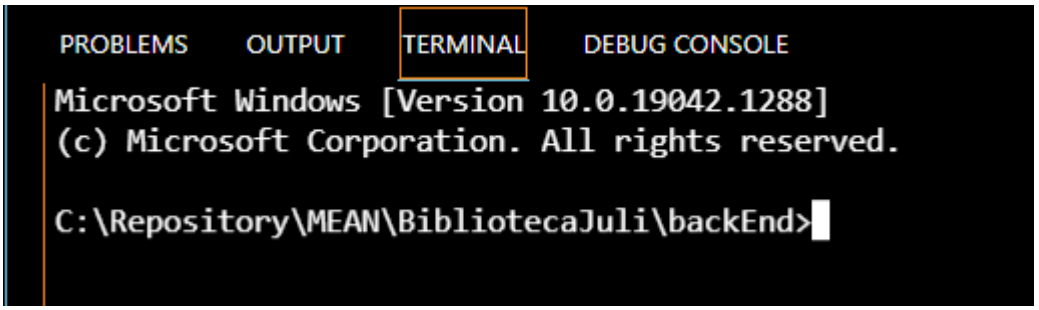
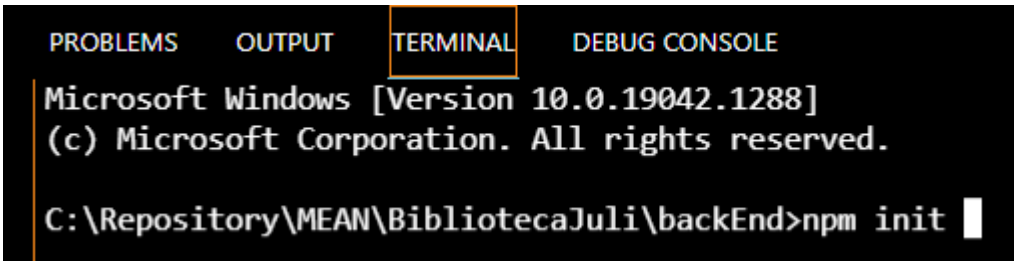
1. Creación del proyecto BibliotecaJuli y de carpeta BackEnd dentro.

1.1 Open terminal y validación ubicación carpeta BackEnd.



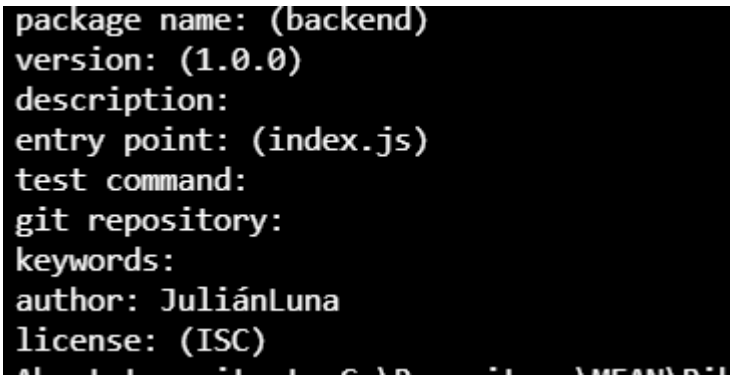
2. Comandos configuración proyecto en NoDeJs.

2.1 Inicializar el NoDeJS en proyecto con: **npm init**.

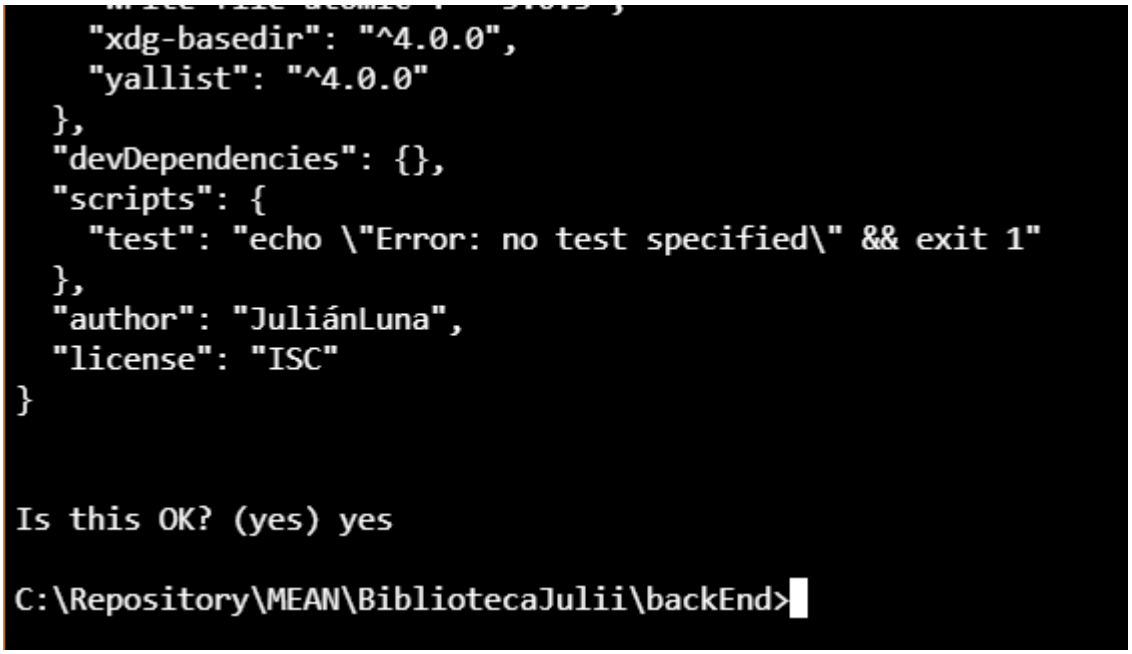


2.2 Inicia la configuración del proyecto con algunas preguntas (se pasa a la siguiente pregunta con ENTER).

2.2.1. **Package Name:** (backend): Nombre del paquete del proyecto / Lo dejé por defecto.
2.2.2 **Version (1.0.0)**: Versión del proyecto.
2.2.3 **Description:** Si se requiere agregar alguna descripción. Es libre.
2.2.4 **Entry point: (index.js)** :Pregunta si este es el archivo principal. index.JS es el archivo principal en BackEnd.
2.2.5 **Test command**: Comandos para pruebas o testing.
2.2.6 **Git Repository**: Opción para agregar RepositorioGitHub.
2.2.7 **Keywords:** En caso de que el proyecto tenga palabras clave.
2.2.8 **Author:** Nombre del autor del proyecto.
2.2.9 **License: (ISC).** Indica la licencia que tiene el proyecto. Recomendado en este caso ISC: Permite el libre uso. (Existen varios tipos de licencia.)

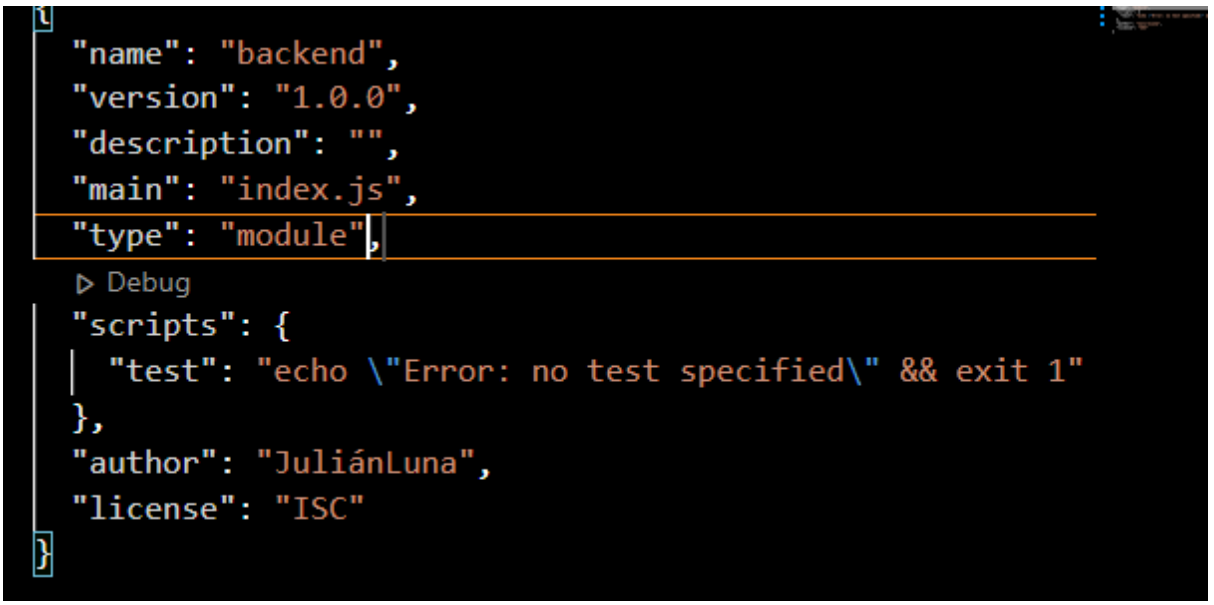


2.3. Al realizar todas la configuraciones anteriores nos pide la confirmación y si todo está en orden se indica **YES**, en caso de no estarlo confirmamos **NO** y se reinicia el proceso.



2.4 Al confirmar, se crea el archivo package.json donde se verán todas estas configuraciones. Archivo

2.4.1 Ingresamos el archivo package.json y bajo la configuración “main ”: “index.js” escribimos **“type”: “module”**
Esta configuración nos ayuda a definir el tipo de organización que tendrá nuestro proyecto.
Se selecciona la opción **module**, esto por ser la opción más moderna, considerada también como la opción considerada como buena práctica. Hace referencia a una organización de arquitectura de capas.



3. Instalación de librerías necesarias para el desarrollo del proyecto. Se inicia con el comando **npm** (todo dentro del mismo comando solo dando espacio entre librerías)**.** En el siguiente orden:

3.1.1 **i** ó **install** : instalar.
3.1.2 **express**: Para crear apps web y API.
3.1.3 **mongoose:** Va admin. mongoDB desde NoDeJS.
3.1.4 **cors.**Tiene todos los protocolos de comunicación y conexión entre el front y back. Intercambio de recursos y comunicación.
3.1.5 **dotenv:** Para la creación y manipulación de las variables de entorno.
3.1.6 **jsonwebtoken**: Encripta info de los users en tokens.
3.1.7 **bcrypt:** Encriptación de contraseñas.
3.1.8 **moment**: Maneja formatos de Dates(fechas) específicos.
3.1.9 **connect-multiparty**: Permite cargar archivos a la app.

```
C:\Repository\MEAN\BibliotecaJulii\backEnd>npm i express
 mongoose cors dotenv jsonwebtoken bcrypt moment connect
-multiparty
```

Enter:

```
[                    ] / idealTree: timing arborist:ctor
```

```
changed 8 packages, and audited 281 packages in 10s

22 packages are looking for funding
  run `npm fund` for details

3 moderate severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

3.2. Al instalarse las librerías pueden quedar algunas vulnerabilidades (tal como se muestra en la img anterior), con el comando **npm audit fix** se pueden arreglar.

```
C:\Repository\MEAN\BibliotecaJulii\backEnd>npm audit fix
```

Estos pueden no solucionarse y no pasa nada.

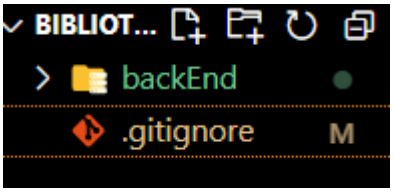
```
Severity: moderate
 Inefficient Regular Expression Complexity in chalk/ansi-
regex - https://github.com/advisories/GHSA-93q8-gq69-wqmw
fix available via `npm audit fix`
node_modules/ansi-regex
  strip-ansi  4.0.0 - 5.2.0
  Depends on vulnerable versions of ansi-regex
  node_modules/strip-ansi
    string-width  2.1.0 - 4.1.0
    Depends on vulnerable versions of strip-ansi
    node_modules/string-width

3 moderate severity vulnerabilities

To address all issues, run:
  npm audit fix
```

3.3. Instalación de otra librería. iniciando **npm i**:
3.3.1 **nodemon**: Para que los cambios se tomen en “RealTime ”
3.3.2 **-D:** Librería para desarrolladores.

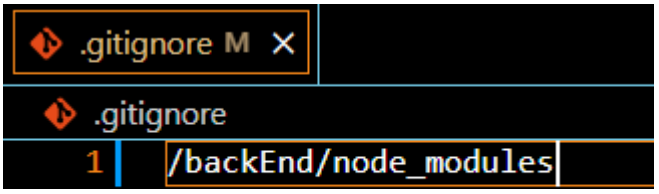
```
C:\Repository\MEAN\BibliotecaJulii\backEnd>npm i nodemon -D
```

4. Con el fin de no subir archivos que no sean necesarios a GitHub realizamos el siguiente proceso.

4.1 Creación de File fuera del folder backEnd.



4.1.2 Dentro del file indicar que archivos se ignorarán por parte de GitHub. (Ubicación exacta con nombre exacto)



5. Conectar DB

Dentro del folder BackEnd crear otro folder con nombre: **db** y un file dentro de este llamado **db.js**. Allí dentro iniciamos la conexión de la **DataBase.**

5.1 Importar la librería que administra MongoDB desde NoDeJS.

```
1  import mongoose from "mongoose";
```

5.2 Crear función la cual va ayudar a crear la conexión entre el BackEnd y la DataBase.

```
3  const dbConnection = () => {
```

5.2.1 Dentro la función anterior agregamos trycatch. Este nos informará los errores que salgan al momento de realizar la conexión con DB.

```
try {

} catch (error) {

}
```

5.2.2 En try registramos:

```
try {
  mongoose.connect(process.env.DB_CONNECTION)
} catch (error) {
```

Aquí vamos a conectarnos a la librería. Process es la variable que nos permite conectar con los distintos archivos. En este caso, **proces.env** nos permitirá ingresar a las variables de entorno específicamente la .DB_CONNECTION. / Para que esto funcione continuar al siguiente paso./

5.2.3 Crear file dentro de la backEnd pero fuera de db.
Este file lleva por nombre .env y contendrá las variables de entorno.

backEnd > .env

```
PORT=3001
DB_CONNECTION=mongodb://localhost:27017/bibliojulidb
SK_JWT=T0511DB
```

5.2.4 Creación de callback como parámetro dentro try en función dbConnection.

useNewUrlParser : Hace que la url no sea visible en consola.
useUnifiedTopology: Hace que la escritura de la consola sea entendible para nosotros.

```
mongoose.connect(process.env.DB_CONNECTION, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
```

5.2.5 Crear mensajes de consola.

El primero dentro del try por si todo sale OK y el segundo dentro del catch, quien recibe los errores y envía entonces su mensaje.

```
  console.log("Connection MongoDB : ON ");
} catch (failed) {
  console.log("Unable to connect MongoDB \n " + failed);
}
```

5.2.6 Hacer que la función se asíncrona para que pueda ejecutar varias funciones a la vez. **async y await.**

```
const dbConnection = async () => {
  try {
    await mongoose.connect(process.env.DB_CONNECTION, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });

    console.log("Connection MongoDB : ON ");
  } catch (failed) {
    console.log("Unable to connect MongoDB \n " + failed);
  }
};
```

5.3 Exportar la función.

```
export default { dbConnection };
```

6. Creación de servidor.

Creación de file index.js. Aquí se toman las peticiones del front y se llevan al backend.

6.1 Importar las librerías que se utilizarán aquí.

```
import express from "express";
import cors from "cors";
import db from "./db/db.js";
import dotenv from "dotenv";
```

6.1 Al agregar la función a continuación logramos que cuando se ejecute.env se guarden las variables en la RAM.

```
dotenv.config();
```

6.2 Agregamos la variable principal de nuestro servidor.

```
const sft = express();
```

6.3 Estas configuraciones en el servidor nos indican que: **express.json** servidor solo utilizará datos json y **cors** para entender las peticiones que vienen del front.

```
sft.use(express.json());
sft.use(cors());
```

6.4 Conexión con el puerto: **listen**
Aquí **proces.env** nos permite escuchar a las variables de entorno específicamente el Port.
Console.log mensaje para confirmación de que si conectó.

```
sft.listen(process.env.Port, () => console.log("Server: " + process.env.PORT));
```

6.5 Para conectar todo finalmente llamamos la conexión de MongoDB.

```
db.dbConnection();
```

6.6 Para poder ejecutar el servidor vamos a package.json y realizamos los siguientes ajustes. para que lo cambios salgan en real time y estén en el index.js

Pasando de:

```
"test": "echo \"Error: no test specified\" && exit 1"
```

Cambiandolo a:

```
"start": "nodemon index.js"
```

7. Validación de la conexión.

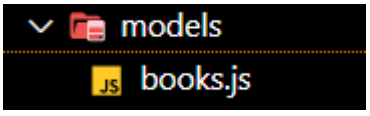
En terminal comando **npm start**. (debemos tener abierto el servidor **Mongod**).

```
Microsoft Windows [Version 10.0.19042.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Repository\MEAN\BibliotecaJulii\backEnd>npm start

> backend@1.0.0 start
> nodemon index.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Server: 3001
Connection MongoDB : ON
```

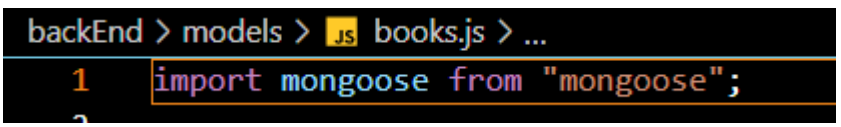
Creación de servicios de la app.

8. Una vez validados los esquemas iniciamos con la creación de estos.

8.1 Crear dentro BackEnd la carpeta models y dentro de esta los archivos .js según el esquema de planeación. Crear archivo books.js dentro carpeta de models.



8.2 Importar librería mongoose para administrar la DB desde NoDeJS.



8.3 Creación de esquema. Creando una variable constante y asignado a la variable new mongoose con la función Schema para crear un esquema nuevo.

```
const bookSchema = new mongoose.Schema({
```

8.4 Creación de la estructura. Agregamos los datos primitivos según la planeación y las buenas prácticas actuales.
En el caso de registerDate este no es un dato primitivo y necesitamos que registre la fecha actual del sistema, por eso se registra de la forma en la que vemos la imagen.

```
  name: String,
  description: String,
  dbStatus: Boolean,
  registerDate: { type: Date, default: Date.now },
});
```

8.5 Ahora, fuera de la función bookSchema creamos una línea de código que recoja la información anterior.

```
const library = mongoose.model("books", bookSchema);
```

Específicamente lo que estamos haciendo aquí es: en la constante library vamos a guardar desde la liberia mongoose utilizando la función model, la cual crea un collection la cual vamos a llamar books, guardando la función bookSchema en la collection.

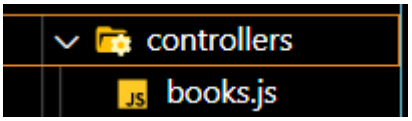
8.6 Exportar la variable library para utilizar en otros documentos. Se exporta sin llaves por ser una variable y no una función.

```
export default library;
```

9. Funcionalidad del proyecto.

Comunica si los datos se están guardando, siendo consultados, eliminados o editados.

9.1 Crear dentro de backEnd el Folder Controllers, se llama así puesto que es el archivo que va a controlar la funcionalidad del proyectos, crear file books.js.



9.2 Iniciamos la primera línea de código importando la collection realizada en el paso anterior, puesto que aquí se validará el funcionamiento de la misma.

```
import library from "../models/books.js";
```

9.3 Creamos la función saveBook. Aquí específicamente se validará el correcto funcionamiento de registro y lo que sucederá en caso de errores.

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

La función es **async** para que se ejecute al mismo tiempo que otros procesos.
req: significa lo que él recibe / **res**: lo que este devuelve.

9.4 Dentro de la función iniciamos las validación primaria. Datos completos, no vacíos. Para este proceso utilizamos condicionales.

```
if (!req.body.name || !req.body.description)
return res.status(406).send("Incomplete data");
```

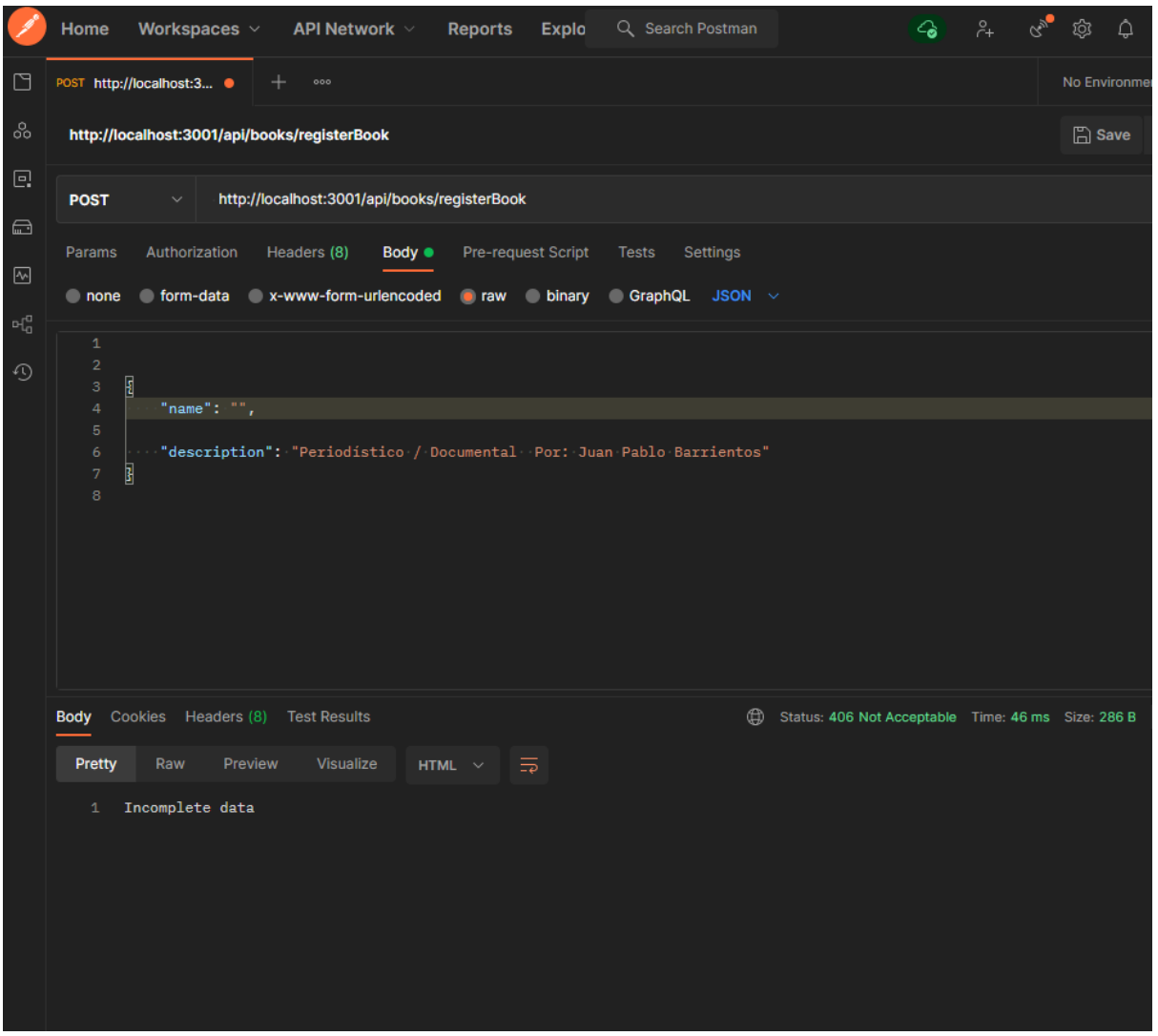
req: recibe **body**: cuerpo de lo que recibe(json)
.name ó **.description:** campo del sistema a validar
!: Hace que la condición se niegue directamente, es decir esta pasa de true a false .

```
return res.status(406).send("Incomplete data");
```

return: es importante pues este reinicia el programa hasta que sea correcto continuar. En caso de no estar y de cumplirse la condición el programa continuará ejecutándose.

res: devuelve **status:** función del servidor **(406):** Es un código de error del cliente y en este caso significa específicamente que **No es aceptable** la información proporcionada. Hay varios códigos y dependiendo de cada caso se pueden aplicar 400 y 404 es de los más famosos.

Decimos entonces que si **req.body.name** o **req.body.description** están vacíos nos **res** el mensaje **“Incomplete Data”**



9.4 Continuando con las validaciones, igualmente dentro la misma función iniciamos la creación de una variable que nos permitirá saber si el libro ya se encuentra registrado.

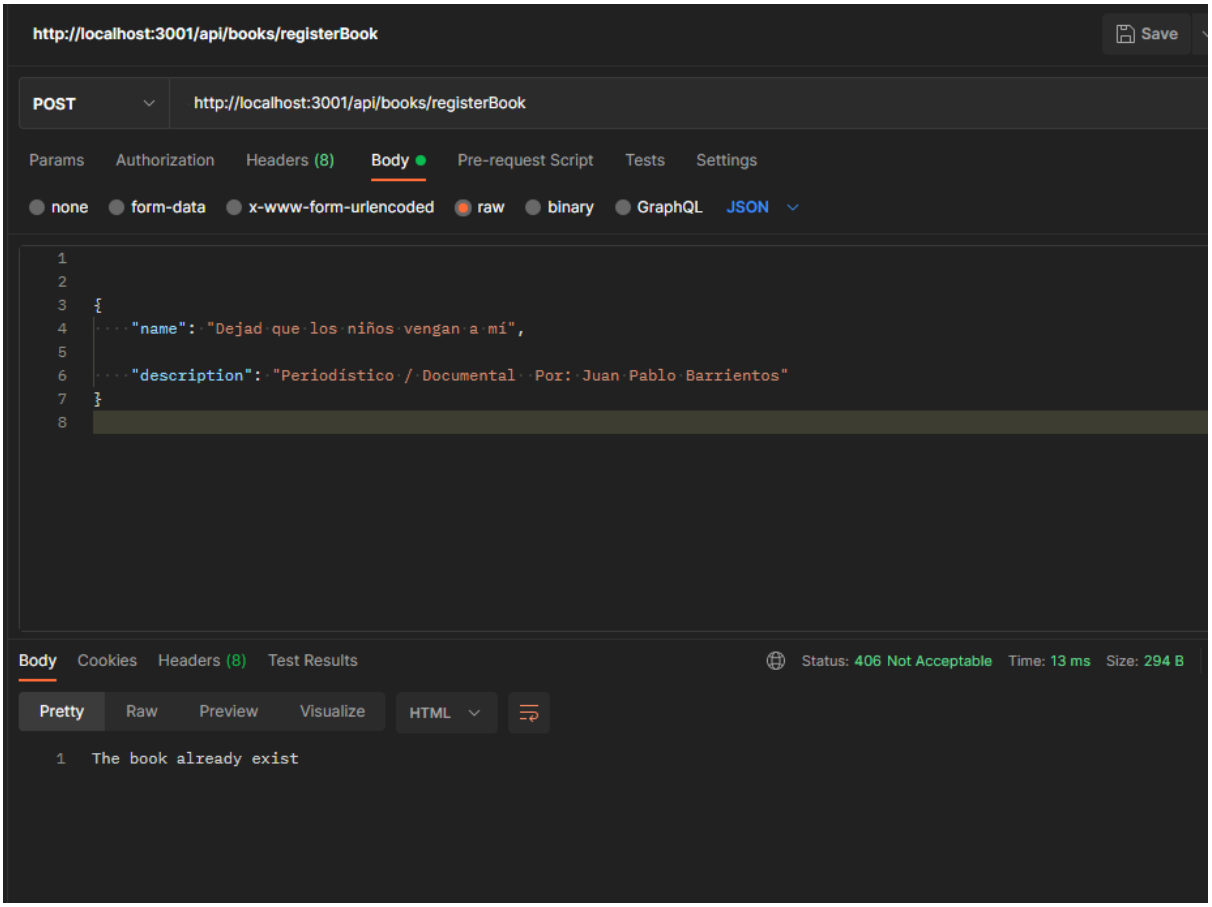
```
const bookInStock
```

… esto asignándole la promesa **await** para ejecutar la tarea, prometiendo ir a el registro de la DB **library** utilizando también la función **findOne** la cual busca una coincidencia para poder continuar, en este caso esta coincidencia está siendo validada con el campo **name:**.

```
const bookInStock = await library.findOne({ name: req.body.name});
```

Es necesario demostrar que el libro ya está y nos podemos valer de una condicional nuevamente.

```
if (bookInStock) return res.status(406).send("The book already exist");
```



9.5 Ahora necesitamos una variable constante que nos ayude creando un nuevo esquema en caso de que la info dada sea válida. (es decir que no cumpla con los puntos 9.3 y 9.4). Dentro de la función que venimos trabajando creamos:

```
const bookSchema = new library ({
```

name ó description: lo que recibimos lo va a agregar en el campo, bien sea name ó description.

```
const bookSchema = new library ({
    name: req.body.name,
    description:req.body.description,
    dbStatus: true,
});
```

9.6 Es necesario guardar el proceso 9.5 en la DB.

```
const result = await bookSchema.save();
```

Si después de todo el esquema no se guarda continuamos a informar esto con otra condicional. En donde sí **result** llega vacío (es false) pues el sistema **res** el mensaje de falla.

```
if (!result) return res.status(400).send("Failed to register book");
```

Por otro lado, si el sistema logra guardar de forma correcta la info en **result** tenemos el siguiente proceso.

```
res.status(200).send({ result });
```

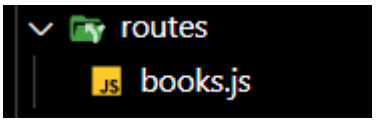
9.7 Exportar la función en la que venimos trabajando para poder utilizarla en lo que resta del proceso continuando a las rutas para consumir la API.

```
export default { saveBook };
```

10. Rutas.

Aquí encontraremos las rutas url API a consumir

10.1 Crear dentro BackEnd el Folder routes con el file books.js



10.2 Importar las librerías y archivos necesarios.

```
1   import express from "express";
2   import books from "../controllers/books.js";
```

10.3 Se necesita que la API tenga conexión con el servidor por lo tanto:
Creamos la variable router para que el servidor tome las rutas por eso es **expres.Router. Router:** función de express.

```
const router = express.Router();
```

```
router.post("/registerBook", books.saveBook);
```

Aquí indicamos la función a realizar router.**post**: para registrar.
Al validar el link permite el acceso al archivo importado en el paso 10.2 y continuando con el proceso que ahí se encuentra.

10.4 Exportar la función.

```
export default router;
```

11. Conexión final.

En el archivo index.js continuamos con la instalación final del proceso.

11.1 Importar las rutas Api.

```
import books from "./routes/books.js";
```

11.3 Especificación de rutas para ir a consumir carpeta routes.

```
sft.use("/api/books", books);
```

A partir del servidor BackEnd el cual se comunica con protocolos tenemos el siguiente link. **http://localhost:3001/** continuando con la dirección especificada en **index.js** para que el **sft (nombre dado a la app)** utilice **/api/books** llevándonos a routes completando el link con **/registerBook** y de ahí vaya realizando los proceso.