**Prueba. APIs - Ejercicio 1 (Solución)**

**Entorno de Trabajo**

* **Editor utilizado:** Visual Studio Code.
* **Control de Versiones (Local):** Git.
* **Control de Versiones (en la nube):** GitHub.
* **Sistema Operativo:** Windows 64bits.
* **Navegador:** Firefox.

**Desarrollo**

* Para empezar, inicializamos el repositorio git local.
  + **git init**
  + **git add -A**
  + **git commit -m "initial"**
* Utilizamos el manejador de paquetes “npm” para instalar “express”, “swagger-jsdoc”, “swagger-ui-express” y “nodemon”. Express es el framework backend que utilizamos para crear el servidor. Nodemon es una herramienta que se encarga de reiniciar el servidor cada vez que realizamos un cambio en el código del mismo. Swagger es una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs.
  + **npm init -y**
  + **npm install express swagger-jsdoc swagger-ui-express**
  + **npm install nodemon -S -D**

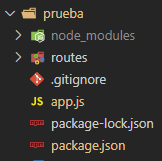
El segundo comando instala las dependencias “swagger-jsdoc” y “swagger-ui-express”, que nos premiten implementar Swagger a nuestro proyecto.

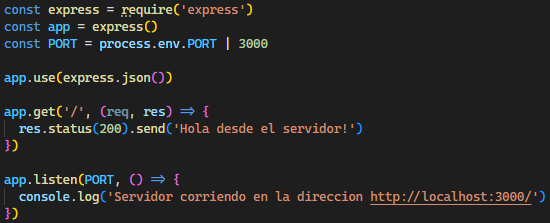
Con este último comando instalamos “nodemon”, donde “-S” guarda la versión de la dependencia y “-D” instala la dependencia como una dependencia de desarrollo.

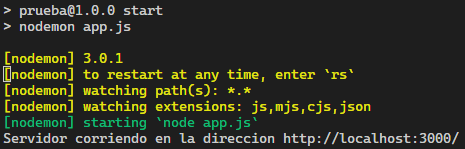
* Creamos un script en nuestro archivo package.json para poder correr el servidor con nodemon. Este script tiene como nombre “start”.

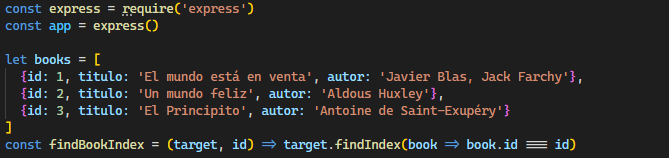


* Se crea el repositorio remoto Github llamado “[Deusto-Tema6-APIs-Prueba](https://github.com/UnshiftedSet54/Deusto-Tema5-NodeJS-Prueba)” y se enlaza con el repositorio local git.
  + **git remote add origin https://github.com/UnshiftedSet54/Deusto-Tema6-APIs-Prueba.git**
* Creamos el fichero app.js en la ruta del proyecto y un directorio llamado “routes”. Además, creamos la estructura básica del servidor. Donde definimos el puerto del servidor (en este caso 3000) y creamos un método “get” para la ruta “/” que devuelve el mensaje “Hola desde el servidor!” con el código de status 200. Por último, imprime un mensaje en la consola que dice “Servidor corriendo en la dirección http://localhost:3000/” al correr el comando: **npm start.**

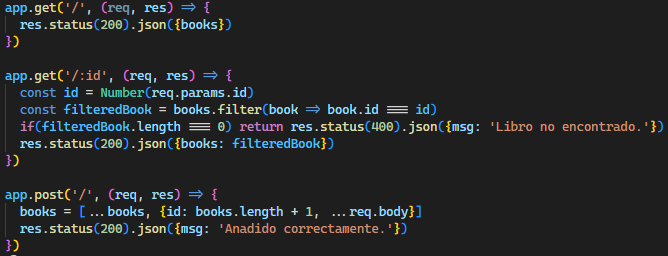






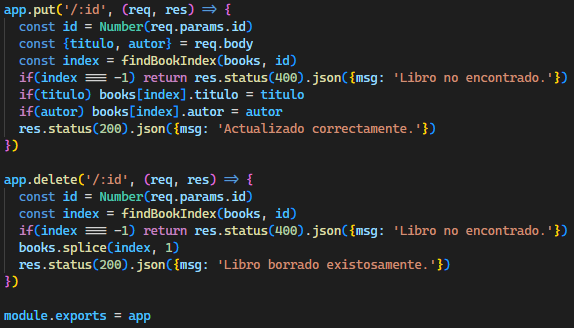
* En el directorio “routes” creamos un fichero “books.js” donde colocamos los métodos “get”, “post”, “put” y “delete”, que se implementaran para la ruta <http://localhost:3000/Books>. Creamos un arreglo de objetos llamado “books”. Cada objeto tendrá las siguientes propiedades: “id”, “nombre” y “autor”. Creamos una función llamada “findBookIndex” que nos permitirá encontrar el índice (index) de un determinado objeto dentro del arreglo, partiendo de un “id”. 

Para crear la función “findBookIndex” usamos el método “findIndex” y nos retorna un arreglo con todos los objetos que cumplan con la condición de que el “id” de cada objeto coincida con el “id” proporcionado.



Para obtener un libro en específico, hacemos uso del método “get” al cual le pasamos la ruta con el parámetro id (‘/:id’). Obtenemos este parámetro usando “req.params.id” y usamos Number() para convertir este parámetro obtenido de “string” a “number”. Se devuelve un mensaje de error con status 400 en caso de que no se consiga el libro. Si se consigue el libro se devuelve un objeto con el libro conseguido con status 200.

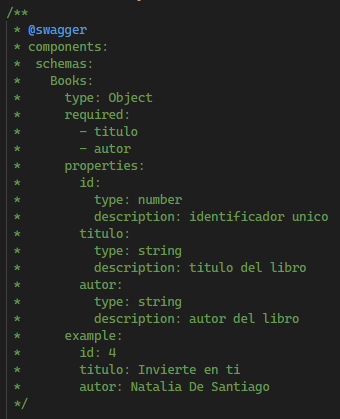
Para el método “post” usamos el operador spread (…books) para ingresar todo el arreglo “books” y le añadimos el objeto nuevo que con un “id” nuevo y los datos proporcionados en la petición (…req.body).



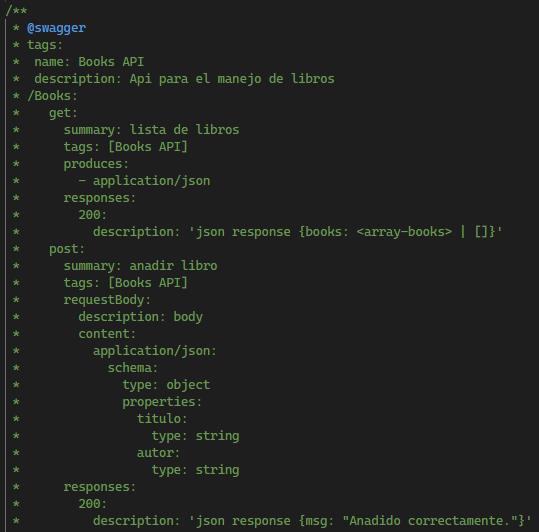
En el método put usamos la desestructuración para obtener los valores de “título” y “autor”, a partir de req.body. En caso de no encontrar un libro se retorna un mensaje de error. Si se consigue el libro, se actualiza el título y/o autor de ser necesario y se retorna un mensaje exitoso con status 200.

Module.exports nos permite modularizar el código para ser utilizado en otros lados del programa.

* Por último, implementamos, en el fichero books.js, los comentarios de Swagger para poder generar la documentación de la API. Estos comentarios se agregar al inicio del fichero.



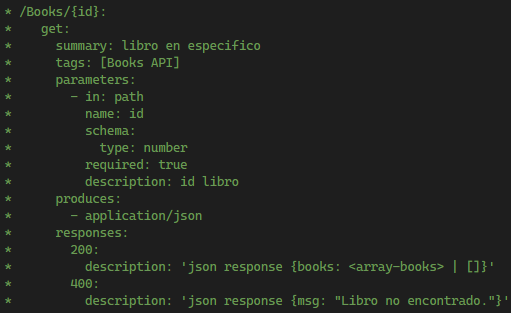
En este apartado iniciamos Swagger y definimos nuestro componente y su esquema en este caso “Books” que es de tipo Objeto y tiene dos propiedades requeridas: “titulo” de tipo string y “autor” de tipo string, además, tiene una propiedad no requerida llamada “id” de tipo number. Proveemos un ejemplo del esquema.

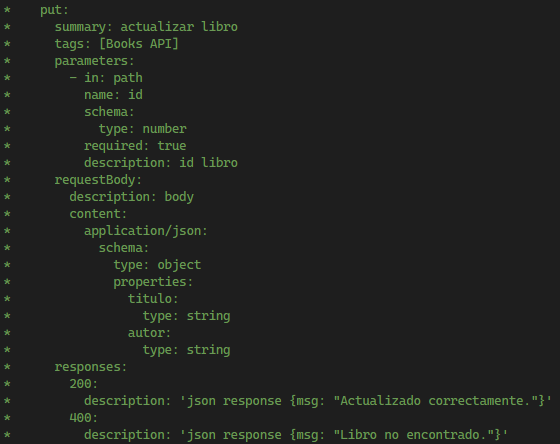


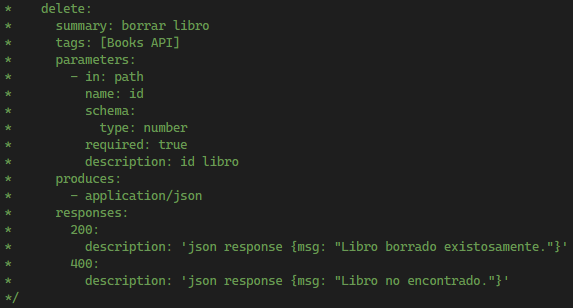
Iniciamos con @swagger y definimos un “tag” o etiqueta llamada “Books API” y proporcionamos una la descripción “Api para el manejo de libros”.

Luego, realizamos la documentación de cada ruta. En este caso la ruta “/Books” tenemos dos métodos: get y post. Para el método “get” le proporcionamos una descripción (lista de libros), la etiqueta ([Books API]), lo que produce (application/json) y su respuesta con el status y una descripción (status 200, json response {books: <array-books> | []}).

En el método ‘post’ proporcionamos la descripción ‘añadir libro’, la etiqueta ([Books API]) y el ‘requestBody’ que contiene un esquema de tipo ’object’ con el ‘titulo’ de tipo ‘string’ y el ‘autor’ de tipo ‘string’.



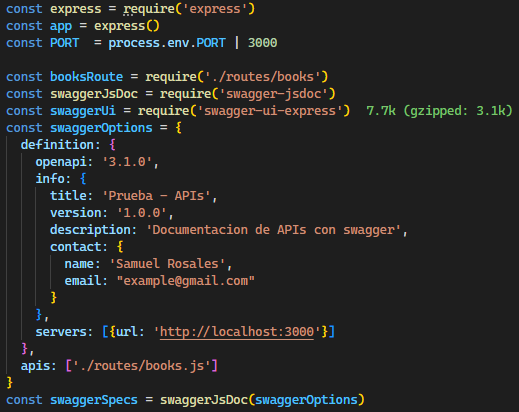




Para la ruta “/Books/{id}”, en los métodos ‘get’ y ‘delete’ definimos el parametro “id” estableciendo que el parámetro deseado está en la ruta (- in: path), tiene el nombre de “id”, es de tipo “number”, es un parámetro requerido y tiene la descripcion “id libro”.

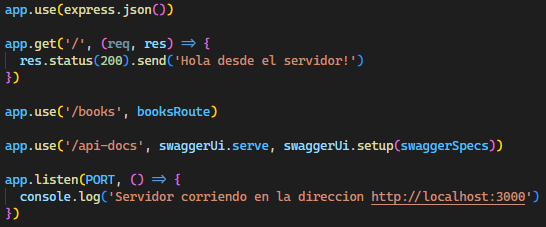
En el caso del método ‘put’, definimos los parámetros en el ‘id’ (- in: path). El ‘requestBody’ contiene un esquema de tipo ‘objeto’ con un título y autor de tipo ‘string’.

* Modificamos el fichero app.js para configurar e implementar Swagger y añadir la ruta de los libros (Books).



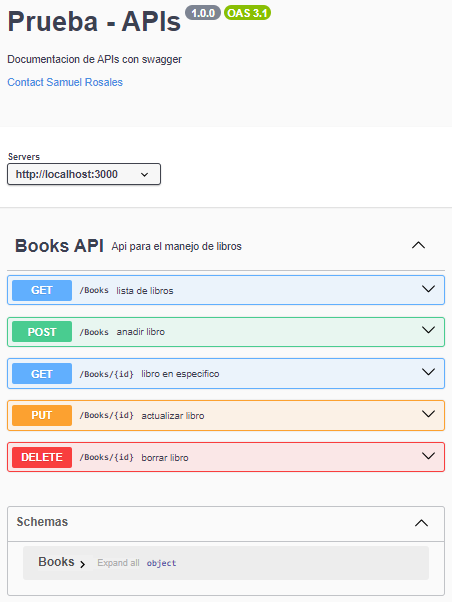
Importamos el fichero books.js (const booksRoute = require(‘./routes/books’)), importamos swagger-jsdoc (const swaggerJsDoc = require(‘swagger-jsdoc’)), importamos swagger-ui-express (const swaggerUi = require(‘swagger-ui-express’)). Creamos un objeto con las opciones necesarias: definimos la versión de openapi que utilizaremos (3.1.0), información del proyecto (título: Prueba – APIs, versión: 1.0.0, descripción: Documentación de APIs con swagger e información de contacto), los servidores (la dirección de nuestro servidor en ejecución) y la ruta de nuestra API (apis: [‘./routes/books.js’]).

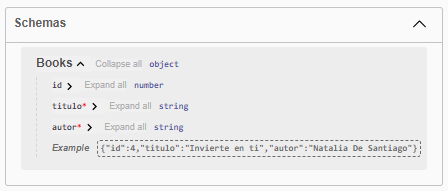
Luego le pasamos estas opciones a const swaggerSpecs = swaggerJsDoc(swaggerOptions).



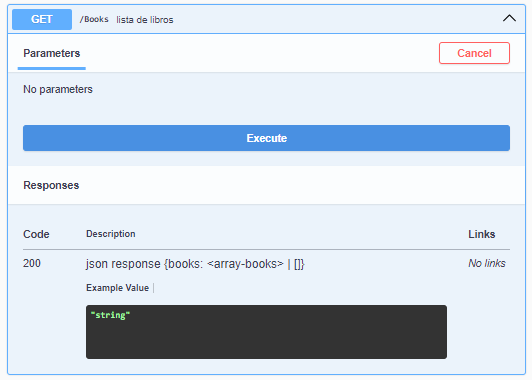
Implementamos la booksRoute con la ruta ‘/books’ (app.use(‘/books’, booksRoute)). Creamos la ruta ‘/api-docs’ y, le pasamos swaggerUi.serve y swaggerUi.setup(swaggerSpecs) que son las opciones de interfaz de usuario de swagger.

* Por último, visitamos la dirección ‘http://localhost:3000/api-docs/’, nos aseguramos que la documentación sea correcta y realizamos pruebas de la API.

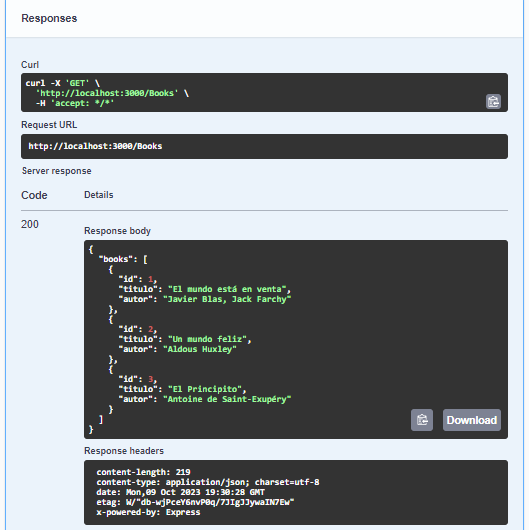




Al entrar a la dirección, nos encontramos con la el título del proyecto, una descripcion del proyecto y la información de contacto. Luego tenemos una lista de servidores. Después, encontramos la documentación de la API con el título, descripción de la API y cada uno de los métodos documentados en el fichero books.js. Por último, tenemos el schema definido y un ejemplo del mismo.

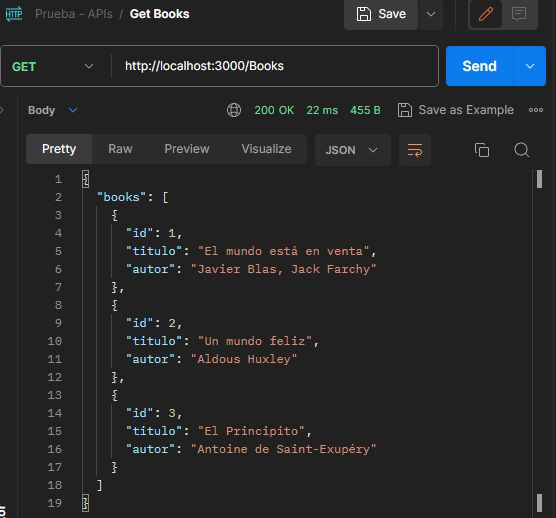


Para el este método get tenemos que la ruta es ‘/Books’ con la descripcion ‘lista de libros’, los parámetros requeridos (en este caso no tenemos) un botón de ejecutar y la descripción de las repuestas posibles.

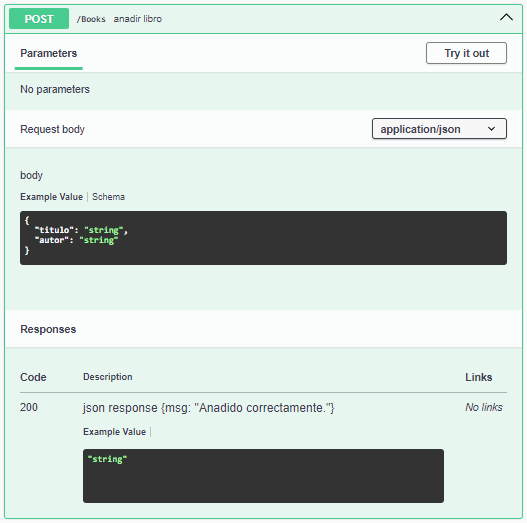


Al hacer click en el botón de ejecutar nos salen los datos de la petición que realizo swagger: el comando que utilizo para ejecutar la petición, url de la petición (http://localhost:3000/Books), la respuesta del servidor (un objeto con el arreglo de books) con su código de status (200) y los response headers.

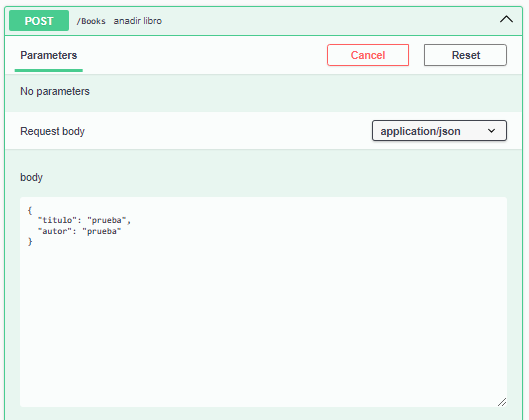
Verificamos los resultados haciendo una prueba en postman:



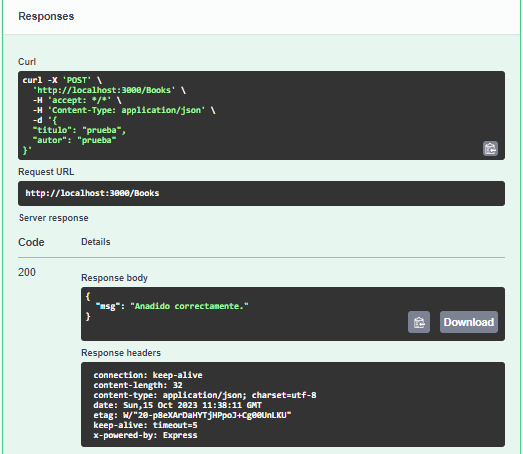
Los resultados obtenidos concuerdan con la documentación.



Para el método “post” de la ruta ‘/Books’, tenemos la descripción ‘añadir libro’. En este caso los parámetros (título y autor) de los pasamos en el ‘body’ en formato ‘application/json’ y la respuesta documentada es un json con un mensaje exitoso y status 200.

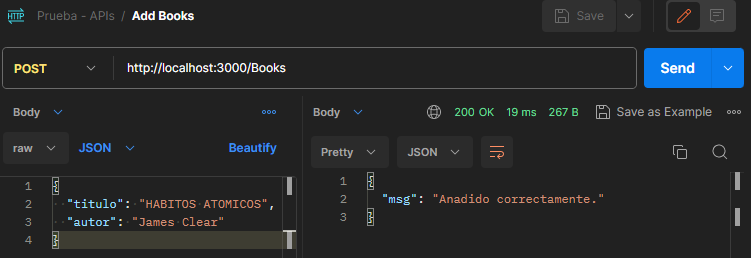


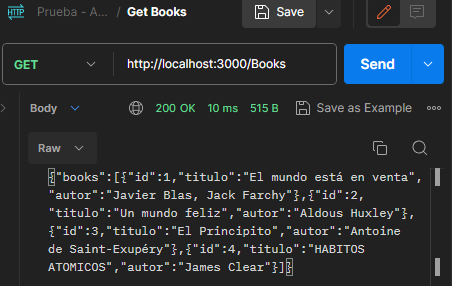
Para realizar la prueba le pasamos un json con el título igual a ‘prueba’ y autor del libro igual a ‘prueba’.



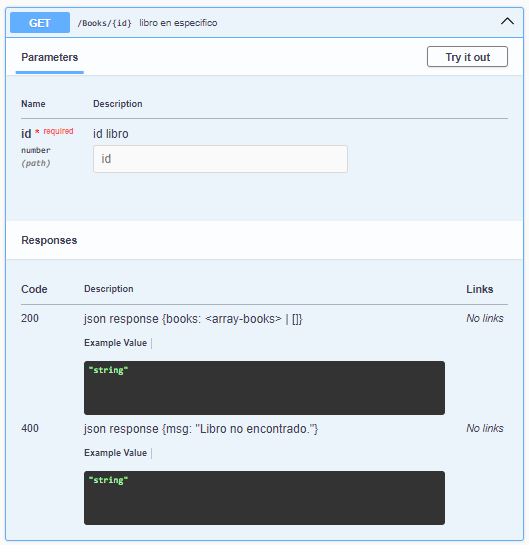
Al ejecutar la prueba, tenemos que la url usada es ‘http://localhost:3000/Books’ y la respuesta es un objeto con un mensaje exitoso y status 200.

Verificamos los resultados obtenidos con postman:

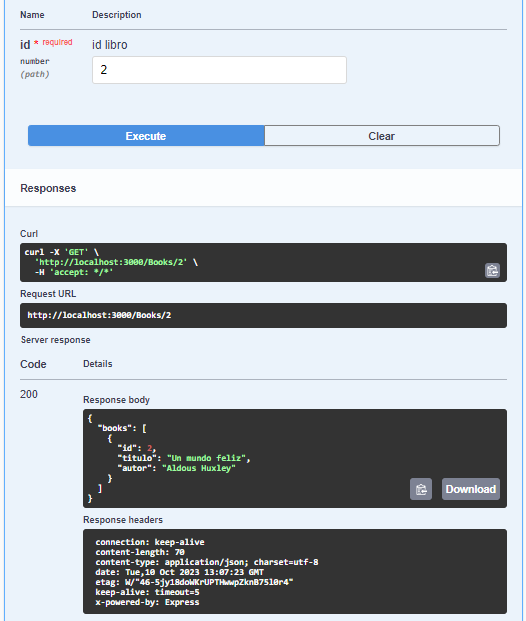




Nos damos cuenta que el libro se ha añadido correctamente y obtenemos un mensaje exitoso. Concordando con la documentación de la API.

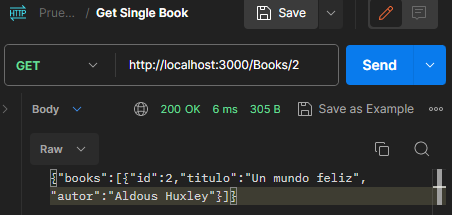


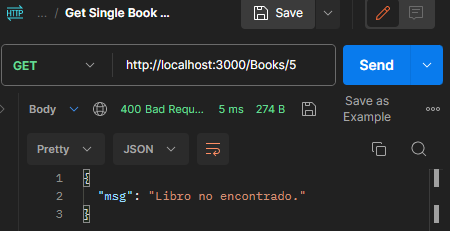
Para el método ‘get’ de la ruta ‘/Books/{id}’, la descripción es libro en específico, tenemos un parámetro ‘id’ requerido de tipo “number” y las repuestas son: en caso de ser exitoso, un json con el libro solicitado y status 200 y, un mensaje de error con status 400.



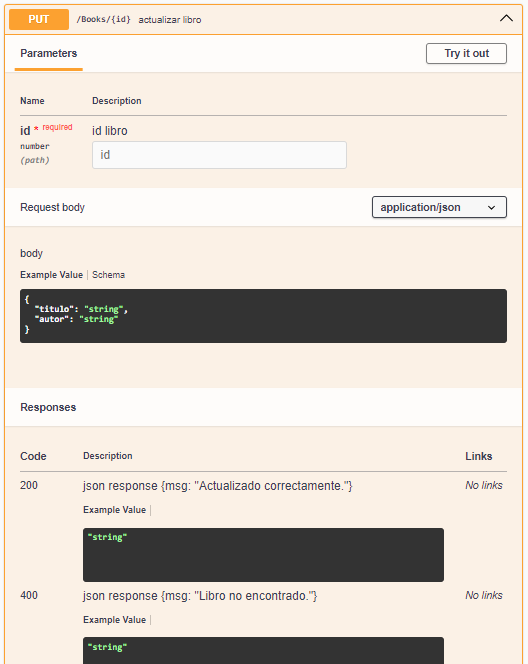
Para esta prueba, proporcionamos el valor de ‘id’ igual a 2. Esto hace que el url utilizado para realizar la petición es ‘http://localhost:3000/Books/2’. La respuesta obtenida es un objeto con el libro solicitado.

Procedemos a verificar los resultados en postman:



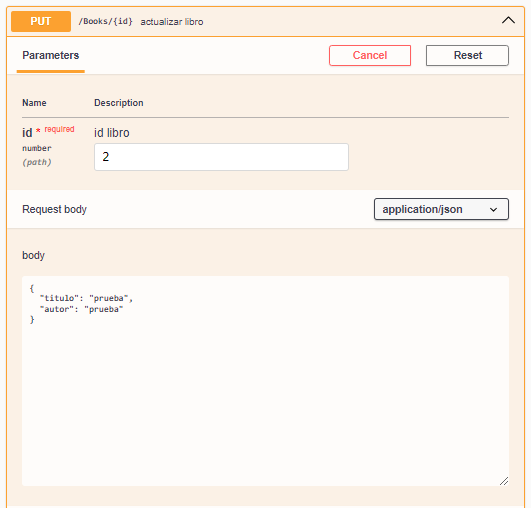


A partir de esta prueba obtenemos que en casi de ser exitoso se retorna el libro solicitado y en caso de no encontrar el libro retorna un mensaje de error, tal como se especifica en la documentación.

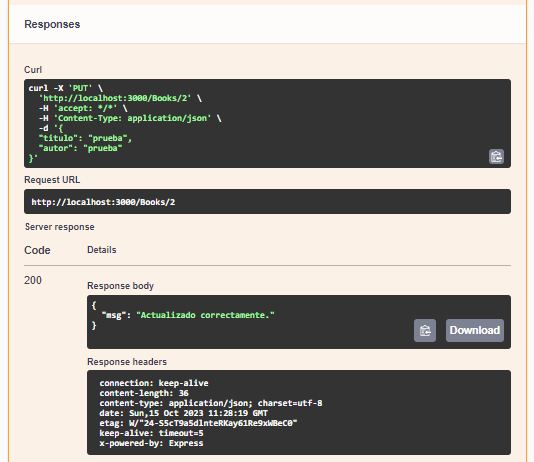


Para el método PUT de la ruta ‘/Books/{id}’, la descripción es ‘actualizar libro’, un parámetro proporcionado en la url es ‘id’ de tipo number y un ‘request body’ de tipo ‘application/json’, donde se proporciona el título y autor deseado.

Esta petición tiene dos respuestas posibles: en caso de ser exitoso se retorna un json con un mansaje de éxito y un status de 200. Si no es exitoso retorna un json con un mensaje de error con status 400.

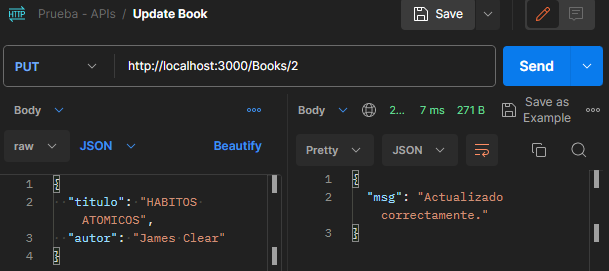


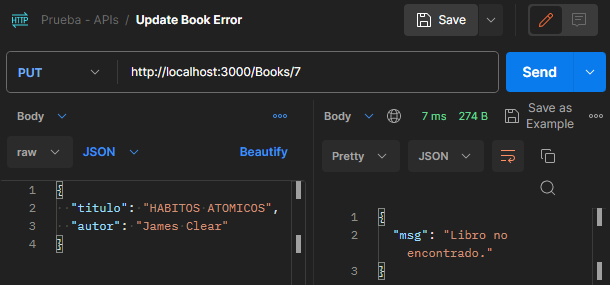
Par realizar la prueba le pasamos el parametro ‘id’ igual a 2 y el ‘body’ en formato json con el titulo igual a ‘prueba’ y el autor igual a ‘prueba’.



Al ejecutar la prueba tenemos que la respuesta es satisfactoria y contiene un mensaje exitoso con status 200.

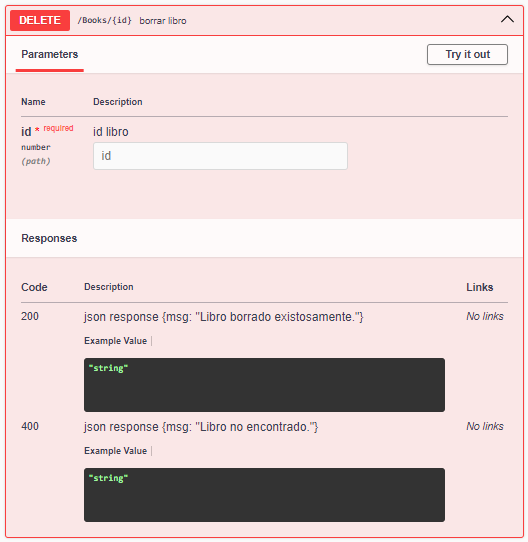
A continuación, realizamos las pruebas de las rutas en postman:





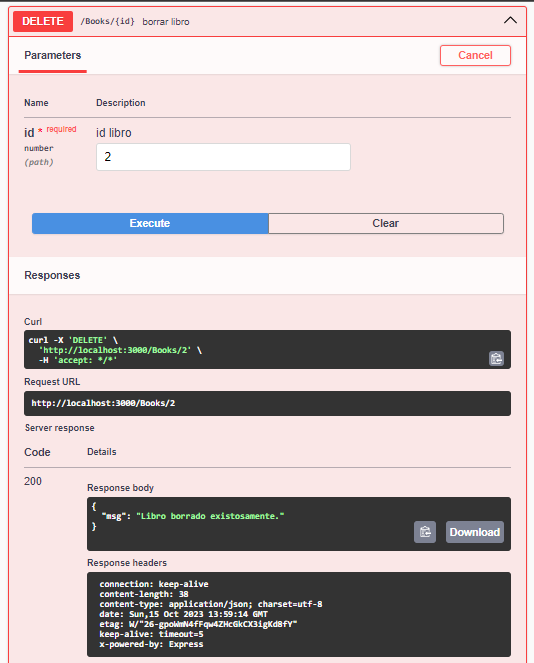


A partir de estas pruebas, tenemos que si la petición es exitosa nos retorna un mensaje de éxito con status 200. Si no se encuentra el libro se retorna un mensaje de error con status 400. En la ultima prueba podemos comprobar que se a actualizado el libro con id: 2 correctamente.



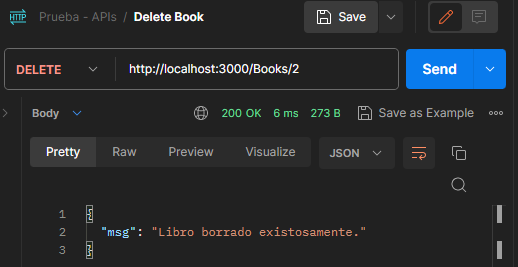
Por último, para el método ‘delete’ de la ruta ‘/Books/{id}’, tenemos que la descripción es ‘borrar libro’, el parámetro es ‘id’ de tipo ‘number’ y tiene dos posibles respuestas.

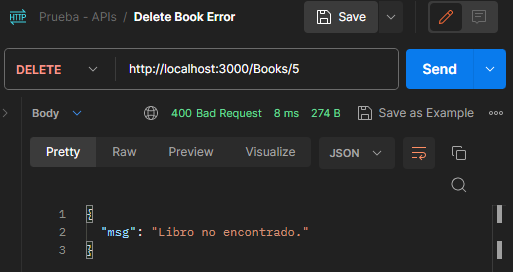
Si la petición es exitosa retorna una respuesta exitosa (status 200) con un mensaje de éxito con. Si la petición no es exitosa, retorna un mensaje de error con status 400.

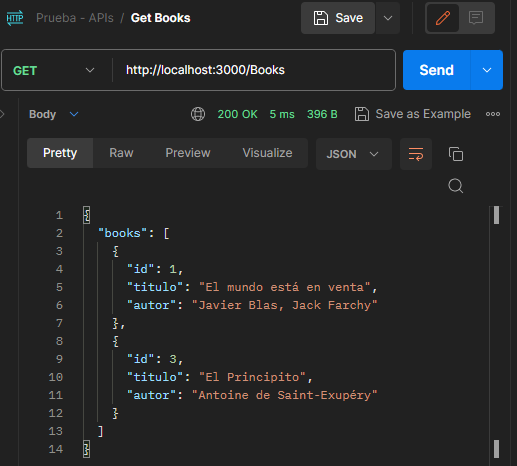


Al ejecutar la prueba en la ruta ‘http://localhost:3000/Books/2’, tenemos que la respuesta es satisfactoria y contiene un mensaje exitoso con status 200.

A continuación, realizamos las pruebas en postman:







Como podemos ver en las pruebas, al borrar un libro retorna un mensaje de éxito. Si no se consigue el libro retorna un mensaje de error. También comprobamos que efectivamente se borró el libro deseado haciendo una petición ‘get’ al servidor.

* Para finalizar, se realiza el commit del proyecto local y se sube a Github.
* **git commit -m "** **prueba terminada"**
* **git push -u origin main**

****