

Introducción a NestJS

Servicio de las Tecnologías de la Información y las Comunicaciones - Universidad de Almería

Tabla de contenidos

Resumen

1. Introducción
2. Creación del proyecto
 - 2.1. Funcionamiento (Servicios y Controladores)
 - 2.2. Definir un prefijo para la API.
3. Creación de nuestro primer servicio y controlador
 - 3.1. El servicio
 - 3.2. El controlador
4. Creación de la primera versión de los endpoints
 - 4.1. Recuperación de un libro
 - 4.2. Filtrado mediante parámetros. Recuperación de todos los libros en orden descendente.
 - 4.3. Creación de un libro
 - 4.4. Eliminación de un libro
 - 4.5. Modificación de un libro
5. Tipado de objetos
 - 5.1. Creación de una interface para libros
 - 5.2. Creación de un DTO para libros
 - 5.3. Modificación del controlador para el uso de tipos
 - 5.4. Modificación del servicio para el uso de tipos
6. Finalización del mockeado
 - 6.1. El servicio
 - 6.2. El controlador
7. Creación de servicios conectados a bases de datos
 - 7.1. Configuración de un servidor MySQL
 - 7.2. ORM y el patrón de repositorio
 - 7.3. Configuración de la conexión a la base de datos
 - 7.4. Creación de entidades
 - 7.5. El servicio
 - 7.6. El controlador
 - 7.7. Módulo para una mejor organización
 - 7.8. Mejora de la configuración del uso del ORM
 - 7.9. Pruebas de los endpoints con persistencia en la base de datos
 - 7.10. Cambio a un servidor PostgreSQL
8. Autenticación con JSON Web Tokens
 - 8.1. Configuración de la estrategia Passport

8.2. Módulo de autenticación

8.3. Restricción del acceso de los endpoints

9. Documentación de la API con Swagger (OpenAPI)

9.1. Documentación de DTOs, entidades, clases e interfaces

9.2. Documentación de los controladores

9.3. Descarga de JSON

9.4. Cambio del frontend

10. Logging

10.1. Configuración de Winston

10.2. Registro de logs con Winston

10.3. Log de operaciones de la API

11. Documentación del código

Apéndice A. Datos de ejemplo

Apéndice B. Generación de código

Generador de archivos para una entidad

Generador del código de la entidad

Resumen

NestJS es un framework para el desarrollo de aplicaciones Node.js en el lado del servidor. Se programa en TypeScript y proporciona una arquitectura en la aplicación que permite el desarrollo de aplicaciones más fáciles de mantener. Su arquitectura está bastante inspirada en Angular lo que facilita el trabajo al equipo de desarrollo al no tener que usar dos formas diferentes de trabajo en el backend y en el frontend.

Objetivos

- Usar el CLI de Nest para la creación de componentes de la aplicación
- Conocer el funcionamiento de los controladores y los servicios
- Crear los métodos básicos CRUD de una aplicación
- Saber cómo capturar los parámetros de las peticiones HTTP
- Conocer las diferencias y utilidades de las *entities* de los ORM (Object Relational Mappers), las interfaces y los DTO (Data Transfer Objects)
- Crear una API sencilla con datos mockeados
- Crear servicios basados en bases de datos
- Usar JWT como mecanismo de control de acceso
- Usar Swagger para la documentación de la API
- Registrar las operaciones de la aplicación en archivos de log
- Usar Compodoc para la documentación de la aplicación



Disponible el repositorio (<https://github.com/ualmtorres/tutorial-nest-js>) usado en este tutorial.

1. Introducción

A la hora de desarrollar un proyecto es importante tener una estructura y una estrategia bien planeada para la organización del código. En situaciones donde además los requerimientos son cambiantes es fácil llegar pronto al desastre. Uno de los motivos por los que surge el framework NestJS (<https://nestjs.com/>) es precisamente el facilitar que los desarrolladores puedan tener una estructura modular de código, lo que facilita el desarrollo de aplicaciones NodeJS empresariales.

NestJS es un framework NodeJS construido sobre NodeJS y TypeScript, y que hace uso de Express (<https://expressjs.com/es/>). Además ofrece soporte para las principales bases de datos (MySQL, PostgreSQL, Oracle, SQLite, MongoDB, ...), Swagger (OpenAPI) (<https://swagger.io/>), autenticación, logging, y una arquitectura inspirada en Angular (<https://angular.io/>), características que lo hacen un framework bastante interesante.

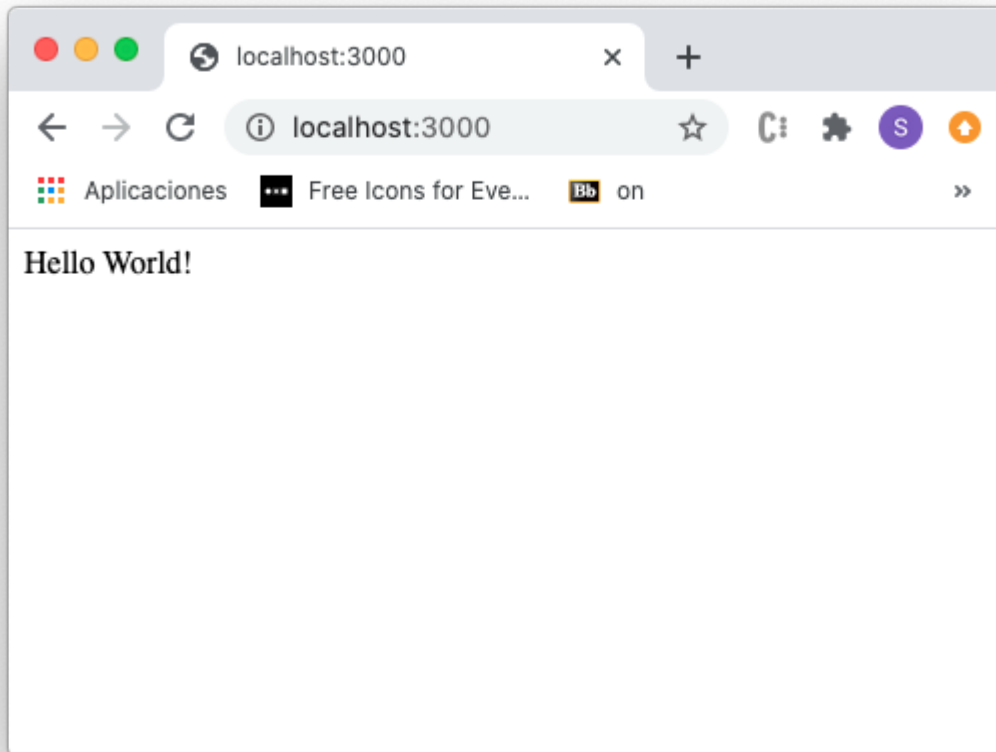
En este tutorial desarrollaremos una API (repositorio (<https://github.com/ualmtorres/tutorial-nest-js>)) sobre bases de datos (MySQL y PostgreSQL) que implementa endpoints para las operaciones básicas (`find`, `findOne`, `create`, `update`, `delete`). Comenzaremos creando un armazón con los controladores y servicios funcionando en modo mock. Una vez probada la conexión correcta entre ellos, se sustituirán los servicios para que interactúen con la base de datos. Además, la API implementará control de acceso a los endpoints mediante JSON Web Tokens (<https://jwt.io/>), quedará documentada con Swagger y registrará sus operaciones en archivos de log.

2. Creación del proyecto

```
$ nest new tutorial-nest-js  
$ cd tutorial-nest-js  
$ npm run start:dev
```

BASH

Esto crea un proyecto y lo ejecuta en el puerto 3000 en modo *live reload*.



Se puede cambiar el puerto en el que se sirve la aplicación modificando el archivo `main.ts`

```
await app.listen(3000); 1
```

TS

1 Cambiar por el puerto deseado

2.1. Funcionamiento (Servicios y Controladores)

Los servicios se encargan de abstraer la complejidad y la lógica del negocio a una clase aparte. El

CLI de NestJS añade el decorador `@Injectable` a los servicios durante su creación. Estos servicios se podrán inyectar en controladores o en otros servicios.

Archivo `app.service.ts`

```
import { Injectable } from '@nestjs/common';TS  
  
@Injectable() 1  
export class AppService {  
  getHello(): string { 2  
    return 'Hello World!';  
  }  
}
```

1 Decorador que permite que el servicio pueda ser inyectado en controladores y en otros servicios

2 Función que proporciona una funcionalidad determinada

El controlador se encarga por un lado de escuchar las peticiones que llegan a la aplicación. Por otro lado, se encarga de preparar las respuestas que proporciona la aplicación. El CLI de NestJS añade el decorador `@Controller` a los controladores durante su creación. NestJS permite el uso de rutas como parámetros del decorador `@Controller`

Archivo `app.controller.ts`

```
import { Controller, Get } from '@nestjs/common';TS  
import { AppService } from './app.service'; 1  
  
@Controller() 2  
export class AppController {  
  constructor(private readonly appService: AppService) {} 3  
  
  @Get() 4  
  getHello(): string { 5  
    return this.appService.getHello(); 6  
  }  
}
```

1 Importación del servicio

2 Decorador que indica a NestJS que es un controlador

3 Inyección del servicio

4 Tipo de petición HTTP y ruta (vacía) atendida por el controlador

- 5 Función a ejecutar al tras invocar la ruta con una petición GET
- 6 Invocación al servicio que resuelve la petición

2.2. Definir un prefijo para la API.

Archivo main.ts

```
import { NestFactory } from '@nestjs/core';  
import { AppModule } from './app.module';  
  
async function bootstrap() {  
  const app = await NestFactory.create(AppModule);  
  app.setGlobalPrefix('api/v1');  
  await app.listen(3000);  
}  
bootstrap();
```

TS

- 1 Prefijo global

La aplicación ahora deberá ser llamada incluyendo el prefijo:

```
http://localhost:3000/api/v1
```

BASH

Si no incluimos el prefijo y seguimos invocando a `http://localhost:3000` obtenendremos el siguiente error. Este error indica que la aplicación no tiene nada que responda en esa ruta a ese tipo de petición HTTP.

```
{  
  "statusCode": 404,  
  "message": "Cannot GET /",  
  "error": "Not Found"  
}
```

JSON

3. Creación de nuestro primer servicio y controlador

Desde la línea de comandos usaremos el CLI de NestJS.

```
$ nest g service books
$ nest g controller books
```

BASH

El servicio creado está disponible en `books/books.service.ts` y el controlador creado está disponible en `books.controller.ts`. Los archivos `.spec.ts` son archivos para pruebas que no trataremos aquí.



El CLI de NestJS ha generado el archivo del servicio `books/books.service.ts` con el decorador `@Injectable` y el archivo del controlador `books.controller.ts` con el decorador `@Controller`

La creación del servicio y del controlador han modificado el archivo `app.module.ts` incorporándolos a la lista de servicios y controladores de la aplicación.

El archivo `app.module.ts`

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { BooksService } from './books/books.service';
import { BooksController } from './books/books.controller';

@Module({
  imports: [],
  controllers: [AppController, BooksController], 1
  providers: [AppService, BooksService], 2
})
export class AppModule {}
```

TS

1 Lista de controladores

2 Lista de providers



Los *providers* son un concepto de un nivel de abstracción mayor al de los servicios. Cuando decíamos que los servicios se encargaban de abstraer la complejidad y la lógica del negocio a una clase aparte, realmente se debía a que esta abstracción es propia de los *providers*. Al ser un servicio un tipo particular de *provider* simplemente heredan su comportamiento.

Un *provider* puede ser un servicio, pero también puede ser un repositorio, una factoría o un *helper*.

3.1. El servicio

Implementamos las funciones que proporcionan los datos.



Es buena práctica comenzar desarrollando todas las funciones que necesitemos ofreciendo inicialmente la funcionalidad de mostrar simplemente que han sido llamadas. Posteriormente, le iremos añadiendo su lógica real de forma progresiva. Esto nos permite tener inicialmente los componentes y las llamadas funcionando e interactuando sin adentrarnos en la complejidad del dominio.

Archivo `books/book.service.ts`

```
import { Injectable } from '@nestjs/common';  
  
@Injectable()  
export class BooksService {  
  findAll(): any {  
    return 'findAll funcionando';  
  }  
}
```

TS

- 1 Ejemplo de función que se limita a indicar que está funcionando cuando es llamada

3.2. El controlador

Comenzamos añadiendo simplemente por ahora:

- El constructor donde se inyecta el servicio para poder usarlo
- Creando la primera ruta y el método HTTP asociado que vamos a probar

TS

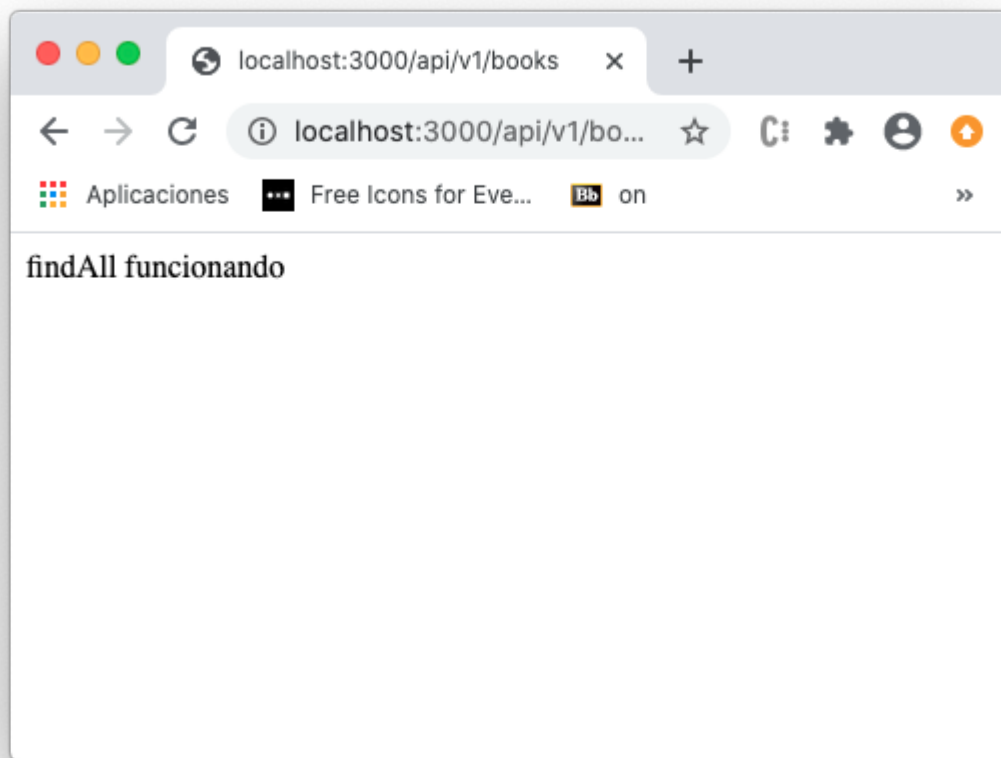
```
import { Controller, Get } from '@nestjs/common';
import { BooksService } from '../books.service'; 1

@Controller('books')
export class BooksController {
  constructor(private booksService: BooksService) {} 2

  @Get() 3
  findAll() { 4
    return this.booksService.findAll(); 5
  }
}
```

- 1 Importación del servicio que proporciona los datos
- 2 Constructor con el servicio inyectado
- 3 Decorador para indicar la ruta atendida y el método HTTP
- 4 Método asociado a la petición
- 5 Llamada al método del servicio que resuelve la petición

Si ahora llamamos a `http://localhost:3000/api/v1/books` el controlador interceptará la petición, usará el servicio y obtendremos la respuesta siguiente.



4. Creación de la primera versión de los endpoints

Comenzaremos haciendo el *armazón (scaffolding)* de los endpoints para todas las rutas permitidas pero en una versión muy preliminar. Los servicios se limitarán a mostrar que han sido llamados y a mostrar los parámetros pasados. Una vez que todos funcionen correctamente podremos sustituirlos por servicios que tengan la respuesta real que exige el problema.

Table 1. Endpoints

Método	Endpoint	Descripción
GET	/api/v1/books	Obtener lista de libros
GET	/api/v1/books/{bookId}	Devuelve información sobre un libro específico
POST	/api/v1/books	Crear un libro
DELETE	/api/v1/books/{bookId}	Eliminar un libro específico
PUT	/api/v1/books/{bookId}	Modificar un libro específico

4.1. Recuperación de un libro

4.1.1. El servicio

Añadimos la función que implementa el servicio de recuperación de un libro específico. Tomará como argumento el `id` del libro e inicialmente se limitará a devolver un mensaje con el propio nombre de la función y el `id` pasado como argumento. Esto permite comprobar que la función ha sido llamada correctamente.

Archivo `books/book.service.ts`

```
...
findBook(bookId: string) {
  return `findBook funcionando con bookId: ${bookId}`;
}
...
```

TS

4.1.2. El controlador

Añadimos la ruta que implementa la petición. Tomará como parámetro el `id` del libro (`bookId`). Usaremos el decorador NestJS `@Param` para obtener el parámetro de la petición.

Archivo `books/book.controller.ts`

```
import { Param } from '@nestjs/common';  
...  
@Controller('books')  
export class BooksController {  
  ...  
  @Get('/:bookId') 1  
  findBook(@Param('bookId') bookId: string) { 2  
    return this.booksService.findBook(bookId); 3  
  }  
  ...  
}
```

TS

- 1 `bookId` es el nombre que se le da al argumento en la petición
- 2 Método asociado a la petición con referencia al argumento de la petición y variable asociada para el método
- 3 Llamada al método del servicio que resuelve la petición

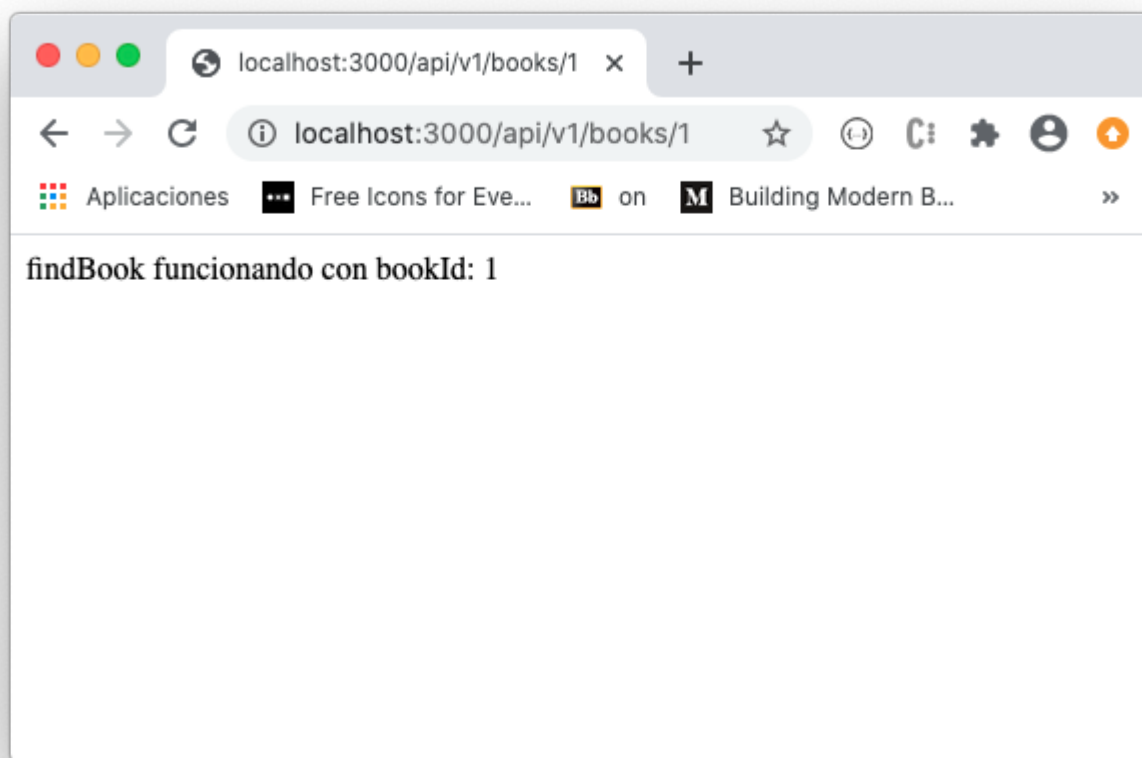
Normalmente se usa el mismo nombre para el parámetro HTTP que para la variable que lo maneja en el método. Sin embargo, son dos objetos diferentes. A continuación se muestra con quien empareja cada uno.



```
@Get('/:RequestedBookId')  
findBook(@Param('RequestedBookId') methodBookId: string) {  
  return this.booksService.findBook(methodBookId);  
}
```

TS

Si ahora llamamos a `http://localhost:3000/api/v1/books/1` el controlador interceptará la petición, asignará `1` al parámetro `bookId` y obtendremos la respuesta siguiente.



4.2. Filtrado mediante parámetros. Recuperación de todos los libros en orden descendente.

En la URL se pueden pasar parámetros en forma de una lista de pares clave valor. Por ejemplo: `http://localhost:3000/api/v1/books?sort=1`. Los parámetros son recogidos en NestJS con el decorador `@Query()`

Nuevo endpoint o sólo parámetros

Puede surgir la duda de si la recuperación de libros de forma ordenada es un nuevo endpoint o se trata de añadir parámetros a un endpoint existente. Es decir, se trata de elegir entre estas dos alternativas:

1. `http://localhost:3000/api/v1/books/sort/1`
2. `http://localhost:3000/api/v1/books?sort=1`

Para resolver la duda nos debemos plantear si la estructura de los datos devueltos cambia de un caso a otro o es la misma en los dos casos. Si cambia estaríamos ante un nuevo

endpoint. En cambio, si es la misma, estaríamos ante parámetros.

En este caso, la ordenación sigue presentando los datos siguiendo la misma estructura. Es decir, sigue siendo una lista de libros igualmente. Lo único es que se presenta ordenada. El servicio tendrá que capturar los parámetros y devolver los datos de acuerdo a la petición realizada.

Esta misma solución es aplicable si hay varios parámetros. Por ejemplo, ordenación, limitación de cantidad de resultados, offsets, filtrado por algún campo, etc. En todos estos casos se sigue devolviendo una lista de resultados con la misma estructura (p.e. libros).



La alternativa de uso de parámetros reduce la cantidad de endpoints a tratar y permite que los parámetros sean opcionales. El servicio tendrá que encargarse de determinar cómo trabajar con los parámetros de la petición.

Como la petición de recuperación de libros de forma ordenada sigue devolviendo una lista de libros con la misma estructura, optamos por implementar esta funcionalidad mediante parámetros, trasladando la lógica de su interpretación al servicio.

4.2.1. El servicio

La versión preliminar del servicio parametrizado modificará el servicio existente de recuperación de libros. La función tomará los argumentos y se limitará a devolver un mensaje con el propio nombre de la función y el argumento (si existe). Esto permite comprobar que la función ha sido llamada correctamente.

Archivo `books/book.service.ts`

```
...
findAll(params): any {
  return params.length > 0
    ? `findAll funcionando con ${params}`
    : 'findAll funcionando';
}
...
```

TS

4.2.2. El controlador

Modificamos la ruta que implementa la petición. Tomará como parámetro el tipo de ordenación. Usaremos el decorador NestJS `@Query` para obtener el parámetro de la petición.

Archivo `books/book.controller.ts`

```
import { Query } from '@nestjs/common';  
...  
@Get()  
findAll(@Query('order') order: string) {  
  1 let params = [];  
    2  
    if (order !== undefined) {  
      3 params.push(`'${order}'`);  
    }  
    4  
    return this.booksService.findAll(params);  
  }  
  ...
```

TS

- 1 Captura del parámetro `order` en una variable `order`
- 2 Array para almacenamiento de parámetros
- 3 Si se ha pasado el parámetro en la petición, se introduce en el array de parámetros
- 4 Llamada al servicio con los parámetros leídos

4.2.3. Una solución más dinámica

La solución planteada para el uso de parámetros hace que ante nuevos parámetros en las peticiones se tenga que modificar tanto el controlador (añadiendo nuevos decoradores `@Query` para los nuevos parámetros) como el servicio, que es el que hace uso de ellos.

El decorador `@Req` nos permite acceder a todos los datos de una petición. En nuestro caso estamos interesados en acceder a `query`. Esta `query` contiene un JSON con los pares parámetro-valor pasados en la petición. La idea es pasar directamente este JSON al servicio y que sea el servicio en que se encargue de acceder a su contenido y actuar como corresponda.

El servicio `books/book.service.ts` adaptado para un nuevo parámetro (`limit`) quedaría así.

TS

```
...
findAll(params): any {
  let msg = `findAll funcionando. Parámetros:`;

  if (params.order !== undefined) {
    msg = msg + ` order: ${params.order}`;
  }

  if (params.limit !== undefined) {
    msg = msg + ` limit: ${params.limit}`;
  }

  return msg;
}
...
```

El controlador `books/book.controller.ts` ahora quedaría así:

```
import { Req } from '@nestjs/common';
import { BooksService } from '../books.service';
import { Request } from 'express';
...

@Controller('books')
export class BooksController {
  constructor(private booksService: BooksService) {}

  @Get()
  findAll(@Req() request: Request) {
    return this.booksService.findAll(request.query);
  }
  ...
}
```

TS

- 1 Inyección del objeto `request`
- 2 Llamada al servicio con el JSON con los pares clave-valor de los parámetros de la petición



Si hiciéramos la petición `http://localhost:3000/api/v1/books?order=1&limit=10`, `request.query` contendría lo siguiente:

```
{ order: '1', limit: '10' }
```

JSON

La pantalla siguiente muestra el resultado de realizar la petición con dos parámetros `order` y `limit`.



4.3. Creación de un libro

Los objetos a crear se pasarán en el `body` de la petición en formato JSON. El cuerpo de la respuesta contendrá el objeto creado.

Supongamos que deseamos insertar el libro siguiente:

```
{
  "title": "El enigma de la habitación 622",
  "genre": "Ficción contemporánea",
  "description": "Vuelve el «principito de la literatura negra contemporánea, el niño mimado de la industria literaria» (GQ): el nuevo thriller de Joël Dicker es su novela más personal. ",
  "author": "Joël Dicker",
  "publisher": "Alfaguara",
  "pages": 624,
  "image_url": "https://images-na.ssl-images-amazon.com/images/I/41KiZbw0hhL._SX315_B01,204,203,200_.jpg"
}
```

JSON

4.3.1. El servicio

La versión preliminar del servicio para crear un nuevo libro se limitará a devolver el libro que le

llega como parámetro. Esto permite comprobar que la función ha sido llamada correctamente.

Archivo `books/book.service.ts`

```
...
createBook(newBook: any) {
  return newBook;
}
... TS
```

4.3.2. El controlador

El decorador `@Body` nos permite acceder al `body` enviado en una petición.

Archivo `books/book.controller.ts`

```
import {
  Post,
  Body,
} from '@nestjs/common';
import { BooksService } from '../books.service';
...

@Controller('books')
export class BooksController {
  constructor(private booksService: BooksService) {}
  ...
  @Post() 1
  createBook(@Body() body) { 2
    let newBook: any = body; 3
    return this.booksService.createBook(newBook); 4
  }
}
```

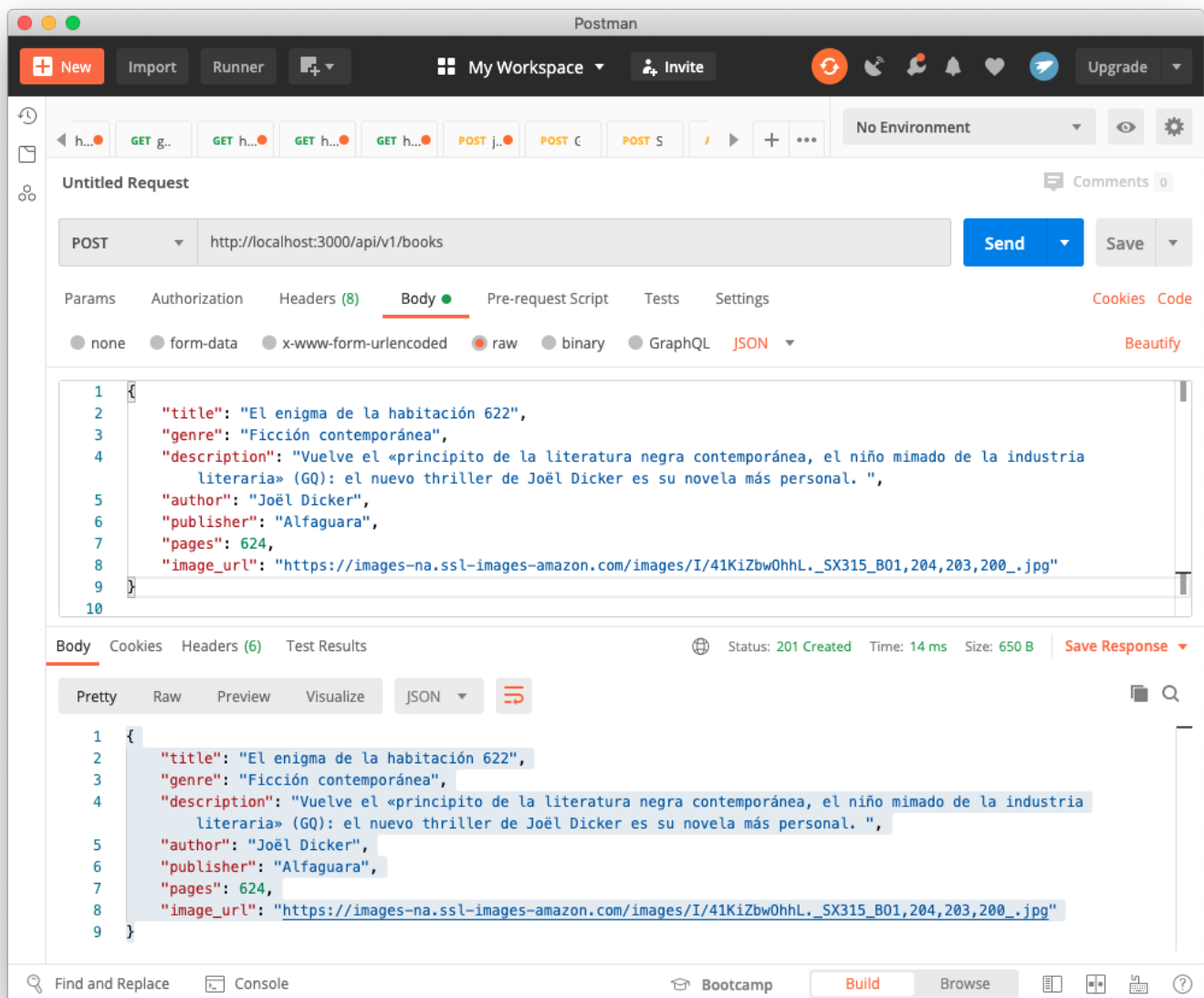
1 Decorador para el método `Post`

2 Decorador para el objeto `body`. Los datos pasados para el nuevo libro se tratan en la variable `body`

3 Creación de un nuevo objeto para poder tratar los datos recibidos

4 Llamada al servicio de creación de libros con el libro recibido

La figura siguiente muestra el resultado de la operación `POST` con el nuevo libro y la respuesta obtenida.



4.4. Eliminación de un libro

La eliminación es muy similar a la de búsqueda de un elemento por `id`. Se intercepta el `id` de la ruta y se llama al servicio.

4.4.1. El servicio

Añadimos la función que implementa el servicio de eliminación de un libro. Se trata de una función muy similar a la de buscar un libro. Tomará como argumento el `id` del libro e inicialmente se limitará a devolver un mensaje con el nombre de la función y el `id` pasado como argumento. Esto permite comprobar que la función ha sido llamada correctamente.

Archivo `books/book.service.ts`

```
...
deleteBook(bookId: string) {
  return `deleteBook funcionando con bookId: ${bookId}`;
}
...
```

4.4.2. El controlador

Añadimos la ruta que implementa la petición. Tomará como parámetro el `id` del libro (`bookId`). Usaremos el decorador NestJS `@Delete`

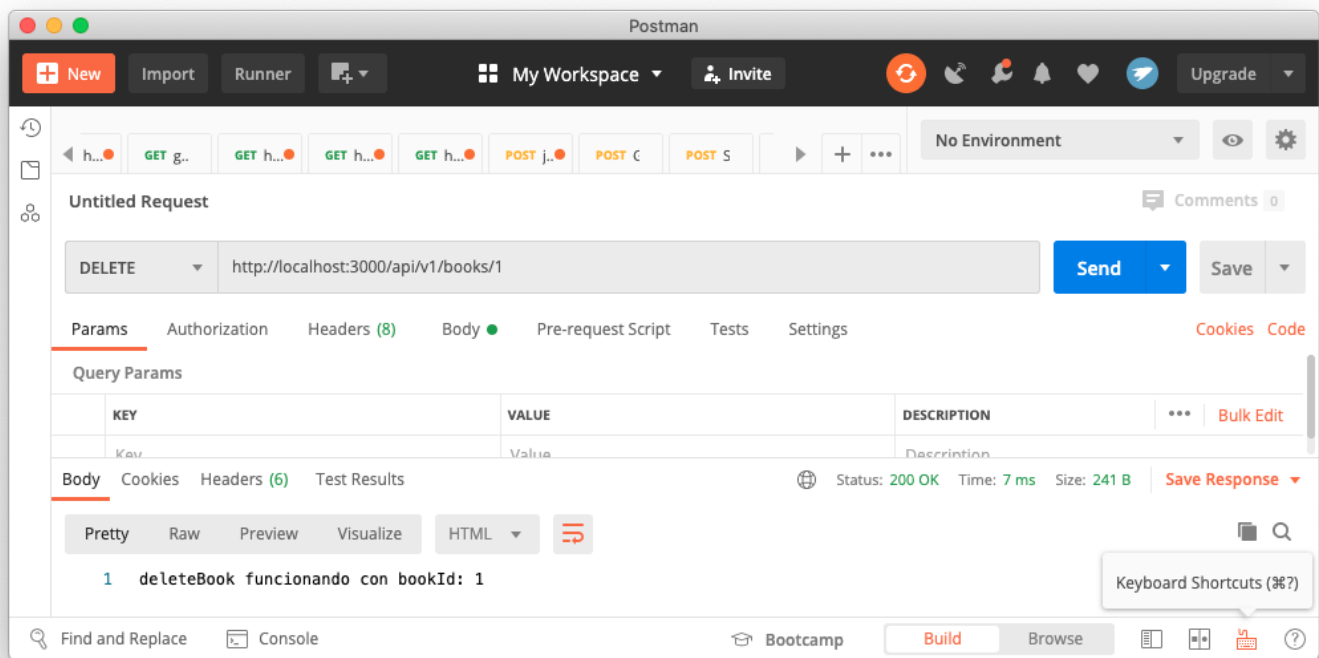
Archivo `books/book.controller.ts`

```
...
@Controller('books')
export class BooksController {
  ...
  @Delete(':bookId') 1
  deleteBook(@Param('bookId') bookId: string) { 2
    return this.booksService.deleteBook(bookId); 3
  }
  ...
}
```

TS

- 1 `bookId` es el nombre que se le da al argumento en la petición
- 2 Método asociado a la petición con referencia al argumento de la petición y variable asociada para el método
- 3 Llamada al método del servicio que resuelve la petición

Si ahora hacemos un `DELETE` contra `http://localhost:3000/api/v1/books/1` el controlador interceptará la petición, asignará `1` al parámetro `bookId` y obtendremos la respuesta siguiente.



4.5. Modificación de un libro

La modificación se puede ver como una operación que combina búsqueda y paso del `body` con los datos a actualizar. Se intercepta el `id` de la ruta el `body` de la petición.

4.5.1. El servicio

Añadimos la función que implementa el servicio de modificación de un libro. Tomará como argumentos el `id` del libro y los nuevos datos del libro. Inicialmente devolverá los datos del libro modificado. Esto permite comprobar que la función ha sido llamada correctamente.

Archivo `books/book.service.ts`

```
...
updateBook(bookId: string, newBook: any) {
  return newBook;
}
...
```

TS

4.5.2. El controlador

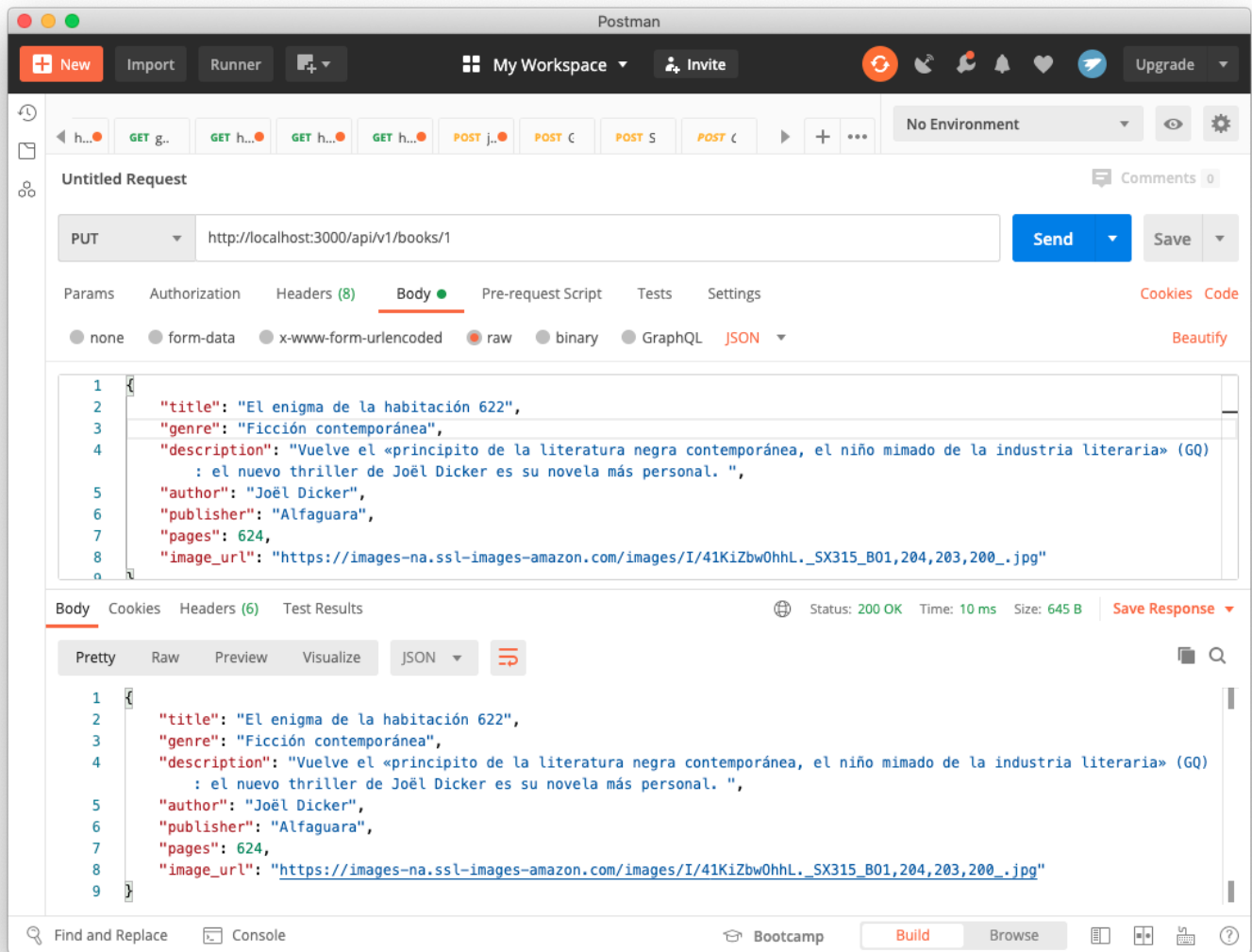
Añadimos la ruta que implementa la petición. Tomará como parámetro el `id` del libro (`bookId`). Usaremos el decorador NestJS `@Put`

Archivo `books/book.controller.ts`

```
...
@Controller('books')
export class BooksController {
  ...
  @Put('/:bookId') 1
  updateBook(@Param('bookId') bookId: string, @Body() body) { 2
    let newBook: any = body;
    return this.booksService.updateBook(bookId, newBook); 3
  }
  ...
}
```

- 1 bookId es el nombre que se le da al argumento en la petición
- 2 Método asociado a la petición con referencia al argumento de la petición, variables asociada para el método y cuerpo con los nuevos datos del libro
- 3 Llamada al método del servicio que resuelve la petición

Si ahora hacemos un UPDATE contra `http://localhost:3000/api/v1/books/1` y le pasamos en el body el JSON con los nuevos datos del libro, el controlador interceptará la petición, asignará 1 al parámetro bookId, pasará el cuerpo, el controlador los pasará al servicio y obtendremos la respuesta siguiente con los nuevos datos del libro.



5. Tipado de objetos

Hasta ahora hemos tratados con el objeto libro, con el `body` de las peticiones que hacen `POST` o `PUT` y en ninguna hemos indicado un tipo de datos. Su tipo queda entonces como `any`. Sin embargo, esto no es una buena práctica. El uso de tipos nos permitirá durante el desarrollo determinar las propiedades aplicables a un objeto, la estructura que tienen que tener los objetos de las peticiones, y demás.

En este tutorial vamos a ver distintos tipos aplicables a los objetos. Para favorecer su comprensión seguimos con el ejemplo de los libros y suponemos que vamos a usar una base de datos para persistir los datos. En este caso tendríamos lo siguiente:

- En la capa de base de datos los libros se podría modelar como una tabla en una base de datos relacional, como una colección en una base de datos de documentos,
- Las **entities**. Si decidimos usar un ORM (https://es.wikipedia.org/wiki/Mapeo_objeto-relacional), ODM (<https://www.quora.com/What-is-Object-Document-Mapping>) o similar, necesitaremos crear un objeto `entity` que represente la estructura de lo que se almacena en la base de datos. En nuestro caso, el objeto `entity` para libro podría tener las mismas propiedades que el objeto de la base de datos. Los objetos `entity` son los que se almacenan y se leen de la base de datos.
- Las **interfaces**. En el nivel de desarrollo necesitamos manipular las propiedades de un objeto para no hacer referencia a propiedades inexistentes, evitar errores de tipado al trabajar con las propiedades de los objetos, y demás. Para ello, necesitaremos tener un *tipo* que represente a los objetos del negocio desde el punto de la programación. Estos tipos no tienen por qué ser sustituidos por los tipos anteriores de los ORM/ODM, ya que nuestra aplicación puede que no use ORM/ODM y no por ello dejarían de ser necesarios los tipos. Los tipos en este nivel los denominamos interfaces.
- Los **DTO (Data Transfer Objects)**. Por último, hemos visto que las peticiones envían sus datos para que sean procesados por los servicios. Sin embargo, los datos enviados en las peticiones no tienen por qué tener la misma estructura que las interfaces o que las *entities* definidas. Por ejemplo, en la petición para crear un libro puede que no se envíe el `id` del libro a crear porque se trata de un valor generado por el sistema. Por tanto, el tipo usado en la petición podría no coincidir con alguno de los tipos anteriores (*entities*, DTO). Estaríamos hablando de un tipo exclusivo para la creación de libros (el tipo que contiene las propiedades que se pasan para crear un libro). Además, operaciones diferentes podrían usar tipos diferentes. Un caso sería que las modificaciones no permitiesen modificar todos los campos de un libro. Estaríamos ante un nuevo tipo, el tipo de los objetos a modificar. A este tipo de objetos se les denomina DTO. (Es habitual usar `CreateBookDTO`, `UpdateBookDTO` para representar los tipos

de los datos pasados al crear y actualizar libros si los tipos son diferentes)

5.1. Creación de una interface para libros

Se define una interface con las propiedades que representan a un libro. En nuestro caso crearíamos un archivo `book.class.ts`

```
export class Book {  
  id: number;  
  title: string;  
  genre: string;  
  description: string;  
  author: string;  
  publisher: string;  
  pages: number;  
  image_url: string;  
}
```

TS



Definimos una clase en un lugar de una interface para poder instanciarla y simplificar el mockeado.

5.2. Creación de un DTO para libros

Se define una clase `BookDto` que representa a las propiedades de un libro que se especifican y se envían cuando se realiza una petición para crear un libro. Hablamos de los datos que van en la petición y no tienen por que tener una correspondencia directa con un objeto completo del dominio. Incluso pueden contener propiedades de varios objetos del dominio. Como su nombre indica, los DTO (Data Transfer Object) representan a la estructura o al tipo de los datos que se están intercambiando.

```
export class BookDto {  
  readonly title: string;  
  readonly genre: string;  
  readonly description: string;  
  readonly author: string;  
  readonly publisher: string;  
  readonly pages: number;  
  readonly image_url: string;  
}
```

TS



El DTO de los libros no contiene el `id` del libro. Esto se debe a que es una propiedad que los usuarios no envían en sus peticiones.

5.3. Modificación del controlador para el uso de tipos

Archivo `books/book.dto.ts`

```
...
import { BookDto } from './book.dto'; 1

@Controller('books')
export class BooksController {
  ...

  @Post()
  createBook(@Body() newBook: BookDto) { 2
    return this.booksService.createBook(newBook); 3
  }

  ....

  @Put('/:bookId')
  updateBook(@Param('bookId') bookId: string, @Body() newBook: BookDto) { 4
    return this.booksService.updateBook(bookId, newBook); 5
  }
}
```

- 1 DTO de libro
- 2 Emparejamiento de lo recibido en el `body` de un `POST` al tipo `BookDto`
- 3 Llamada al servicio de creación de libros con el libro ya tipado
- 4 Emparejamiento de lo recibido en el `body` de un `PUT` al tipo `BookDto`
- 5 Llamada al servicio de actualización de libros con el libro ya tipado



En este ejemplo se observa que se los objetos nuevos y los objetos modificados tienen el mismo tipo. Es decir, cuando se pasa un objeto a modificar, en el `body` se pasa el libro sin `id`.

Este tipado permite manipular de forma segura las propiedades de los libros ayudando a detectarse errores derivados de asignación de valores a tipos incorrectos.

Uno o varios DTO

Un objeto puede tener DTO diferentes para operaciones diferentes. Por ejemplo, si decidiéramos que el DTO de un libro nuevo no contuviese el `id`, pero el DTO de un libro a

modificar sí lo contuviese, tendríamos un caso de DTOs diferentes (p.e. `CreateBook.dto.ts` y `UpdateBook.dto.ts`)

Archivo `CreateBook.dto.ts`

```
export class CreateBookDto {  
  readonly title: string;  
  readonly genre: string;  
  readonly description: string;  
  readonly author: string;  
  readonly publisher: string;  
  readonly pages: number;  
  readonly image_url: string;  
}
```

TS

Archivo `UpdateBook.dto.ts`

```
export class UpdateBookDto {  
  readonly id: number; 1  
  readonly title: string;  
  readonly genre: string;  
  readonly description: string;  
  readonly author: string;  
  readonly publisher: string;  
  readonly pages: number;  
  readonly image_url: string;  
}
```

TS

- 1 DTO de un libro para modificar que sí lleva el `id` del libro modificado

5.4. Modificación del servicio para el uso de tipos

Archivo `books/book.service.ts`

TS

```
...
import { BookDto } from './book.dto'; 1

@Injectable()
export class BooksService {
  ...
  createBook(newBook: BookDto) { 2
    return newBook;
  }

  ...

  updateBook(bookId: string, newBook: BookDto) { 3
    return newBook;
  }
}
```

- 1 DTO de libro
- 2 Libro tipado al DTO
- 3 Libro tipado al DTO

Este tipado permite manipular de forma segura las propiedades de los libros ayudando a detectarse errores derivados de asignación de valores a tipos incorrectos.

6. Finalización del mockeado

Hasta ahora, las únicas operaciones que estaban mockeadas con objetos del dominio eran las operaciones de creación y de modificación. Las operaciones de consulta y eliminación se limitaban a devolver un texto indicando que se había alcanzado el endpoint. En este apartado, haremos que todas las operaciones trabajen con datos del dominio aunque todavía será algo preliminar, ya que serán sólo un par de libros almacenados en el propio código y ninguna operación tratará con datos reales (p.e. la búsqueda de un libro siempre devolverá el mismo libro, la actualización/eliminación siempre informará que se ha modificado/eliminado el mismo libro). No obstante, esto permite que el controlador ya trate con los tipos de datos que devolverán los servicios cuando implementen su funcionalidad real.

6.1. El servicio

El archivo `books/boo.service.ts`

TS

```
import { Injectable, HttpStatus, HttpException } from '@nestjs/common';
import { BookDto } from './book.dto'; 1
import { Book } from './book.class'; 2

@Injectable()
export class BooksService {
  books: Book[] = [ 3
    {
      id: 1,
      title: 'Una historia de España',
      genre: 'Historia',
      description:
        'Un relato ameno, personal, a ratos irónico, pero siempre único, de nuestra
        accidentada historia a través de los siglos. Una obra concebida por el autor para, en
        palabras suyas, «divertirme, releer y disfrutar; un pretexto para mirar atrás desde los
        tiempos remotos hasta el presente, reflexionar un poco sobre ello y contarle por escrito de
        una manera poco ortodoxa.',
      author: 'Arturo Pérez-Reverte',
      publisher: 'Alfaguara',
      pages: 256,
      image_url:
        'https://images-na.ssl-images-amazon.com/images/I/41%2B-
        e981m1L._SX311_B01,204,203,200_.jpg',
    },
    {
      id: 2,
      title: 'Historia de España contada para escépticos',
      genre: 'Historia',
      description:
        'Como escribe el autor, no pretende ser veraz, justa y desapasionada, porque
        ninguna historia lo es. No está hecha para halagar a reyes y gobernantes, ni pretende
        halagar a los banqueros, ni a la Conferencia Episcopal, ni al colectivo gay.',
      author: 'Juan Eslava Galán',
      publisher: 'Booket',
      pages: 592,
      image_url:
        'https://images-na.ssl-images-amazon.com/images
        /I/51IyZ5Mq8YL._SX326_B01,204,203,200_.jpg',
    },
  ];
  findAll(params): Book[] { 4
    return this.books;
  }

  findBook(bookId: string): Book { 5
    return this.books[parseInt(bookId) - 1];
  }

  createBook(newBook: BookDto): Book { 6
    let book = new Book();
```



```
    book.id = 99;
    book.author = newBook.author;
    book.description = newBook.description;
    book.genre = newBook.genre;
    book.image_url = newBook.image_url;
    book.pages = newBook.pages;
    book.publisher = newBook.publisher;
    book.title = newBook.title;

    return book;
}

deleteBook(bookId: string): Book { 7
    return this.books[parseInt(bookId) - 1];
}

updateBook(bookId: string, newBook: BookDto): Book { 8
    return this.books[parseInt(bookId) - 1];
}
}
```

1 DTO del libro (no contiene el `id`)

2 Interface del libro (contiene el `id`)

3 Lista de libros de ejemplo mientras se desarrolla el acceso a BD del servicio

4 El método devuelve un array de `Book` con todos los libros

5 El método devuelve un `Book`, que contiene el `id`. Devuelve un libro a modo de ejemplo

6 El método toma un `BookDto` como argumento (libro sin `id`) y devuelve un libro completo (con el `id`). Devuelve el libro insertado

7 El método devuelve un `Book`, que contiene el `id`. Devuelve un libro a eliminado modo de ejemplo

8 El método toma un `BookDto` como argumento (libro sin `id`) y devuelve un `Book`, que sí contiene el `id`. Devuelve un libro modificado a modo de ejemplo

6.2. El controlador

Se trata de usar los tipos que usan los parámetros de las funciones en las peticiones y de los tipos que devuelven.

Archivo `books/books.controller.ts`

```
import {
  Controller,
  Get,
  Param,
  Req,
  Post,
  Body,
  Delete,
  Put,
} from '@nestjs/common';
import { BooksService } from '../books.service';
import { Request } from 'express';
import { BookDto } from '../book.dto';
import { Book } from '../book.class';

export class BooksController {
  constructor(private booksService: BooksService) {}

  findAll(@Req() request: Request): Book[] {
    console.log(request.query);
    return this.booksService.findAll(request.query);
  }

  findBook(@Param('bookId') bookId: string): Book {
    return this.booksService.findBook(bookId);
  }

  createBook(@Body() newBook: BookDto): Book {
    return this.booksService.createBook(newBook);
  }

  deleteBook(@Param('bookId') bookId: string): Book {
    return this.booksService.deleteBook(bookId);
  }

  updateBook(@Param('bookId') bookId: string, @Body() newBook: BookDto): Book {
    return this.booksService.updateBook(bookId, newBook);
  }
}
```

7. Creación de servicios conectados a bases de datos

Hasta ahora, los servicios que hemos creado en este tutorial se limitan a proporcionar unos datos de prueba generando una salida por la consola. Su cometido se ha estado limitando a comprobar que son alcanzables desde los endpoints definidos en la API, mostrándonos simplemente el eco de su llamada. En este apartado vamos a ver cómo conectar el servicio a bases de datos. Primero lo haremos conectando los servicios a una base de datos MySQL y luego comprobaremos lo fácil que es pasarlo a una base de datos PostgreSQL.

7.1. Configuración de un servidor MySQL

Para trabajar localmente con persistencia necesitamos una base de datos a la que conectarnos. Para no tener que complicarnos con instalaciones y no acoplar el desarrollo a nuestro equipo utilizaremos una imagen Docker de MySQL 5.7 (https://hub.docker.com/_/mysql). Crearemos una base de datos denominada `tutorial`. Usaremos la cuenta `root` con el password `secret`

```
$ docker run --name tutorial_mysql -e MYSQL_ROOT_PASSWORD=secret -p 3306:3306 -d mysql:5.7BASH
```

1

1 Usaremos el password `secret` para la cuenta `root`

Tras unos instantes (algo más si la imagen de MySQL 5.7 no está descargada en el equipo) habrá un contenedor en ejecución con el nombre `tutorial_mysql`. Iniciaremos una sesión interactiva para crear una base de datos, a la que denominaremos `tutorial`

```
$ docker exec -it tutorial_mysql bashBASH
root@d0512407a21d:/# mysql -u root -p
Enter password: 1
...
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
mysql> create database tutorial; 2
Query OK, 1 row affected (0.00 sec)
```

- 1 Introducir el password `secret`
- 2 Crear la base de datos `tutorial`

7.2. ORM y el patrón de repositorio

Un ORM nos abstrae del acceso a un gestor de bases de datos específico. Esto nos aísla del gestor de base de datos elegido y hace que podamos cambiar de gestor de bases de datos de forma muy sencilla. TypeORM (<https://typeorm.io/#/>) es un ORM para TypeScript y JavaScript que facilita la interacción con la base de datos. El uso de TypeORM acelera el proceso de desarrollo modelando entidades en el código y sincronizando estos modelos con la base de datos. Actualmente TypeORM ofrece soporte para varias bases de datos relacionales, como PostgreSQL, Oracle, Microsoft SQL Server, SQLite, e incluso para bases de datos NoSQL, como MongoDB.

Por otro lado, el patrón de repositorio

(<http://blog.sapiensworks.com/post/2012/02/22/The-Repository-Pattern-Explained.aspx>) nos abstrae de los detalles de la persistencia proporcionando métodos abstractos para las operaciones comunes (crear, guardar, buscar, buscar una, actualizar, eliminar, ...).

Resumiendo, el ORM trabaja con objetos de la base de datos y el repositorio trabaja con objetos del dominio.

Instalaremos los paquetes de TypeORM en el proyecto con

```
$ npm install --save @nestjs/typeorm typeorm mysql
```

BASH

7.3. Configuración de la conexión a la base de datos

Haremos la configuración de la base de datos en el archivo `app.module.ts` mediante `TypeOrmModule.forRoot()`. Se le pueden pasar los parámetros de configuración directamente. Sin embargo, existe otra opción que consiste en definir la configuración en un archivo `ormconfig.json`, que es el que de forma predeterminada busca TypeORM.

```
import { TypeOrmModule } from '@nestjs/typeorm';
...
@Module({
  imports: [
    TypeOrmModule.forRoot(),
    ...
  ],
  ...
})
export class AppModule {}
```

TS

- 1 De forma predeterminada, si no se pasa ningún argumento se buscan los valores en `ormconfig.json` en la raíz del proyecto.

A continuación se muestra el archivo `ormconfig.json`. Este archivo se almacena en la raíz del proyecto, junto al `package.json`.

Archivo `ormconfig.json`

```
{  
  "type": "mysql",  
  "host": "localhost",  
  "port": 3306,  
  "username": "root",  
  "password": "secret",  
  "database": "tutorial",  
  "entities": ["dist/**/*.entity.js"], 1  
  "synchronize": true 2  
}
```

JSON

- 1 Dónde localizar los archivos de las entidades
- 2 Sincronización automática de la base de datos con las entidades

Configuración de los datos de conexión en el propio código

También se puede encontrar que los parámetros de conexión son colocados directamente como argumentos de `TypeOrmModule.forRoot()`.

```
...  
TypeOrmModule.forRoot(  
  {  
    type: 'mysql',  
    host: 'localhost',  
    port: 3306,  
    username: 'root',  
    password: 'example',  
    database: 'my_nestjs_project',  
    entities: ['dist/**/*.entity.js'],  
    synchronize: true,  
  }  
)  
...
```

CODE

El problema de este enfoque está en que las credenciales se adjuntarán en los commits que se hagan de este archivo. En cambio, si almacenamos las credenciales en un archivo `ormconfig.json` y lo incluimos en el archivo `.gitignore`, los datos sensibles almacenados en `ormconfig.json` no serán expuestos al hacer commit.

7.4. Creación de entidades

Las entidades son clases que se corresponden con tablas de la base de datos (colecciones si se trata de MongoDB). En las entidades se definen las columnas y relaciones. Una de esas columnas debe ser la clave primaria.

A continuación, para nuestro ejemplo de libros se muestra la definición de una entidad `Book` con las columnas siguientes:

- `id`
- `title`
- `genre`
- `description`
- `author`
- `publisher`
- `pages`
- `image_url`

Archivo `books/book.entity.ts`

TS

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Book {
  @PrimaryGeneratedColumn() 1
  id: number;

  @Column()
  title: string;

  @Column()
  genre: string;

  @Column('text') 2
  description: string;

  @Column()
  author: string;

  @Column()
  publisher: string;

  @Column()
  pages: number;

  @Column()
  image_url: string;
}
```

- 1 Decorador para indicar que es una clave primaria autonumérica
- 2 Decorador para permitir texto largo

7.5. El servicio

El servicio implementa las funciones habituales para operaciones CRUD (find, findOne, create, delete y update). Se usa el patrón repositorio para trabajar directamente sobre objetos del dominio (libros en nuestro caso) y olvidarnos de los detalles de la persistencia. Como todas las funciones interactúan con bases de datos, todas se programan de forma asíncrona y devuelven una promesa, por lo que habrá que llamarlas con `await`.

Promesas, `async` y `await`

Cuando trabajamos con bases de datos las respuestas no son inmediatas. En JavaScript las promesas (https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Promise) representan valores que pueden estar disponibles ahora, en el futuro o nunca. Para facilitar

el trabajo con la programación asíncrona surge la pareja `async/await`. Con esta pareja:

- Las funciones son definidas con `async` para indicar que devuelven una promesa.
- Con `await` indicamos a JavaScript que espere hasta que la promesa se cumpla y devuelva su resultado.



`await` sólo funciona en funciones `async`. Se coloca en funciones `async` basadas en promesas para detener la ejecución hasta que se cumpla la promesa.

Archivo `books/books.service.ts`

TS


```
import { Injectable, HttpStatus, HttpException } from '@nestjs/common';
import { BookDto } from './book.dto'; 1
import { Book } from './book.entity'; 2
import { InjectRepository } from '@nestjs/typeorm'; 3
import { Repository } from 'typeorm'; 4

@Injectable()
export class BooksService {

  constructor(
    @InjectRepository(Book) private booksRepository: Repository<Book>, 5
  ) {}

  async findAll(params): Promise<Book[]> { 6
    return await this.booksRepository.find(); 7
  }

  async findBook(bookId: string): Promise<Book> {
    return await this.booksRepository.findOne({ where: { id: bookId } }); 8
  }

  createBook(newBook: BookDto): Promise<Book> {
    return this.booksRepository.save(newBook);
  }

  async deleteBook(bookId: string): Promise<any> {
    return await this.booksRepository.delete({ id: parseInt(bookId) });
  }

  async updateBook(bookId: string, newBook: BookDto): Promise<Book> { 9
    let toUpdate = await this.booksRepository.findOne(bookId); 10

    let updated = Object.assign(toUpdate, newBook); 11

    return this.booksRepository.save(updated); 12
  }
}
```

- 1 Estructura de un libro para insertar (tiene todo menos el `id`, que se genera en la base de datos)
- 2 Estructura completa de un libro (incluye el `id`)
- 3 Decorador para inyectar repositorios
- 4 Repositorio de TypeORM
- 5 Uso del decorador `@InjectRepository` en el constructor para inyectar el `Repository` que manejará a la entidad `Book`

- 6 Las funciones del servicio se basan en funciones asíncronas del repositorio, que devuelven promesas y tendrán que ser llamadas con `await`. Por tanto, las funciones del servicio son `async` y devuelven promesas personalizadas al tipo con el que trabajan (libros, arrays de libros, ...)
- 7 La llamada a los métodos del repositorio devuelven promesas, por lo que llamaremos con `await` para esperar a que se resuelvan
- 8 Los parámetros en TypeORM se suelen pasar en JSON
- 9 La actualización se implementa como la recuperación del libro a modificar, la sustitución de todos sus valores excepto el `id` por los del libro pasado como parámetro y su posterior almacenamiento en la base de datos
- 10 Recuperación del libro a modificar
- 11 Asignación de todas las propiedades del libro *nuevo* al libro *antiguo*, excepto el `id`, que no está incluida en el libro *nuevo*
- 12 Almacenamiento del libro en la base de datos tras su modificación

7.6. El controlador

Básicamente, el controlador es el mismo que teníamos para el mockup salvo que ahora devuelve promesas, ya que las funciones del servicio ahora devuelven promesas. Además, se cambia el tipo del objeto libro. Dejamos de usar la `interface` para pasar a usar la `entity` del ORM.

Archivo `books/books.controller.ts`

TS

```
import {
  Controller,
  Get,
  Param,
  Req,
  Post,
  Body,
  Delete,
  Put,
} from '@nestjs/common';
import { BooksService } from '../books.service';
import { Request } from 'express';
import { BookDto } from '../book.dto';
import { Book } from '../book.entity'; 1

@Controller('books')
export class BooksController {

  constructor(private booksService: BooksService) {}

  @Get()
  findAll(@Req() request: Request): Promise<Book[]> { 2
    console.log(request.query);
    return this.booksService.findAll(request.query);
  }

  @Get('/:bookId')
  findBook(@Param('bookId') bookId: string): Promise<Book> {
    return this.booksService.findBook(bookId);
  }

  @Post()
  createBook(@Body() newBook: BookDto): Promise<Book> { 3
    return this.booksService.createBook(newBook);
  }

  @Delete('/:bookId')
  deleteBook(@Param('bookId') bookId: string): Promise<Book> {
    return this.booksService.deleteBook(bookId);
  }

  @Put('/:bookId')
  updateBook(
    @Param('bookId') bookId: string,
    @Body() newBook: BookDto, 4
  ): Promise<Book> {
    return this.booksService.updateBook(bookId, newBook);
  }
}
```

- 1 El tipo de la interfaz y el de la entidad coinciden. Nos quedamos con el de la entidad.
- 2 Las funciones ahora devuelven promesas basadas en la `entity`
- 3 Cambiamos el tipo `any` del `body` por el tipo del DTO del libro a crear
- 4 Cambiamos el tipo `any` del `body` por el tipo del DTO del libro actualizado

7.7. Módulo para una mejor organización

Es buena práctica que en lugar de añadir cada uno de los *providers* y los *controllers* a `app.module.ts`, los agrupemos cada uno en un módulo con los *providers* y *controllers*. Posteriormente, ese módulo se importa en el array `imports` de `app.module.ts`. Además, las entidades se colocan en el módulo en un array, como argumento de `TypeOrmModule.forFeature()`.

Archivo `books/books.module.ts`

```
import { Module } from '@nestjs/common';
import { Book } from '../book.entity';
import { BooksService } from '../books.service';
import { BooksController } from '../books.controller';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [TypeOrmModule.forFeature([Book])], 1
  providers: [BooksService], 2
  controllers: [BooksController], 3
})
export class BooksModule {}
```

TS

- 1 Las entidades van aquí
- 2 El servicio
- 3 El controlador

Este archivo ya está preparado para ser colocado en el array `imports` de `app.module.ts`.

7.8. Mejora de la configuración del uso del ORM

Otra mejora que podríamos realizar para la configuración del uso del ORM podría ser el uso de variables de entorno. Esto evita la introducción de valores sensibles en el código, como contraseñas, usuarios de la base de datos, y demás.

La mejora que haremos se basará en lo siguiente:

1. Inicialización de un archivo de variables de entorno.
2. Creación de un servicio de configuración del ORM a partir de los valores de las variables de entorno.
3. Modificación del archivo `app.module.ts` para usar la configuración anterior y cargar los módulos correspondientes (p.e. el de `BooksModule` creado antes).

7.8.1. Inicialización de un archivo de variables de entorno

Archivo `.env`

```
TUTORIAL_HOST=localhost
TUTORIAL_PORT=3306
TUTORIAL_USER=root
TUTORIAL_PASSWORD=secret
TUTORIAL_DATABASE=tutorial
```

ENV

7.8.2. Creación de un servicio de configuración del ORM

Definiremos un servicio de configuración que acceda a las variables de entorno, especifique las variables de entorno que hay que configurar y una función que las configure.



Se trata de un código precocinado que utilizaríamos en cada proyecto con TypeORM. Sólo hay que cambiar el tipo de gestor de base de datos que se va a usar (`mysql`, `postgres`, ...). Actualmente, tiene que estar en el código y no se puede pasar en una variable.

Archivo `config/config.service.ts`

TS

```
import { TypeOrmModuleOptions } from '@nestjs/typeorm'; 1

require('dotenv').config();

class ConfigService {
  constructor(private env: { [k: string]: string | undefined }) {}

  private getValue(key: string, throwOnMissing = true): string {
    const value = this.env[key];
    if (!value && throwOnMissing) {
      throw new Error(`config error - missing env.${key}`);
    }

    return value;
  }

  public ensureValues(keys: string[]) {
    keys.forEach(k => this.getValue(k, true));
    return this;
  }

  public getTypeOrmConfig(): TypeOrmModuleOptions { 2
    return {
      type: 'mysql', 3

      host: this.getValue('TUTORIAL_HOST'), 4
      port: parseInt(this.getValue('TUTORIAL_PORT')),
      username: this.getValue('TUTORIAL_USER'),
      password: this.getValue('TUTORIAL_PASSWORD'),
      database: this.getValue('TUTORIAL_DATABASE'),

      entities: ['dist/**/*.entity.js'], 5
      synchronize: true, 6
    };
  }
}

const configService = new ConfigService(process.env).ensureValues([
  'TUTORIAL_HOST',
  'TUTORIAL_PORT',
  'TUTORIAL_USER',
  'TUTORIAL_PASSWORD',
  'TUTORIAL_DATABASE',
]);

export { configService };
```

- 1 Importación del módulo de configuración de TypeORM
- 2 Función que configura las opciones de TypeORM

- 3 Configuración del gestor de base de datos a usar
- 4 Configuración de valores mediante variables de entorno
- 5 Especificación del directorio de entidades
- 6 Actualización de las tablas ante cambios en las entidades

7.8.3. Actualización de `app.module.ts` para cargar la configuración del ORM y los módulos

Por último, modificamos el archivo `app.module.ts` para usar la configuración anterior y cargar el módulo `BooksModule`, que define su *provider*, controlador y la entidad contra la que se mapea.

Archivo `app.module.ts`

```
import { Module } from '@nestjs/common';TS
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { BooksModule } from './books/books.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { configService } from './config/config/config.service';

@Module({
  imports: [
    BooksModule,
    TypeOrmModule.forRoot(
      configService.getTypeOrmConfig(),
    ),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

- 1 Importación del módulo
- 2 Configuración de los valores de TypeORM

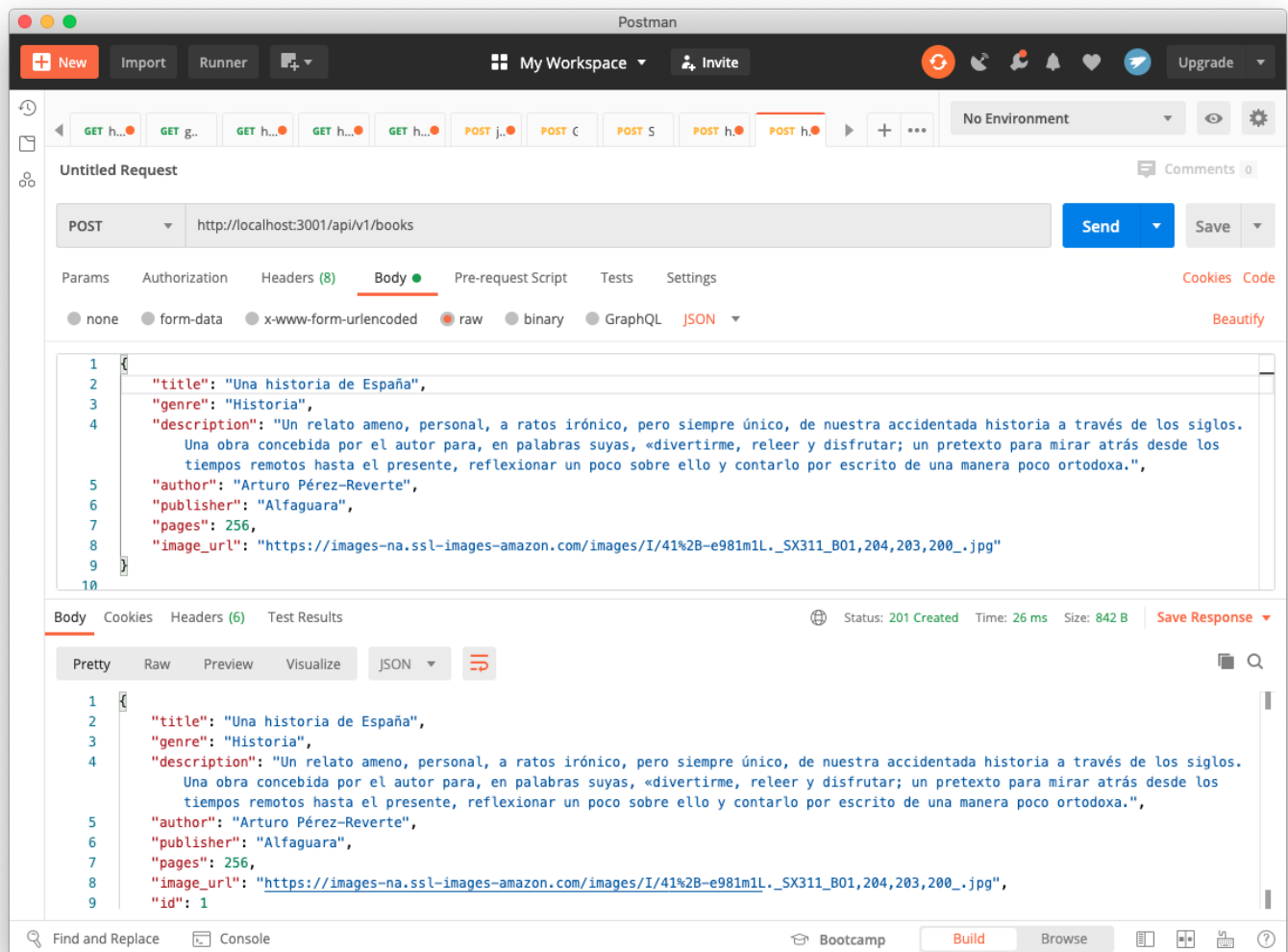
7.9. Pruebas de los endpoints con persistencia en la base de datos

En el Apéndice A. Datos de ejemplo podemos encontrar datos para insertar en la base de datos. Se podrían como `body` en un método `POST` para su creación o `PUT` para su modificación.

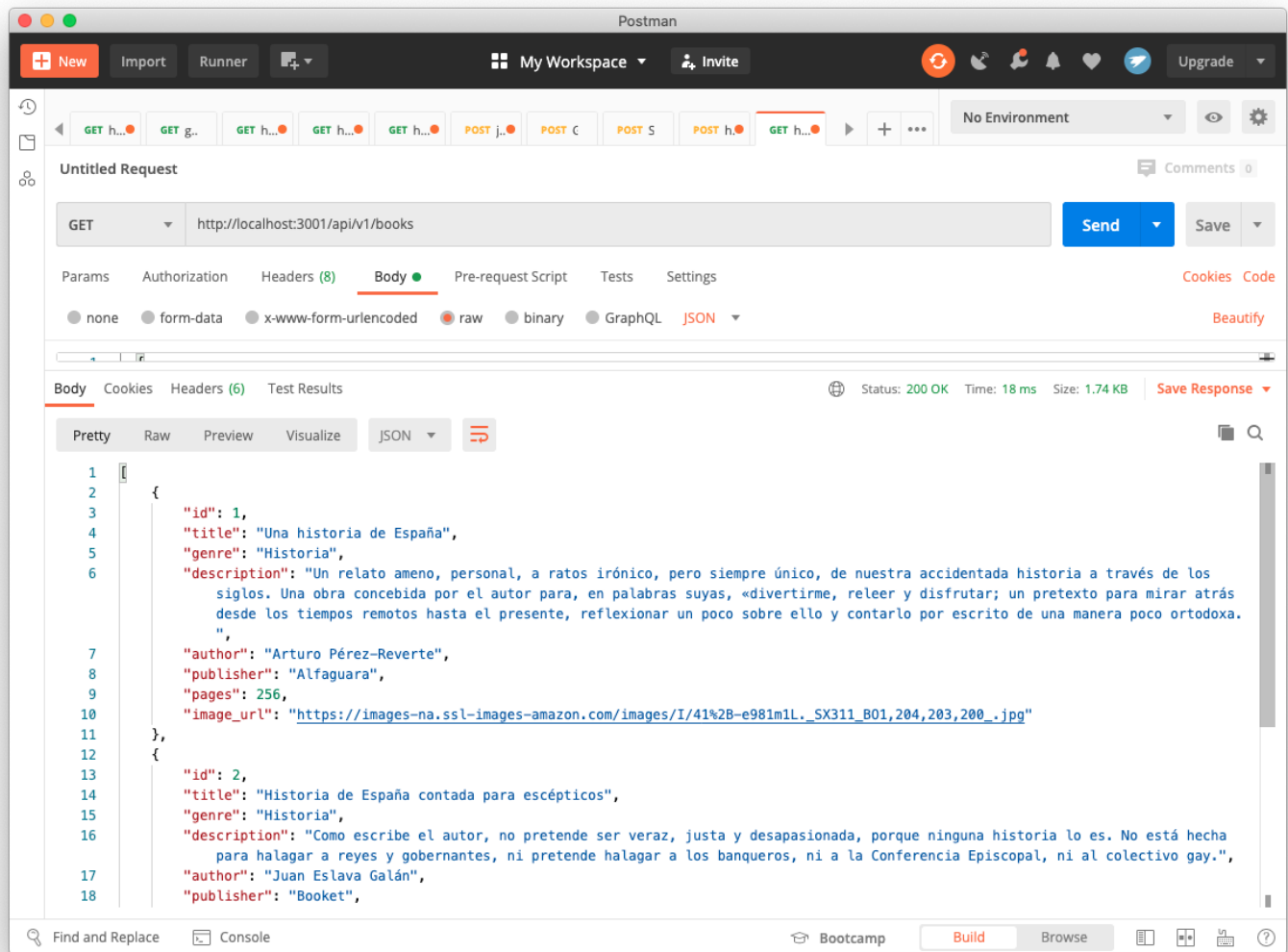
Usaremos Postman para mostrar los resultados de utilizar los distintos endpoints implementados.

La figura siguiente muestra la creación de un libro. El libro nuevo se pasa en el `body`. Se

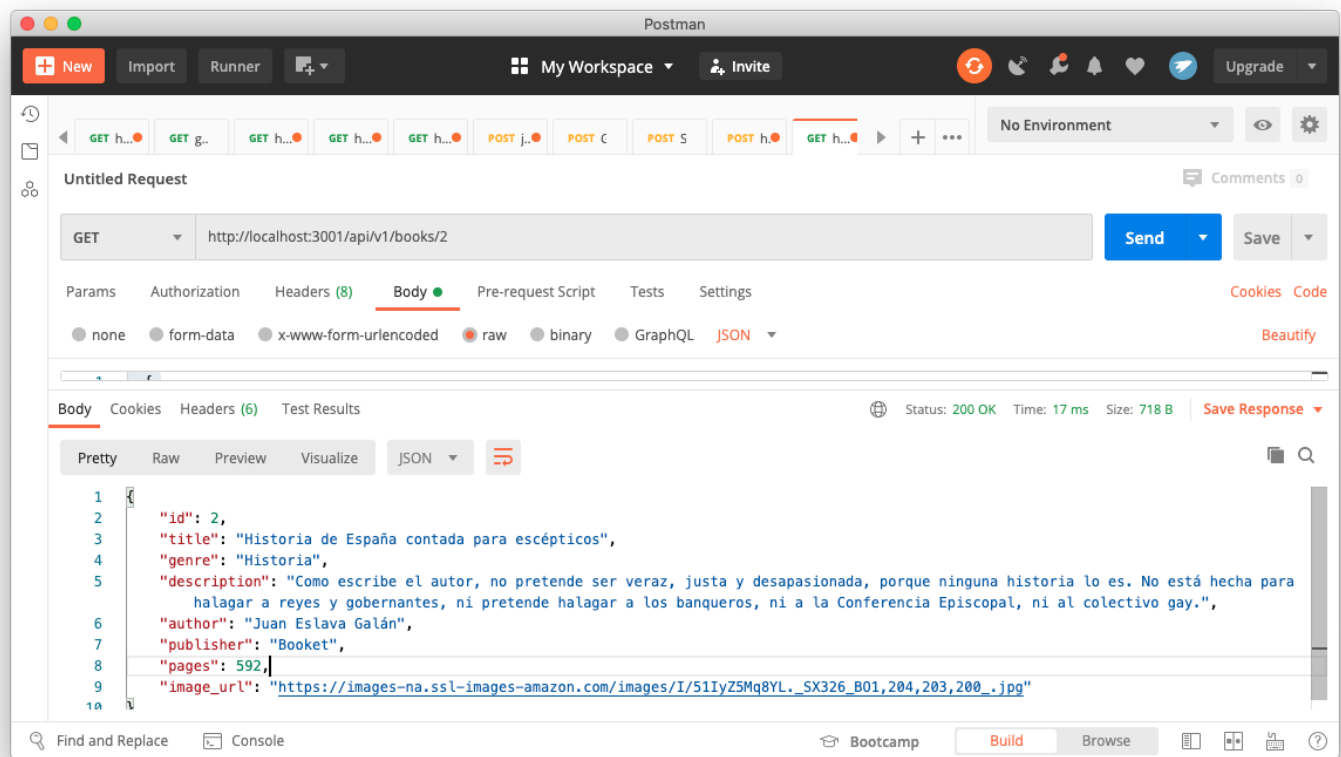
devuelve el libro insertado, junto al `id` generado en la base de datos. El endpoint usado es `/api/v1/books` con el método `POST`.



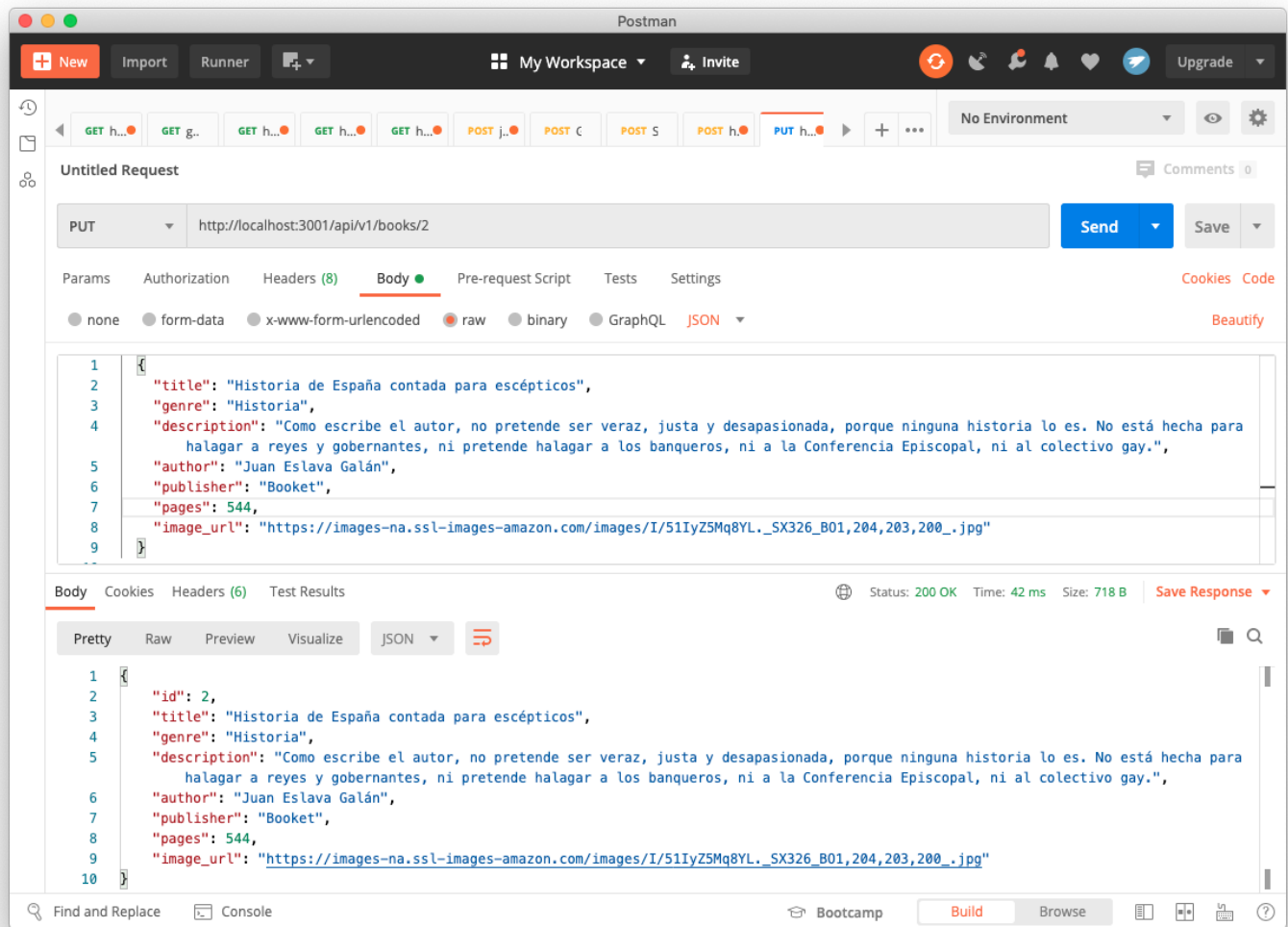
Tras insertar todos los libros del Apéndice A. Datos de ejemplo, la figura siguiente muestra el listado de todos libros. El endpoint usado es `/api/v1/books` con el método `GET`.



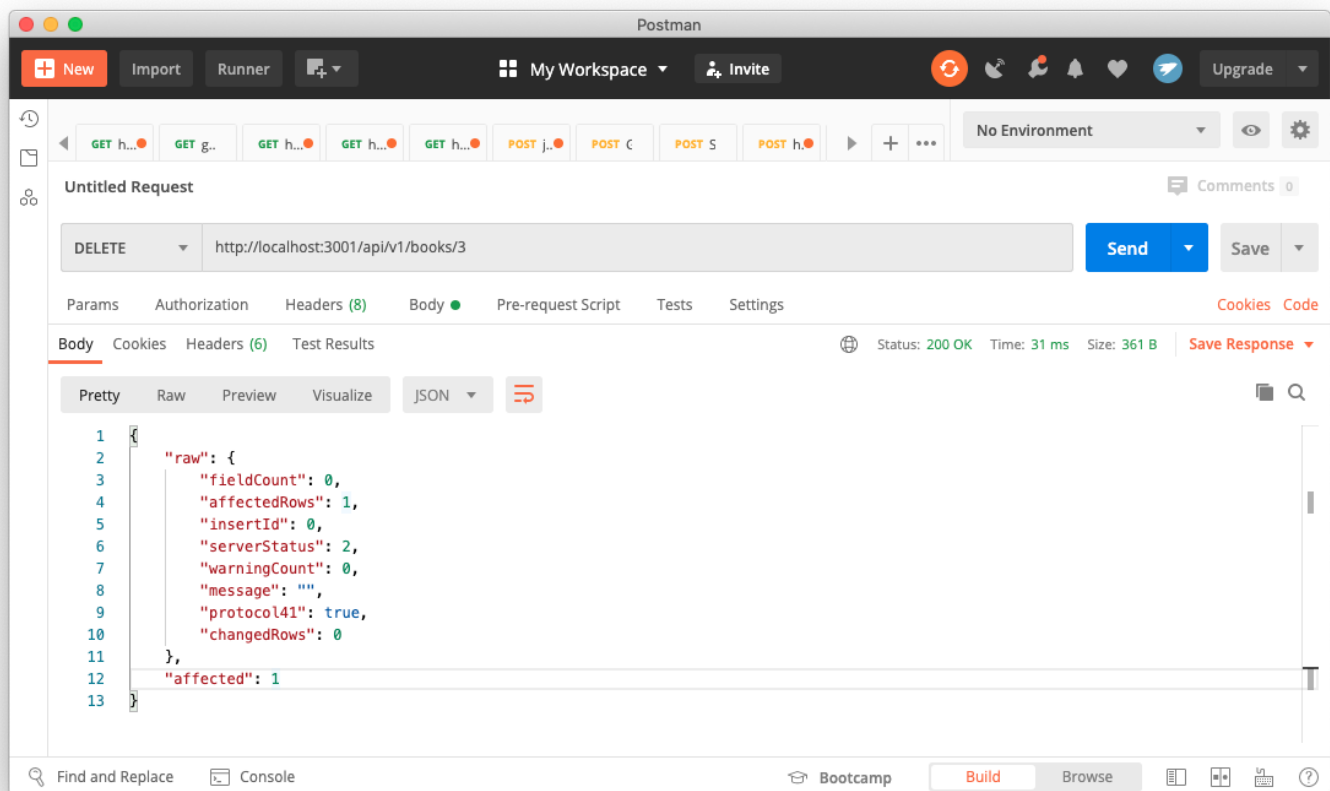
La figura siguiente muestra los detalles de un libro concreto (el 2). El endpoint usado es `/api/v1/books/2` con el método `GET`.



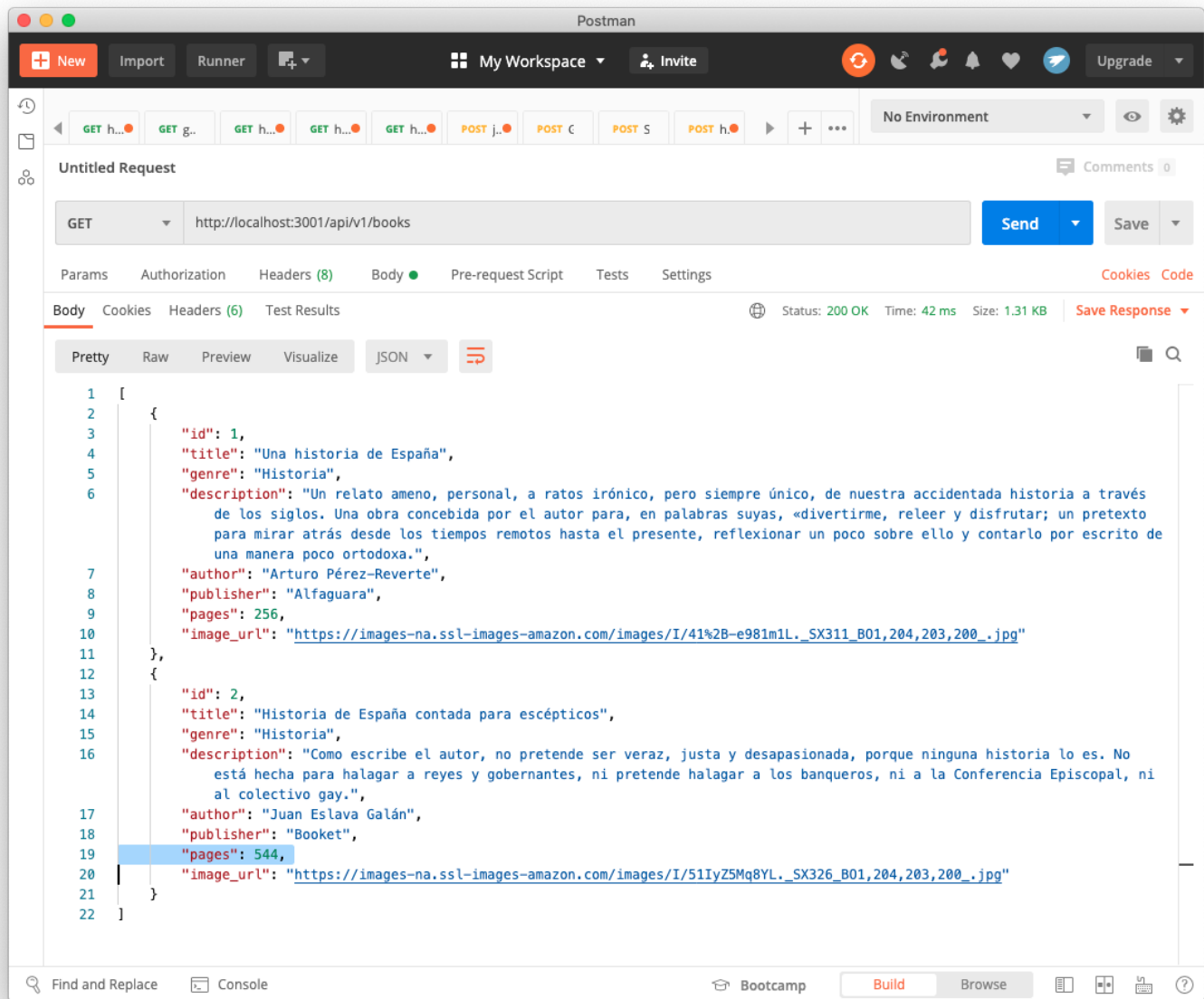
La figura siguiente muestra la modificación de un libro. El `id` del libro a modificar se pasa como parámetro en la ruta y los datos del libro con sus modificaciones se pasan en el `body`. Se devuelve el libro modificado. El ejemplo muestra el cambio del número de páginas del libro 2 al valor 544. El endpoint usado es `/api/v1/books/2` con el método `PUT`.



La figura siguiente muestra la eliminación de un libro. El `id` del libro a eliminar se pasa como parámetro en la ruta. Se devuelve un JSON con los libros eliminados (`affected`). Por ejemplo, para eliminar el libro con `id` 3 usaríamos el endpoint `/api/v1/books/3` con el método `DELETE`.



Si ahora volvemos a consultar todos los libros se verán los cambios en el número de páginas del libro 2 y que el libro 3 ha sido eliminado.



7.10. Cambio a un servidor PostgreSQL

El cambio a un nuevo servidor de bases de datos es bastante sencillo. Se tendrían que seguir estos pasos:

1. Instalación de los paquetes del nuevo gestor de bases de datos
2. Cambiar las variables de entorno con los nuevos valores de conexión a la base de datos
3. Cambio del tipo de base de datos en TypeORM

7.10.1. Instalación de los paquetes de PostgreSQL

```
npm install --save pg
```

BASH

Creación de un contenedor con PostgreSQL

Para facilitar la configuración de la base de datos, el script siguiente lanza un contenedor PostgreSQL y crea una base de datos `tutorial` con el password `secret` (los mismos datos que se usaron para el ejemplo con MySQL)

Archivo `start-postgres.sh`

SH

```
#!/bin/bash
set -e

SERVER="tutorial_postgres";
PW="secret";
DB="tutorial";

echo "echo stop & remove old docker [$SERVER] and starting new fresh instance of
[$SERVER]"
(docker kill $SERVER || :) && \
  (docker rm $SERVER || :) && \
  docker run --name $SERVER -e POSTGRES_PASSWORD=$PW \
  -e PGPASSWORD=$PW \
  -p 5432:5432 \
  -d postgres

# wait for pg to start
echo "sleep wait for pg-server [$SERVER] to start";
SLEEP 3;

# create the db
echo "CREATE DATABASE $DB ENCODING 'UTF-8';" | docker exec -i $SERVER psql -U
postgres
echo "\1" | docker exec -i $SERVER psql -U postgres
```

7.10.2. Modificación de las variables de entorno

Cambios a realizar: en el archivo `.env` :

BASH

```
TUTORIAL_HOST=localhost
TUTORIAL_PORT=5432 1
TUTORIAL_USER=postgres 2
TUTORIAL_PASSWORD=secret
TUTORIAL_DATABASE=tutorial
```

- 1 Puerto de PostgreSQL
- 2 Usuario de PostgreSQL

7.10.3. Modificación del tipo de gestor de bases de datos

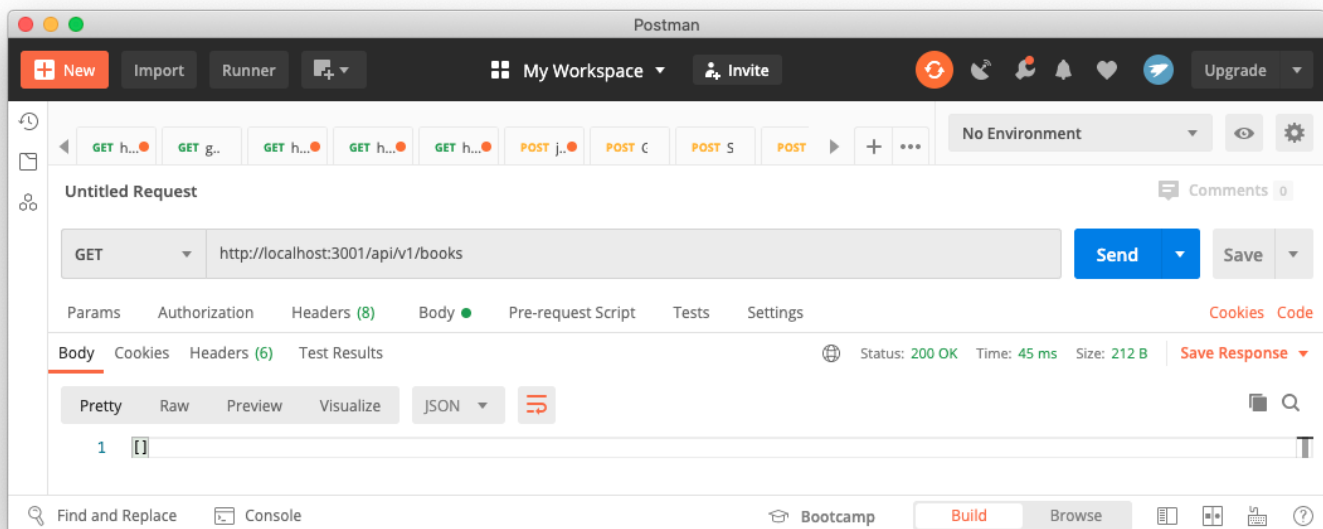
Archivo `config/config.service.ts`

```
public getTypeOrmConfig(): TypeOrmModuleOptions {  
  return {  
    type: 'postgres',  
  
    host: this.getValue('TUTORIAL_HOST'),  
    port: parseInt(this.getValue('TUTORIAL_PORT')),  
    username: this.getValue('TUTORIAL_USER'),  
    password: this.getValue('TUTORIAL_PASSWORD'),  
    database: this.getValue('TUTORIAL_DATABASE'),  
  
    entities: ['dist/**/*.entity.js'],  
    synchronize: true,  
  };  
}
```

TS

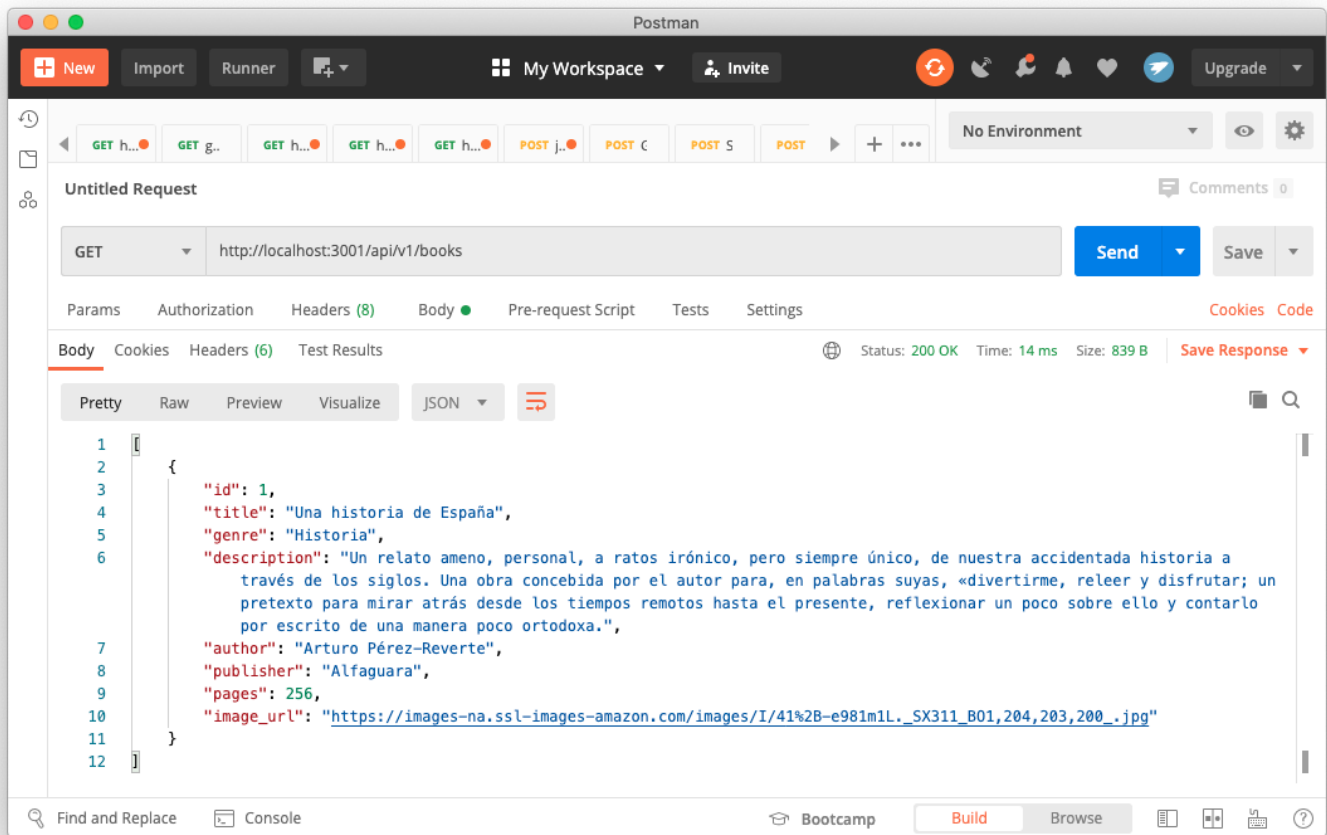
- 1 Servidor de bases de datos

Si ahora pedimos que nos devuelva todos los libros con el endpoint `/api/v1/books` y un método GET obtendremos una lista vacía, ya que partimos de una base de datos Postgres vacía.



Tras introducir un nuevo libro y volver a consultar los libros vemos cómo se recuperan los datos

sin problema, confirmándose lo sencillo que es cambiar de gestor de bases de datos si se usa un ORM.



8. Autenticación con JSON Web Tokens

Queremos restringir el acceso a los endpoints de la aplicación de forma que sólo tengan acceso los usuarios autenticados. Pero no queremos que se tengan que autenticar para cada petición. Necesitamos una forma que permita a los usuarios indicar que tienen una sesión iniciada válida.

Una forma sencilla de hacer esto es mediante JWT. En nuestro caso, ya partimos de un servidor de autorización que genera tokens de acceso a partir de usuario y contraseña. En este tutorial sólo añadiremos a la aplicación la parte de comprobación de la validez de los tokens y la restricción del acceso a los endpoints para tokens válidos.

JWT (JSON Web Tokens)

JWT es un estándar que define un método compacto y autocontenido que permite compartir de forma segura entre dos partes aserciones (claims) sobre una entidad (subject). Los datos están codificados en formato JSON incluidos en un *payload* o cuerpo del mensaje y están firmados digitalmente.

De forma predeterminada, los tokens no están cifrados. La cadena del token es una serialización en Base64 que se puede decodificar fácilmente (<https://jwt.io/>). La cadena del token está formada por tres partes:

- Cabecera: Indica algoritmo (p.e. HS256) y tipo de token (p.e. jwt)
- Payload o cuerpo: Aparecen todos los datos que queremos añadir
- Firma: Permite verificar si el token es válido



La firma del token se crea de forma que se pueda verificar si el remitente es quien dice ser. Dado que el token es una cadena fácilmente descifrable, si alguien manipula el token incluyendo datos o modificando el *payload* se verificaría que la firma del token no es correcta y no se puede confiar en el token recibido



Es conveniente incluir en el token una fecha de caducidad. Un token firmado es válido mientras no se haya superado su fecha de caducidad. Así, si alguien intercepta un token, sólo podrá usarlo mientras no caduque. Una fecha de caducidad corta no expondrá los recursos protegidos de la misma forma que si se intercepta una contraseña, que dejará los recursos expuestos mientras no se detecte la pérdida de la contraseña y no se cambie.

Instalaremos los paquetes siguientes:

```
$ npm install @nestjs/jwt passport passport-jwt @nestjs/passport
```

BASH

El JWT se enviará en la cabecera como `Bearer Token`.



Bearer Token o token de autorización es un esquema de autenticación HTTP (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication>). El método de autenticación *Bearer* debe entenderse como "dale acceso al portador (*bearer*) de este token".

Además, necesitaremos una *estrategia Passport* para la validación del token y configurar la clave secreta que se usó para firmar el token.

Passport y estrategias Passport

Passport (<http://www.passportjs.org/>) es un middleware de autenticación para Node. Se usa para autenticar peticiones. Usa un mecanismo de estrategias (<http://www.passportjs.org/packages/>) para configurar la forma de autenticación (Facebook, Twitter, GitHub, Auth0, OAuth, Google, LDAP, ...). El módulo `passport-jwt` es una estrategia Passport que permite asegurar peticiones usando JWT sin sesiones.

Crearemos una carpeta `utilities` donde guardaremos dos archivos:

- Estrategia JWT para Passport
- Módulo de autorización para ser importado por los controladores que quieran asegurar sus

endpoints

8.1. Configuración de la estrategia Passport

Configuraremos JWT como estrategia Passport para la autenticación. Definiremos:

- Extracción de JWT en cabecera como tipo `Bearer`
- Clave de verificación de firma del token
- Función de validación del *payload*

Archivo `utilities/jwt.strategy.ts`

```
import { PassportStrategy } from '@nestjs/passport';  
import { ExtractJwt, Strategy } from 'passport-jwt';  
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';  
  
@Injectable()  
export class JwtStrategy extends PassportStrategy(Strategy) {  
  constructor() {  
    super({  
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),  
      secretOrKey: 'secret',  
    });  
  }  
  
  async validate(payload: any): Promise<any> {  
    if (!payload) {  
      throw new HttpException('Invalid token', HttpStatus.UNAUTHORIZED);  
    }  
    return payload;  
  }  
}
```

TS

- 1 La clase extiende la estrategia de Passport
- 2 Extracción del token de la cabecera de la petición
- 3 Clave de verificación de la firma del token
- 4 Función de validación del token

8.2. Módulo de autenticación

El módulo de autenticación define JWT como la estrategia Passport a usar para los que importen este módulo. Además, define una propiedad (`user`) para enviar el *payload* del token en las peticiones.

Archivo `utilities/auth.module.ts`

```
import { Module } from '@nestjs/common';
import { PassportModule } from '@nestjs/passport';
import { JwtStrategy } from '../jwt.strategy';
@Module({
  imports: [
    PassportModule.register({ 1
      defaultStrategy: 'jwt', 2
      property: 'user', 3
      session: false,
    }),
  ],
  controllers: [],
  providers: [JwtStrategy], 4
  exports: [PassportModule], 5
})
export class AuthModule {}
```

TS

- 1 Configuración del módulo Passport
- 2 Configuración a estrategia `jwt`
- 3 Definición de propiedad `user` para el envío del *payload* en las peticiones
- 4 *provider* configurado en el paso anterior
- 5 Exportar el módulo ya configurado



El valor `jwt` definido en `defaultStrategy` se usará posteriormente a la hora de proteger los endpoints.

8.3. Restricción del acceso de los endpoints

Añadimos el módulo `AuthModule` definido en el paso anterior al módulo de los endpoints que queremos proteger. El módulo `AuthModule` definía la configuración de la estrategia y el servicio de validación JWT a utilizar.

Archivo `books/books.module.ts`

TS

```

...
import { AuthModule } from '../utilities/auth.module';

@Module({
  imports: [
    ...
    , AuthModule], 1
  providers: [...],
  controllers: [...],
})

export class BooksModule {}

```

1 Importación del módulo definido

Una vez definido el módulo, ya sólo falta proteger los endpoints. Podremos hacerlo de dos formas:

- Proteger de una vez todos los endpoints del controlador
- Proteger sólo los endpoints indicados

La protección se hará usando el decorador `@UseGuards()`. Si el decorador se coloca antes de la definición de la clase, quedan protegidos todos los endpoints definidos en la clase. Si no se desea una protección de todos los endpoints, se colocará `@UseGuards()` antes de la definición de aquellos endpoints que se quieran proteger.

A `@UseGuards()` se le pasa como argumento el nombre de estrategia de autenticación definida. En nuestro caso, la nuestra la habíamos definido como `jwt` en `Auth.module.ts`.

Archivo `books.controller.ts`

```

import {
  ...
  UseGuards, 1
} from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport'; 2
...
@Controller('books')
@UseGuards(AuthGuard('jwt')) 3
...
export class BooksController {
  ...
}

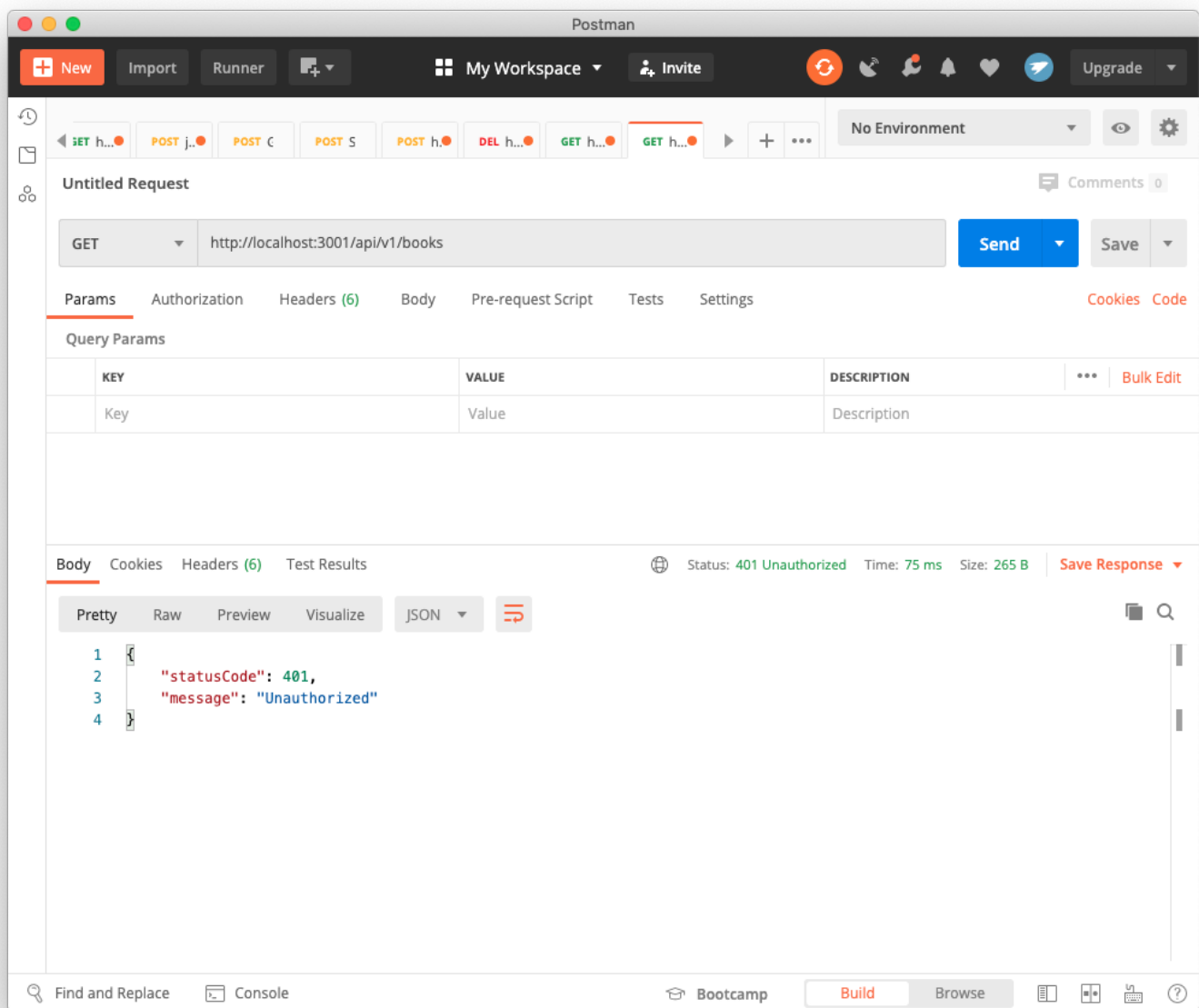
```

TS

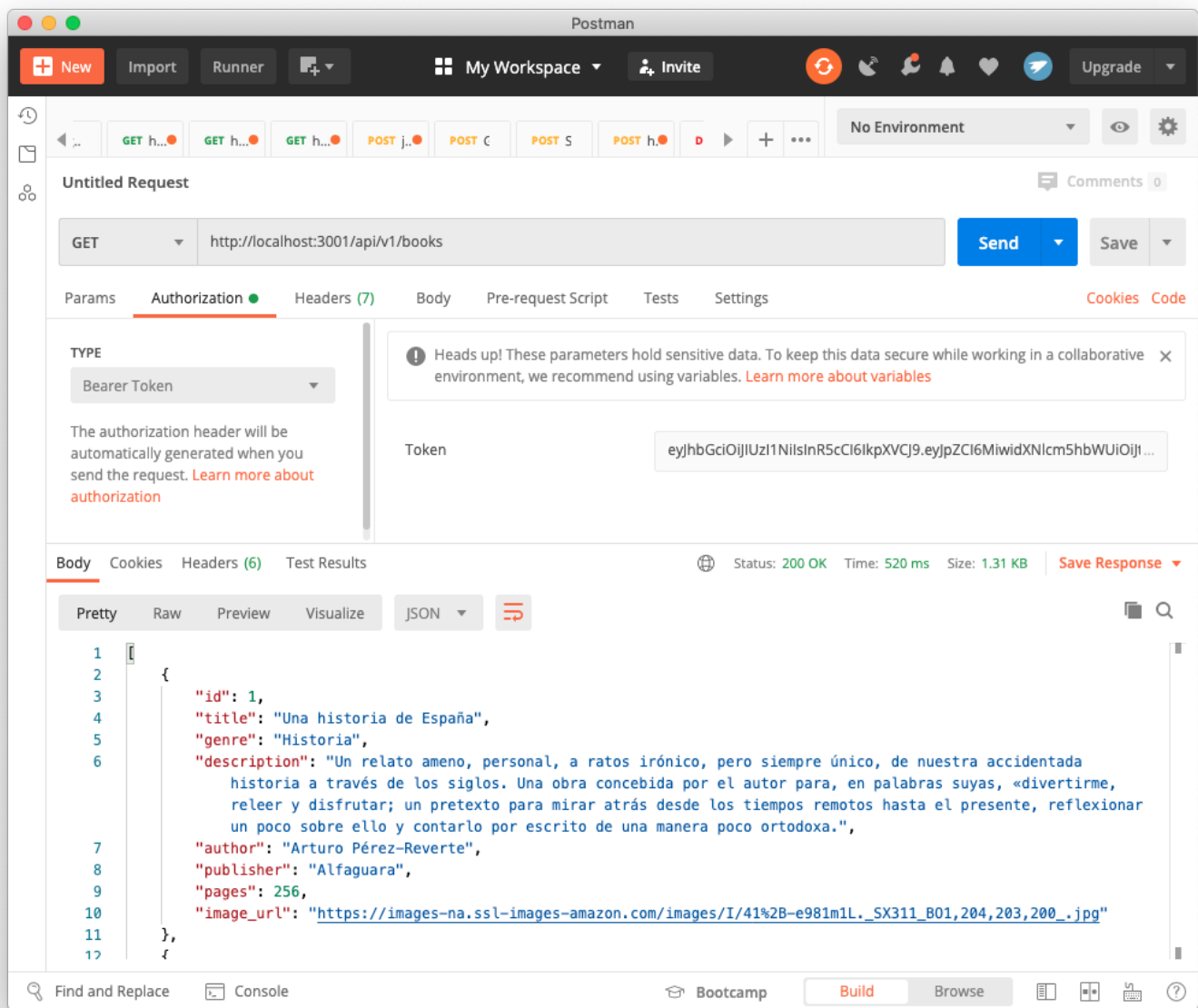
1 Importación del decorador UseGuards

- 2 Importación de AuthGuard para especificar la estrategia de autenticación a utilizar
- 3 Restricción del acceso a jwt de forma global (a nivel de clase) para todos los endpoints del controlador

Si tratamos de acceder sin token o con un token inválido a cualquier endpoint definido, obtendremos un mensaje de error 401 Unauthorized , tal y como muestra la figura.



Si pasamos en la cabecera de autorización pasamos el token indicando que es Bearer Token tendremos acceso a los endpoints, tal y como muestra la figura.



9. Documentación de la API con Swagger (OpenAPI)

NestJS cuenta con un módulo que permite la generación automática de la documentación en Swagger (OpenAPI) (<https://swagger.io/>). Esto permite obtener la documentación de la API y sus endpoints mediante decoradores en el código.

Comenzaremos instalando los paquetes de Swagger en el proyecto.

```
$ npm install --save @nestjs/swagger swagger-ui-express
```

A continuación hay que modificar el archivo `main.js` usando la clase `SwaggerModule`.

```
import { NestFactory } from '@nestjs/core';  
import { AppModule } from './app.module';  
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';  
  
async function bootstrap() {  
  const app = await NestFactory.create(AppModule);  
  app.setGlobalPrefix('api/v1');  
  
  // Configurar títulos de documentación  
  const options = new DocumentBuilder()  
    .setTitle('Bookstore REST API')  
    .setDescription('API REST de Bookstore')  
    .setVersion('1.0')  
    .addBearerAuth(  
      { type: 'http', scheme: 'bearer', bearerFormat: 'JWT', in: 'header' },  
      'access-token',  
    )  
    .build();  
  const document = SwaggerModule.createDocument(app, options);  
  
  // La ruta en que se sirve la documentación  
  SwaggerModule.setup('docs', app, document);  
  
  await app.listen(3000);  
}
```

TS

- 1 Importaciones necesarias
- 2 Configuración de opciones generales de la documentación (título, versión, ...)
- 3 Habilita el uso de autenticación JWT con Bearer Token
- 4 Nombre asignado a esta configuración de autenticación

- 5 Creación de la documentación con las opciones configuradas
- 6 Especificación de la ruta relativa donde se sirve la documentación Swagger



La configuración de `in: header` en `addBearerAuth()` permite una autenticación global asignándole un nombre (p.e. `access-token`). Si a nivel de clase se especifica `@ApiBearerAuth('access-token')` todos los endpoints quedarían autenticados tras la autenticación global. En cambio, si se opta por una autenticación individual, habría que incluir `@ApiBearerAuth('access-token')` antes de cada endpoint que quisiera usar el método de autenticación denominado `access-token`.

9.1. Documentación de DTOs, entidades, clases e interfaces

En clases DTO, así como en entidades, clases e interfaces, incluiremos un decorador `@ApiProperty()` antes de cada propiedad. A este decorador se le puede pasar un ejemplo que facilite la introducción al uso de la API.



El uso de decoradores en los DTO y entidades permite que aparezcan el tipo y un ejemplo definido siempre que use un DTO o una entidad, lo que facilita bastante la interacción con la documentación.

Archivo `books/book.dto.ts`

TS

```
import { ApiProperty } from '@nestjs/swagger'; 1

export class BookDto {
  @ApiProperty({ example: 'Don Quijote de la Mancha' }) 2
  readonly title: string;

  @ApiProperty({ example: 'Novela' })
  readonly genre: string;

  @ApiProperty({
    example: 'Esta edición del Ingenioso hidalgo don Quijote de la Mancha ...',
  })
  readonly description: string;

  @ApiProperty({ example: 'Miguel de Cervantes' })
  readonly author: string;

  @ApiProperty({ example: 'Santillana' })
  readonly publisher: string;

  @ApiProperty({ example: 592 })
  readonly pages: number;

  @ApiProperty({ example: 'www.imagen.com/quijote.png' })
  readonly image_url: string;
}
```

- 1 Importación de decoradores
- 2 Configuración de propiedades

La anotación Swagger de la entidad es prácticamente igual a la del DTO salvo que también incluye el `id`.

Archivo `books/book.entity.ts`

TS

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';
import { ApiProperty } from '@nestjs/swagger';

@Entity()
export class Book {
  @ApiProperty({ example: 99 })
  @PrimaryGeneratedColumn()
  id: number;

  @ApiProperty({ example: 'Don Quijote de la Mancha' })
  @Column()
  title: string;

  @ApiProperty({ example: 'Novela' })
  @Column()
  genre: string;

  @ApiProperty({
    example: 'Esta edición del Ingenioso hidalgo don Quijote de la Mancha ...',
  })
  @Column('text')
  description: string;

  @ApiProperty({ example: 'Miguel de Cervantes' })
  @Column()
  author: string;

  @ApiProperty({ example: 'Santillana' })
  @Column()
  publisher: string;

  @ApiProperty({ example: 592 })
  @Column()
  pages: number;

  @ApiProperty({ example: 'www.imagen.com/quijote.png' })
  @Column()
  image_url: string;
}
```



También hay que incluir decoradores `@ApiProperty` en interfaces y otras clases definidas para tipado.

9.2. Documentación de los controladores

Los métodos de los controladores se pueden agrupar mediante etiquetas Swagger. Para ello se usa el decorador `@ApiTags()`. Se puede usar el decorador a nivel de clase, lo que combinará a todos los métodos en el mismo grupo. También se puede usar a nivel de método.

Si se dispone de autenticación JWT, se incluirá el decorador `@ApiBearerAuth()` con el nombre usado para denominar al método de autenticación definido. Si el decorador se usa a nivel de clase, todos los endpoints de la clase quedarán autenticados al realizar una autenticación global.

En cada operación se incluirá:

- Un decorador `@ApiOperation()` para proporcionar una descripción para la operación
- Un decorador `@ApiResponse()` por cada respuesta que proporcione la operación (p.e. 200, 403, ...)

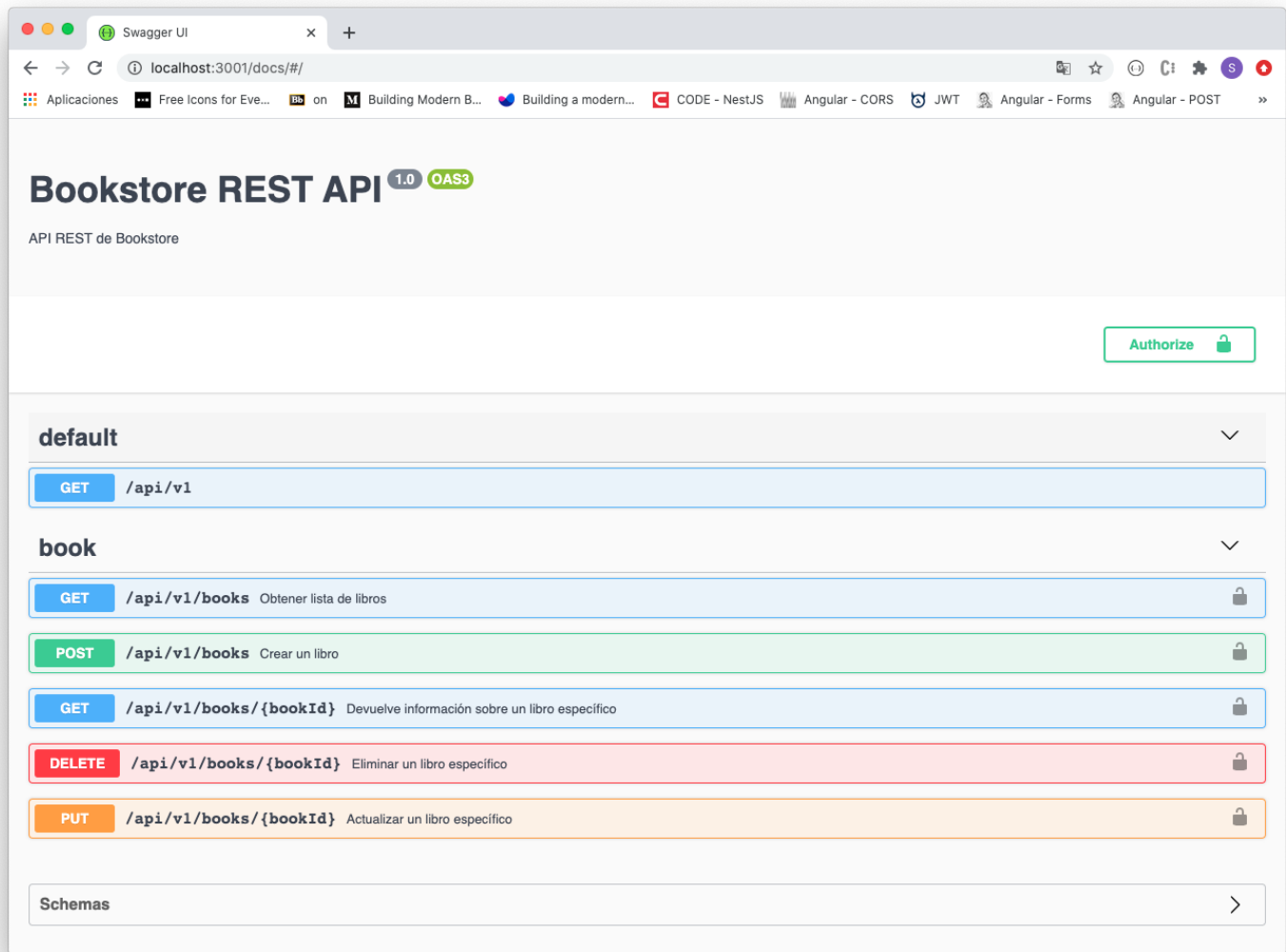
A continuación se muestra un fragmento de la anotación en `books/books.controller.ts`

```
...
import { BookDto } from './book.dto'; 1
import { Book } from './book.entity'; 2
import { 3
  ApiOperation,
  ApiResponse,
  ApiTags,
  ApiBearerAuth,
} from '@nestjs/swagger';
...
@ApiTags('book') 4
@Controller('books')
@UseGuards(AuthGuard('jwt')) 5
@ApiBearerAuth('access-token') 6
export class BooksController {
  ...
  /** 7
   *
   * @returns {Book[]} Devuelve una lista de libros
   * @param {Request} request Lista de parámetros para filtrar
   */
  @Get()
  @ApiOperation({ summary: 'Obtener lista de libros' }) 8
  @ApiResponse({ 9
    status: 201,
    description: 'Lista de libros',
    type: Book, 10
  })
  findAll(@Req() request: Request): Promise<Book[]> {
    ...
  }
  ...
}
```

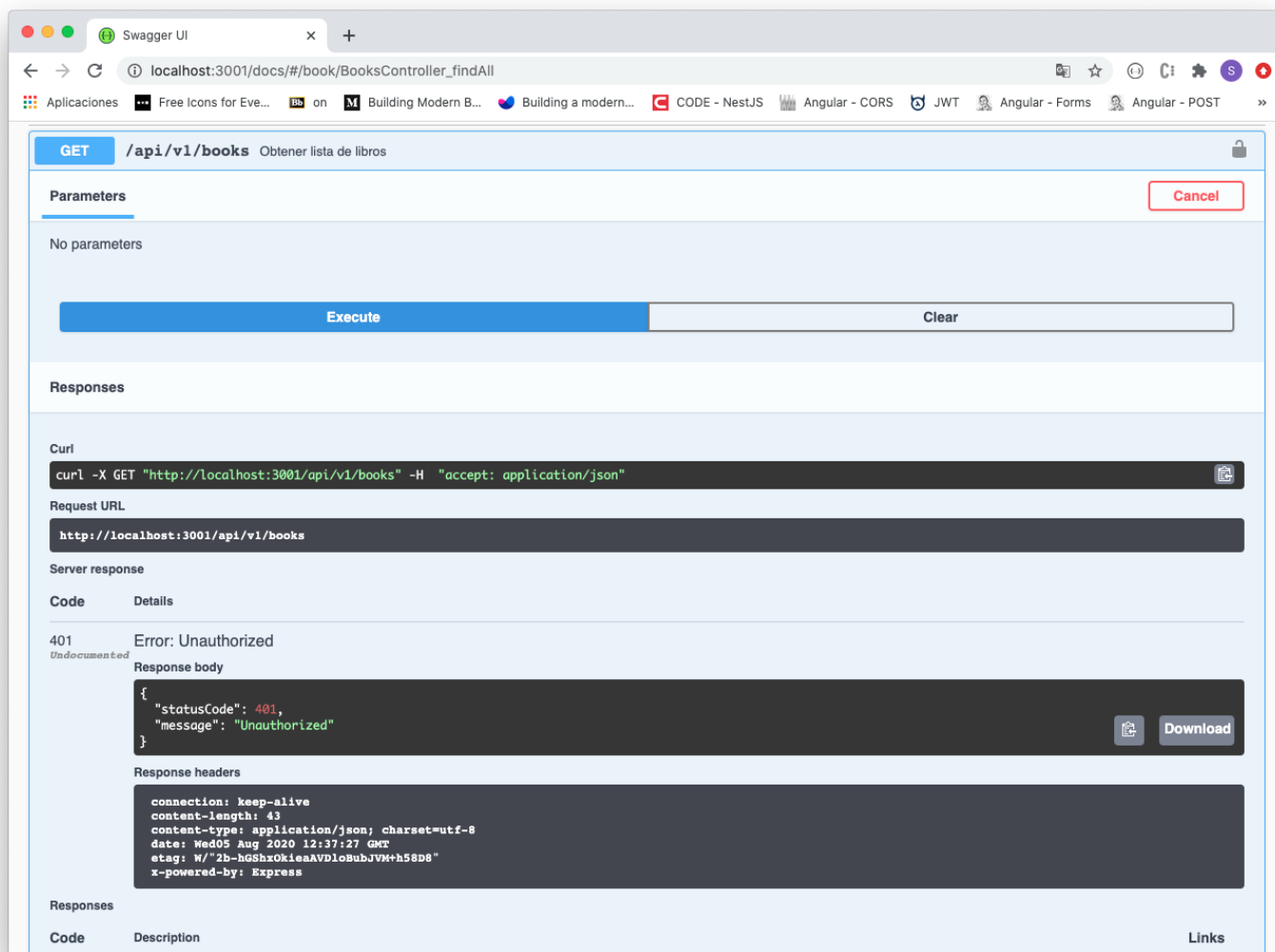
TS

- 1 Importación del DTO para enlazar bien la documentación
- 2 Importación de la entidad para enlazar bien la documentación
- 3 Importación de paquetes Swagger
- 4 Especificación de la etiqueta para combinar a todos las operaciones de este controlador en el grupo book
- 5 Protección con JWT a nivel de clase de todos los endpoints
- 6 Configuración de autenticación en Swagger a nivel de clase
- 7 Documentación del retorno y de los parámetros del endpoint
- 8 Descripción de la operación
- 9 Respuesta 201
- 10 Al especificar el tipo, se puede ver un ejemplo de la respuesta en la documentación

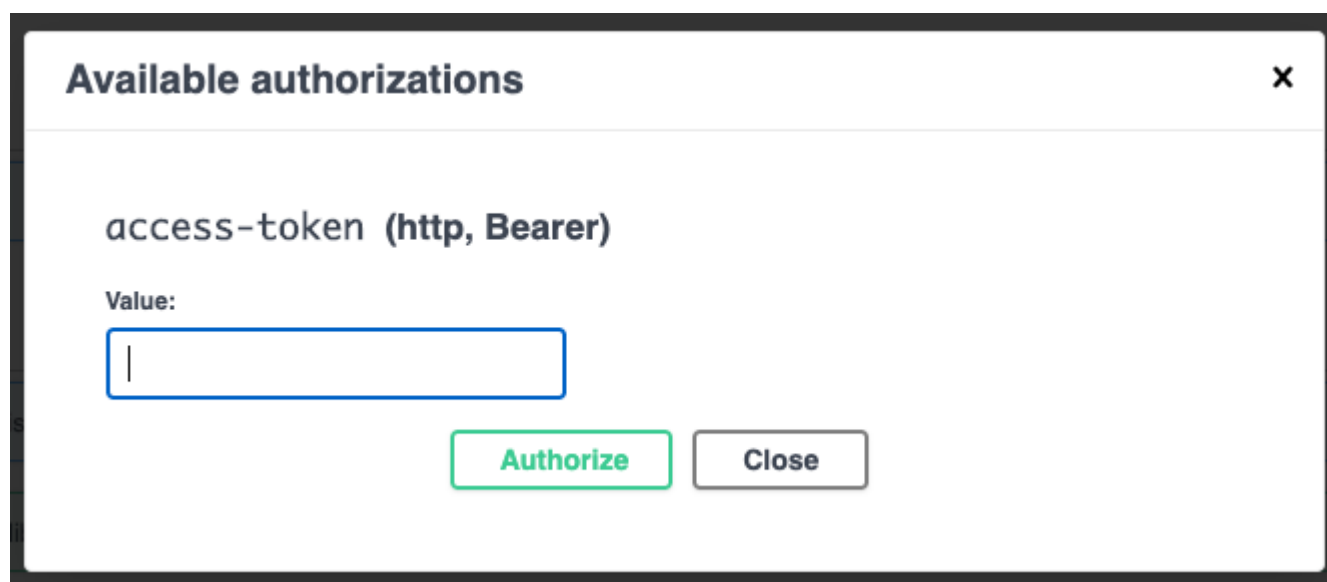
La figura siguiente muestra cómo quedaría inicialmente la documentación servida en la ruta docs . Como aún no se ha proporcionado el token, los endpoints aparecen con un candado abierto indicando que no se puede su acceso.



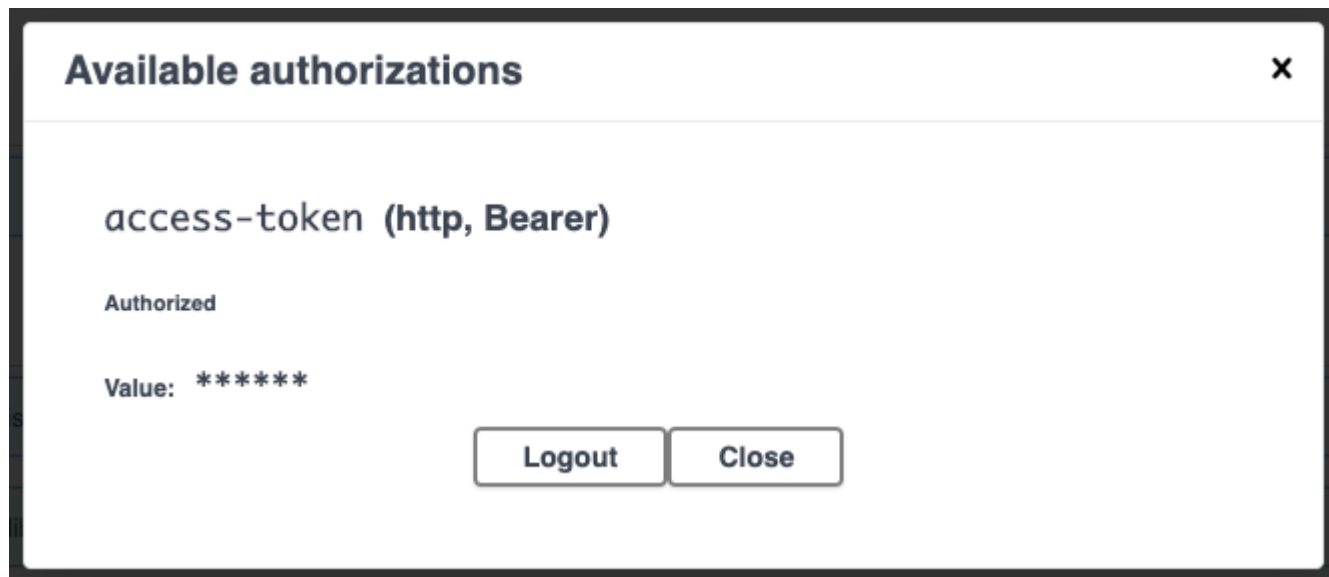
Si probásemos un endpoint (p.e. `GET /books` para obtener la lista de todos los libros) con `Try out` se nos rechazaría el acceso, tal y como ilustra la figura siguiente.



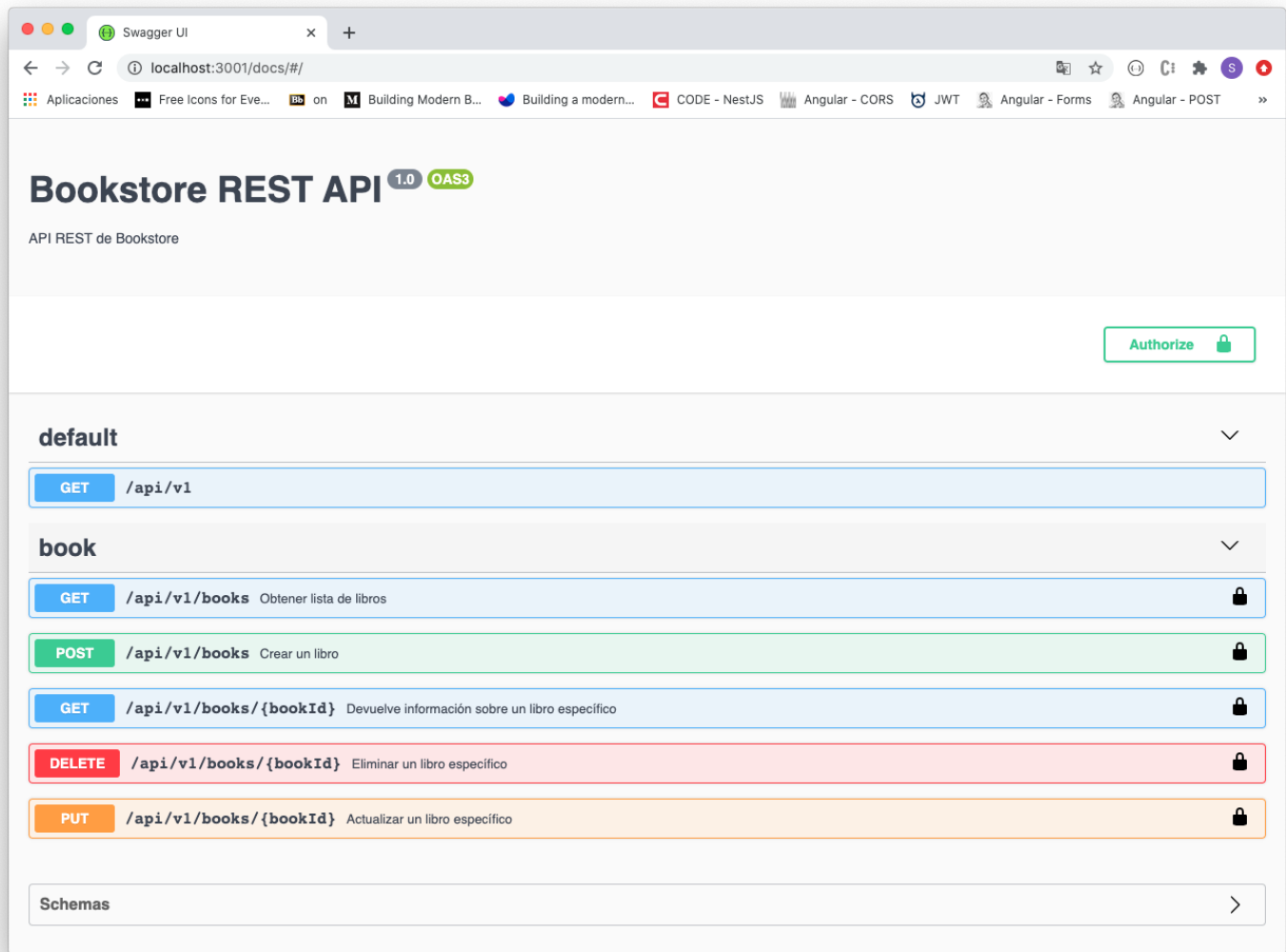
Para introducir el token, pulsaremos el botón `Authorize` superior. En el cuadro de diálogo introducimos el token y pulsamos sobre `Authorize`



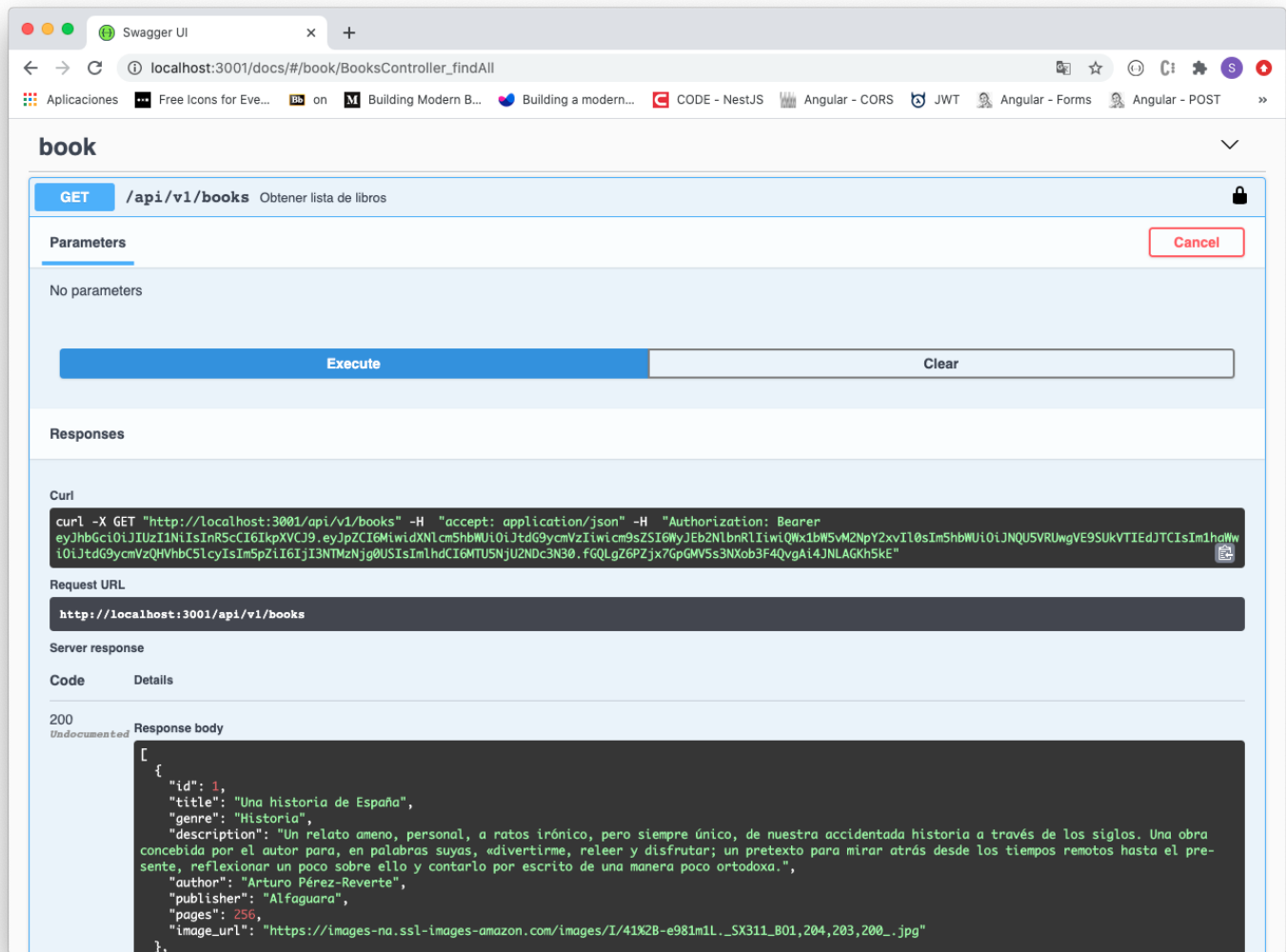
Si el token introducido es válido, quedaremos autorizados.



Al quedar autorizados, como definimos la autenticación para todo el controlador, quedaría abierto el acceso a todos los endpoints, mostrándose ahora todos los candados cerrados.



Si ahora volvemos a probar el endpoint para obtener la lista de libros, la lista se recuperará y se mostrará en el propio Swagger.

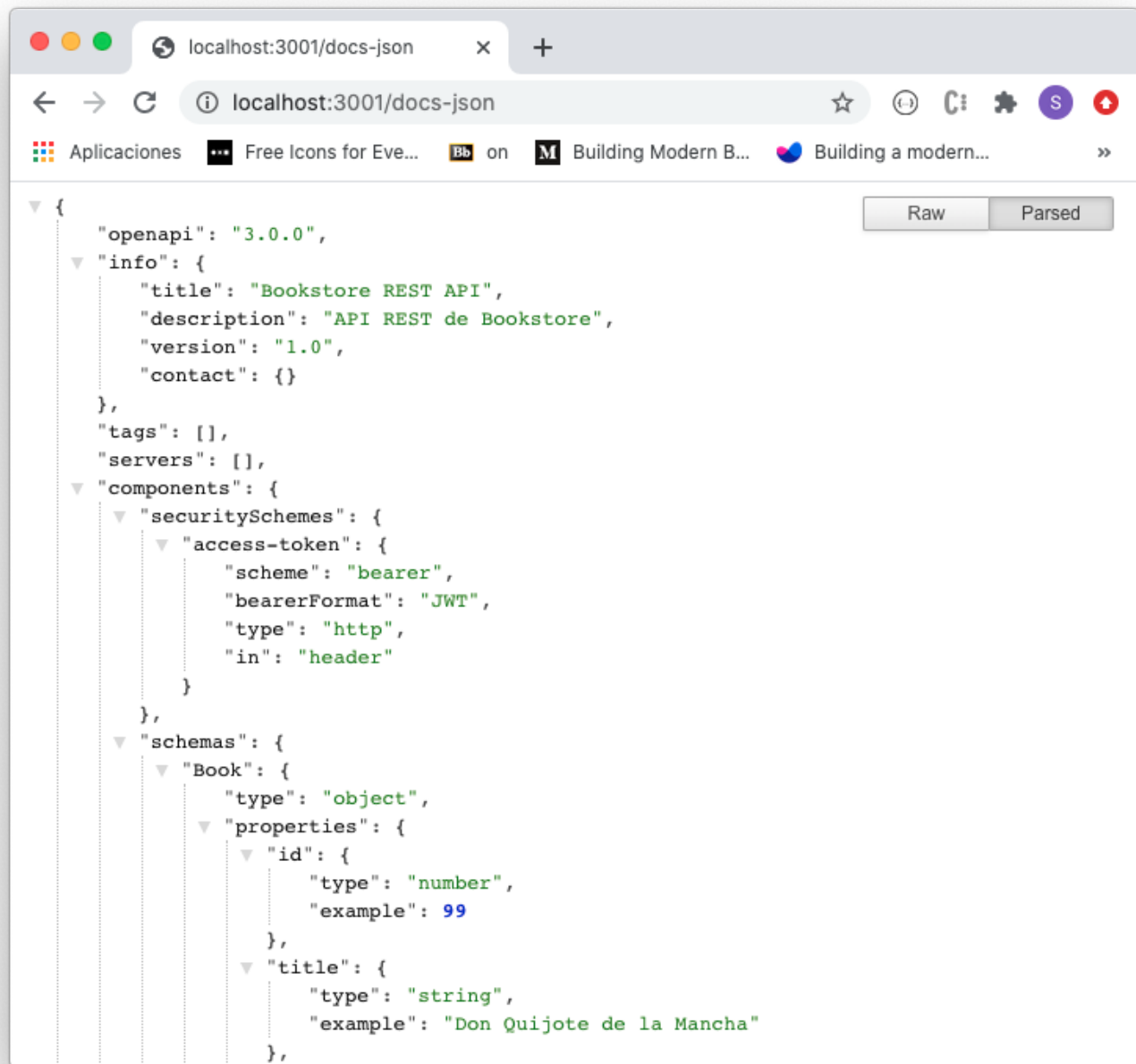


Esto hace a Swagger una opción muy interesante para los proyectos de APIs ya que no sólo es una herramienta de documentación, sino que también permite la interacción directa con la API. Con una buena documentación enriquecida con la descripción de sus parámetros, tipos y ejemplos tendremos una plataforma extraordinaria para la documentación y uso de APIs.

9.3. Descarga de JSON

Para generar y poder descargar un archivo Swagger JSON basta con añadir `-json` a la ruta desde la que se sirve la documentación. Este archivo podrá ser alojado en una plataforma desde la que se sirva la documentación de las APIs de la organización.

En nuestro caso, `http://localhost:3001/docs-json` generará el archivo Swagger JSON de nuestra aplicación.



```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Bookstore REST API",
    "description": "API REST de Bookstore",
    "version": "1.0",
    "contact": {}
  },
  "tags": [],
  "servers": [],
  "components": {
    "securitySchemes": {
      "access-token": {
        "scheme": "bearer",
        "bearerFormat": "JWT",
        "type": "http",
        "in": "header"
      }
    },
    "schemas": {
      "Book": {
        "type": "object",
        "properties": {
          "id": {
            "type": "number",
            "example": 99
          },
          "title": {
            "type": "string",
            "example": "Don Quijote de la Mancha"
          }
        }
      }
    }
  }
}
```



El elemento `servers` está sin definir. De cara a subir este JSON a un servidor de Swagger, se debería configurar este elemento con el nombre DNS o IP del servidor donde se aloja la API para poder interactuar con la API.

Para más información sobre Swagger, consultar la [documentación oficial](https://docs.nestjs.com/recipes/swagger)
(<https://docs.nestjs.com/recipes/swagger>)

9.4. Cambio del frontend

NestJS-Redoc (<https://www.npmjs.com/package/nestjs-redoc>) es un frontend para la especificación de la API en Swagger. Está basado en Redoc (<https://github.com/Redocly/redoc>) y permite una presentación más sencilla y elaborada que la proporcionada por Swagger UI ofreciendo además funciones de búsqueda.

La instalación se realiza con

```
$ npm install --save nestjs-redoc@1.3.1
```

BASH



A fecha de la creación de este tutorial la versión actual de NestJS Redoc (1.3.2) tiene una incompatibilidad con la versión actual de NestJS (7.0.0). Mientras se resuelve hay que usar la versión 1.3.1 de NestJS Redoc.

NestJS-Redoc se apoya en la configuración realizada con Swagger y añade unas opciones propias (p.e. logo y título de la página). Al igual que con Swagger, la configuración de Redoc se realiza en `main.ts`. Sin embargo, hay que indicar que la documentación ya no la sirve Swagger UI, sino Redoc. De esto se encarga el método `setup` de `RedocModule` tal y como se muestra a continuación.

Archivo `main.ts`

TYPESCRIPT

```
...
import { RedocModule, RedocOptions } from 'nestjs-redoc'; 1

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Configurar títulos de documentación
  const options = new DocumentBuilder() 2
    .setTitle('Sample REST API')
    .setDescription('Sample API REST Description')
    .setVersion('1.0')
    .addBearerAuth(
      { type: 'http', scheme: 'bearer', bearerFormat: 'JWT', in: 'header' },
      'access-token',
    )
    .build();
  const document = SwaggerModule.createDocument(app, options);

  const redocOptions: RedocOptions = { 3
    favicon: 'https://www.ual.es/favicon.ico',
    title: 'API Reservas',
    logo: {
      url:
        'https://www.ual.es/application/themes/ual/images/logoual25-300px.png',
      backgroundColor: '#0082B7',
    },
    sortPropsAlphabetically: true,
    hideDownloadButton: false,
    hideHostname: false,
    noAutoAuth: false,
  };

  // La ruta en que se sirve la documentación
  //SwaggerModule.setup('docs', app, document); 4
  await RedocModule.setup('/docs', app, document, redocOptions); 5

  await app.listen(3000);
}
bootstrap();
```

- 1 Importaciones de Redoc
- 2 Configuración de opciones generales de la documentación Swagger
- 3 Configuración de las opciones de Redoc
- 4 La documentación ya no la sirve Swagger UI
- 5 Servir la documentación con Redoc usando las opciones definidas en `redocOptions`

La figura siguiente ilustra el nuevo aspecto de la documentación Swagger.

The screenshot shows the Redoc API documentation for 'API Reservas' at the URL `localhost:3000/docs/#operation/ReservasController_findBook`. The interface is divided into three main sections:

- Left Sidebar:** Contains the 'UNIVERSIDAD DE ALMERÍA' logo, a search bar, and a list of API endpoints categorized by method (GET, POST, DELETE). The selected endpoint is 'Devuelve información sobre una reserva específica' (GET).
- Main Content Area:** Displays details for the selected endpoint, including 'AUTHORIZATIONS: access-token', 'PATH PARAMETERS' (id: string, required), and 'Responses'. The response for status 201 is shown as 'Datos de la reserva'.
- Right Panel:** Shows 'Response samples' for the selected endpoint. It displays a JSON response for status 201 with fields: `"id": 99`, `"user": "ggf906"`, `"start": "2020-09-16T08:00:00"`, and `"end": "2020-09-16T08:00:00"`. Below this, the 'DELETE /reservas/{id}' endpoint is also visible.



Para más información sobre las opciones disponibles en Redoc, consultar la [documentación oficial](https://github.com/mxarc/nestjs-redoc#readme) (<https://github.com/mxarc/nestjs-redoc#readme>).

Puedes encontrar ejemplos de uso de Redoc en:

- [Docker Engine API](https://docs.docker.com/engine/api/v1.25/) (<https://docs.docker.com/engine/api/v1.25/>)
- [Commbbox](https://www.commbbox.io/api/) (<https://www.commbbox.io/api/>)
- [Zuora](https://www.zuora.com/developer/api-reference/) (<https://www.zuora.com/developer/api-reference/>)



La opción de envío de peticiones a la API a través de Swagger (*Try it out*) es una función de pago en Redoc ([Redocly](https://redoc.ly/) (<https://redoc.ly/>)) por lo que el uso de Redoc en su versión open source se limita a la documentación sin contar con la funcionalidad de envío de peticiones.

10. Logging

A medida que las aplicaciones se complican y a medida que se les exige mayor rendimiento se vuelve más necesario contar un registro de logs que nos ayude a encontrar fallos o problemas de rendimiento. NestJS incorpora un sistema de logging que permite controlar los mensajes que se registran en el log y especificar su salida. Sin embargo, Nest recomienda usar otros paquetes de logging más avanzados y versátiles para sistemas en producción, como Winston

(<https://github.com/winstonjs/winston>). Entre las características de Winston se encuentran: soporte para gran cantidad de opciones de almacenamiento, niveles de log y formateo de logs.

- Opciones de almacenamiento: Winston es una librería de logging que permite varios *transportes* (<https://github.com/winstonjs/winston/blob/master/docs/transport.md#winston-core>). Básicamente, un transporte es un dispositivo de almacenamiento para almacenar logs. Cada instancia de un logger de Winston puede tener varios transportes configurados para niveles diferentes. Ejemplos de transportes son consola, archivo, archivos de rotación diaria, Syslog, Datadog, Elasticsearch o MongoDB.



Una opción de transporte centralizada, como la basada en Elasticsearch, evitaría el problema de la fragmentación de logs que se produce cuando tenemos varias copias de la aplicación (p.e. en varios contenedores), cada una con sus archivos de log independientes.

- Niveles: Los niveles de log indican la gravedad, que van desde una caída del sistema hasta el aviso de una función marcada como obsoleta. Los niveles de log ayudan a ver rápidamente los logs que necesitan atención. Para cada nivel se puede configurar la cantidad de datos y de detalles a registrar.



Los niveles de log se priorizan de 0 a 5 (de mayor a menor prioridad)

- 0: error
- 1: warn
- 2: info
- 3: verbose
- 4: debug
- 5: silly

Al especificar un nivel de log para un transporte concreto, se registrará cualquier cosa con ese nivel o con una prioridad mayor (p.e. si se especifica `info`, se registrará cualquier cosa al nivel `info` así como a las niveles `warn` y `error`).

- Formato: Winston ofrece formateo en JSON, uso de colores y manipulación de formatos. ya que posteriormente surgen problemas si todo son cadenas.

10.1. Configuración de Winston

Comenzamos instalando con

```
npm install --save nest-winston winston
```

BASH

A continuación, se configuran las opciones de nivel de log, transporte y formato en `app.module.ts`. En este ejemplo se registran los logs con nivel `info` (que registrará `info`, `warn` y `error`). Las opciones de formato incluyen la fecha, la interpolación de cadenas y la salida en JSON. Como transportes, se usarán 3 archivos de logs independientes (uno para errores, otro para `debug` y otro para `info`) y salida por consola para nivel `debug`.

Archivo `app.module.ts`

TS


```
...
import { WinstonModule } from 'nest-winston'; 1
import * as winston from 'winston';
import * as path from 'path';

@Module({
  imports: [
    ...
    WinstonModule.forRoot({
      level: 'info', 2
      format: winston.format.combine( 3
        winston.format.timestamp({
          format: 'YYYY-MM-DD HH:mm:ss',
        }),
        winston.format.errors({ stack: true }),
        winston.format.splat(),
        winston.format.json(),
      ),
      transports: [ 4
        new winston.transports.File({
          dirname: path.join(__dirname, '../log/debug/'),
          filename: 'debug.log',
          level: 'debug',
        }),
        new winston.transports.File({
          dirname: path.join(__dirname, '../log/error/'),
          filename: 'error.log',
          level: 'error',
        }),
        new winston.transports.File({
          dirname: path.join(__dirname, '../log/info/'),
          filename: 'info.log',
          level: 'info',
        }),
        new winston.transports.Console({ level: 'debug' }),
      ],
    }),
  ],
  controllers: [...],
  providers: [...],
})
export class AppModule {}
```

- 1 Importaciones necesarias de Winston y paths para tratar con las rutas de los archivos de log
- 2 Configuración del nivel info
- 3 Formato definido para las entradas de log

4 Transportes: 3 archivos y salida por consola para nivel mínimo de debug



El *transporte* para archivos tiene otras opciones interesantes como:

- **maxsize** : Tamaño máximo en bytes del archivo de log. Al superar el tamaño se crea un nuevo archivo de log.
- **maxFiles** : Limita el número de archivos a crear cuando se excede el tamaño máximo del archivo de logs
- **zippedArchive** : Si es `true` , se comprimen todos los archivos de log excepto el actual.

10.2. Registro de logs con Winston

Aquí vamos a ver cómo un endpoint registra una entrada de log. En el controlador y en general en cualquier clase que usase Winston, haríamos la configuración siguiente:

```
import { 1
  ...
  Inject } from '@nestjs/common';
import { WINSTON_MODULE_PROVIDER } from 'nest-winston';
import { Logger } from 'winston';

@Controller()
export class SomeController {
  constructor(
    ...
    @Inject(WINSTON_MODULE_PROVIDER) private readonly logger: Logger, 2
  ) {
    ...
  }
  ...
}
```

TS

- 1 Importación de paquetes y opciones de Winston
- 2 Winston se inyecta en el constructor y queda disponible como `logger`



Comprobar que el `Logger` que se importa es el de Winston y no otro, como el de Nest o el de TypeORM.

Para crear una entrada de log se indica el nivel de la entrada de log, y concatenaríamos pares

clave-valor que queremos registrar en el log.

```
this.logger.log({  
  level: 'info',  
  message: 'Hola',  
  service: 'Books',  
});
```

TS

Como se trata de una entrada de tipo `info`, quedaría registrada en `log/info.log`:

```
{"level":"info","message":"Hola","service":"Books","timestamp":"2020-08-05 19:14:08"} 1
```

JSON

1 timestamp puede ser incluido de forma automática si se configura así en las opciones de las entradas de log



En una entrada de log son obligatorios los campos `level` y `message`.

10.3. Log de operaciones de la API

Para finalizar veremos cómo registrar en el log operaciones de la API. Pasaremos por alto el control de errores y sólo haremos el caso feliz en que la operación se lleva a cabo con éxito. La entrada de log incluirá lo siguiente:

- `level`: Indica el nivel de la entrada de log
- `message`: Texto de la entrada
- `statusCode`: Código HTTP de la respuesta
- `method`: Método HTTP de la petición
- `url`: URL solicitada
- `user`: Usuario que ha realizado la petición. Se obtiene del JWT enviado en la cabecera
- `duration`: Tiempo en ms para resolver la petición
- `timestamp`: Instante en el que se ha realizado la petición

La mecánica que usaremos para atender una petición de la API será la siguiente:

1. Obtener la fecha del sistema
2. Llamar al servicio que resuelve la petición

3. Llamada a una función auxiliar que escribe una entrada en el log

4. Devolver los datos de la petición



Para obtener datos de la petición, como el método HTTP, url, usuario y demás, incluiremos un parámetro de tipo `Request` en cada función de la API.

Archivo `books/books.controller.ts`

JSON

```
...  
@Get()  
...  
findAll(@Req() request: Request): Promise<Book[]> { 1  
  let startTime = Date.now(); 2  
  let data = this.booksService.findAll(request.query); 3  
  
  this.writeLog(startTime, request, 200); 4  
  
  return data; 5  
}  
...
```

- 1 Incluir un parámetro `Request` para incluir datos como la url, método HTTP y demás en la entrada de log
- 2 Obtener la hora antes de llamar al servicio que resuelve la petición
- 3 Llamar al servicio
- 4 Llamar a la función auxiliar que escribe la entrada de log
- 5 Devolver los datos de la petición

Función auxiliar

Archivo `books/books.controller.ts`

JSON

```
...
writeLog(startTime: any, request: any, statusCode: number) {
  let finishTime = Date.now();
  let elapsedTime = finishTime - startTime;

  this.logger.log({
    level: 'info',
    message: '',
    statusCode: statusCode,
    method: request['method'],
    url: request['url'],
    user: request['user'].username,
    duration: elapsedTime,
  });
}
...
```

Tras hacer una petición `GET /api/v1/books/1` obtendríamos esta entrada en el archivo de logs `log/info/info.log`

```
{"level":"info","message":"","statusCode":200,"method":"GET","url":"/api/v1/books/1","user":"mtorres","duration":8,"timestamp":"2020-08-06 13:01:49"}
```

JSON



En este ejemplo se ha optado por definir una entrada de log con campos independientes fuera de `message`. Otra opción es incluirlos dentro de `message` y usar interpolación de variables.

11. Documentación del código

NestJS usa Compodoc (<https://compodoc.app/>), una herramienta de documentación para Angular. Al documentar el código, los miembros del equipo de desarrollo podrán entender fácilmente las características de la aplicación o librería. La documentación se anota mediante JSDoc (<https://jsdoc.app/>) siguiendo este esquema:

```
/**  
 * Supported comment  
 */
```

TS

Entre los tags JSDoc, destacan:

- @returns {Type} Description
- @param {Type} Name Description
- @ignore para excluir un fragmento de código de la documentación

Para instalar Compodoc en un proyecto NestJS basta con añadir el paquete:

```
$ npm i -D @compodoc/compodoc
```

BASH

La documentación se generará desde la línea de comandos mediante `npx` (una herramienta para ejecutar paquetes de Node disponible con `npm 6`). Esto generará una carpeta `documentation` en el proyecto que se podrá servir con el proyecto o en un portal de ámbito más global donde estén todas las documentaciones de los proyectos desarrollados por el equipo.

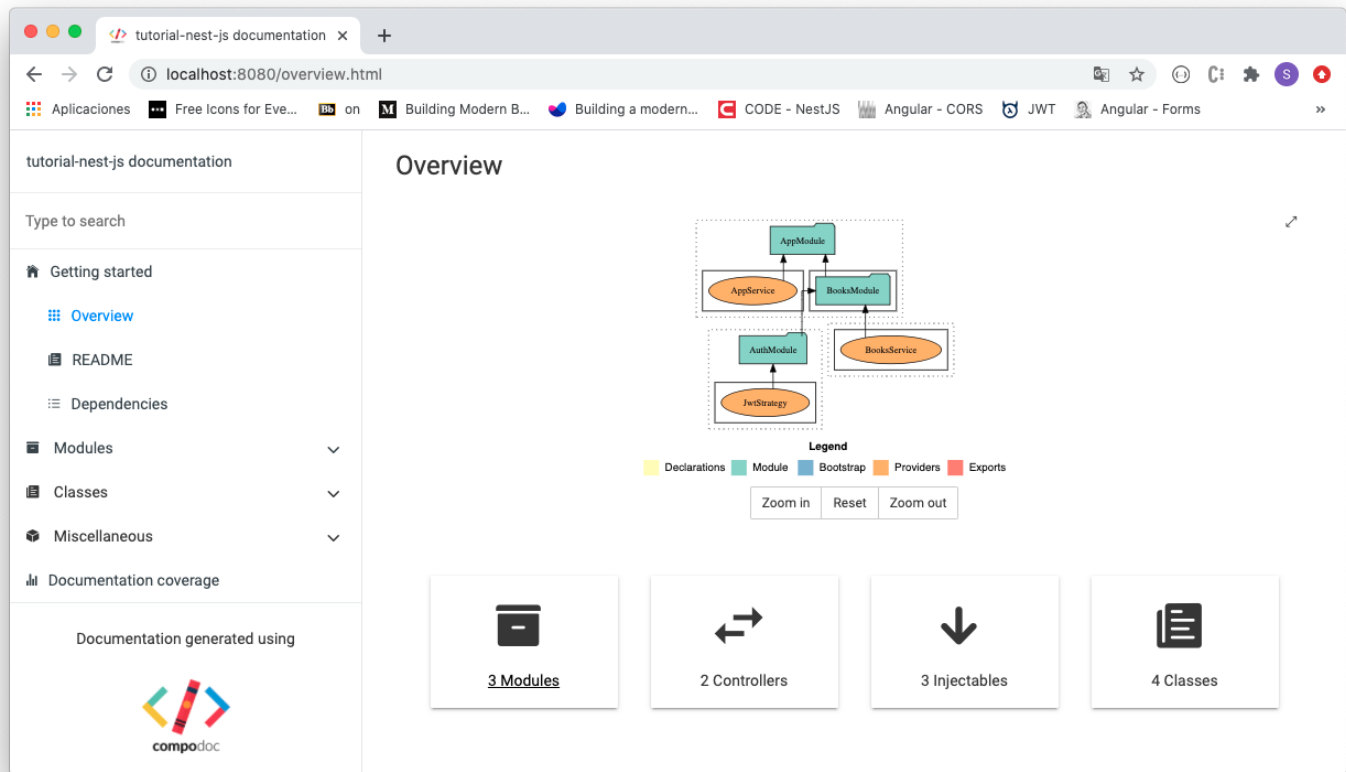
```
$ npx compodoc -p tsconfig.json -s --theme material
```

BASH



El parámetro `-s` inicia un servidor en el puerto 8080 para poder consultar la documentación. El parámetro `--theme material` aplica el tema `material` a la documentación. Para más información sobre las opciones de uso, consultar la [documentación oficial](https://compodoc.app/guides/options.html) (<https://compodoc.app/guides/options.html>)

Compodoc genera una página `Overview` donde presenta un diagrama con los distintos componentes y sus relaciones, algo muy interesante para hacerse una primera idea de la composición e interacción del software desarrollado.



La figura siguiente ilustra el formato de la documentación de un componente de la aplicación.

tutorial-nest-js documentation

Type to search

- Getting started
 - Overview
 - README
 - Dependencies
- Modules
 - AppModule
 - Controllers
 - Injectables
 - AuthModule
 - Injectables
 - BooksModule
 - Controllers
 - BooksController**
 - Injectables
 - Classes

localhost:8080/modules/AuthModule.html

Controllers / BooksController

Info Source

File

src/books/books.controller.ts

Prefix

books

Index

Methods		
createBook	findAll	updateBook
deleteBook	findBook	writeLog

Methods

createBook

```
createBook(request: Request, newBook: BookDto)
```

Decorators :

```
@Post()  
@ApiOperation({summary: 'Crear un libro'})  
@ApiResponse({status: 201, description: 'Datos del libro creado', type: Book})  
@ApiResponse({status: 403, description: 'Forbidden.'})
```

Para más información sobre JSDoc, consultar la [documentación oficial](https://compodoc.app/) (<https://compodoc.app/>)

Apéndice A. Datos de ejemplo

JSON

```
[
  {
    "title": "Una historia de España",
    "genre": "Historia",
    "description": "Un relato ameno, personal, a ratos irónico, pero siempre único, de nuestra accidentada historia a través de los siglos. Una obra concebida por el autor para, en palabras suyas, «divertirme, releer y disfrutar; un pretexto para mirar atrás desde los tiempos remotos hasta el presente, reflexionar un poco sobre ello y contarle por escrito de una manera poco ortodoxa.",
    "author": "Arturo Pérez-Reverte",
    "publisher": "Alfaguara",
    "pages": 256,
    "image_url": "https://images-na.ssl-images-amazon.com/images/I/41%2B-e981m1L._SX311_B01,204,203,200_.jpg"
  },
  {
    "title": "Historia de España contada para escépticos",
    "genre": "Historia",
    "description": "Como escribe el autor, no pretende ser veraz, justa y desapasionada, porque ninguna historia lo es. No está hecha para halagar a reyes y gobernantes, ni pretende halagar a los banqueros, ni a la Conferencia Episcopal, ni al colectivo gay.",
    "author": "Juan Eslava Galán",
    "publisher": "Booket",
    "pages": 592,
    "image_url": "https://images-na.ssl-images-amazon.com/images/I/51IyZ5Mq8YL._SX326_B01,204,203,200_.jpg",
    "__v": 0
  },
  {
    "title": "El enigma de la habitación 622",
    "genre": "Ficción contemporánea",
    "description": "Vuelve el «principito de la literatura negra contemporánea, el niño mimado de la industria literaria» (GQ): el nuevo thriller de Joël Dicker es su novela más personal. ",
    "author": "Joël Dicker",
    "publisher": "Alfaguara",
    "pages": 624,
    "image_url": "https://images-na.ssl-images-amazon.com/images/I/41KiZbwOhhL._SX315_B01,204,203,200_.jpg"
  }
]
```

Apéndice B. Generación de código

A la hora de abordar un proyecto de backend hay tareas repetitivas que son susceptibles de ser sometidas a algún grado de automatización. Esto es algo deseable ya que de forma directa esto aumentará nuestra eficacia por un lado, y por otro reducirá la introducción de errores. Veamos aquí dos generadores de código útiles.

Generador de archivos para una entidad

Normalmente, para cada entidad de nuestro proyecto de backend tendremos que crear un módulo, un controlador, un servicio, una clase para la entidad y algunos DTO (p.e. el de crear y el de modificar). Todos estos archivos pueden ser generados de forma automática por el CLI de NestJS generando lo que se denomina un *recurso*. Desde la carpeta del proyecto ejecutaremos el comando siguiente para cada *recurso* que queramos crear.

```
$ nest generate resource <nombre-recurso>
```

BASH

En primer lugar nos pedirá el tipo de nivel de transporte que queremos usar. Elegiremos REST API

```
? What transport layer do you use? (Use arrow keys)
> REST API
  GraphQL (code first)
  GraphQL (schema first)
  Microservice (non-HTTP)
  WebSockets
```

BASH

En segundo lugar aceptaremos la generación de los endpoints básicos para las operaciones CRUD.

```
? Would you like to generate CRUD entry points? (Y/n)
```

BASH

En el caso de que hayamos elegido crear los recursos para `users` se crearán los archivos siguientes y se actualizará `src/app.module.ts` para añadir el módulo del recurso creado (p.e. `users.module`).

BASH

```
CREATE src/users/users.controller.spec.ts (566 bytes)
CREATE src/users/users.controller.ts (890 bytes)
CREATE src/users/users.module.ts (247 bytes)
CREATE src/users/users.service.spec.ts (453 bytes)
CREATE src/users/users.service.ts (609 bytes)
CREATE src/users/dto/create-user.dto.ts (30 bytes)
CREATE src/users/dto/update-user.dto.ts (169 bytes)
CREATE src/users/entities/user.entity.ts (21 bytes)
UPDATE src/app.module.ts (312 bytes)
```

Archivo users.controller.ts *creado*

```
import { Controller, Get, Post, Body, Put, Param, Delete } from '@nestjs/common';  TYPESCRIPT
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';

@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }

  @Get()
  findAll() {
    return this.usersService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(+id);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.usersService.update(+id, updateUserDto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.usersService.remove(+id);
  }
}
```

Archivo users.service.ts *creado*

TYPESCRIPT

```
import { Injectable } from '@nestjs/common';
import { CreateUserDto } from '../dto/create-user.dto';
import { UpdateUserDto } from '../dto/update-user.dto';

@Injectable()
export class UsersService {
  create(createUserDto: CreateUserDto) {
    return 'This action adds a new user';
  }

  findAll() {
    return `This action returns all users`;
  }

  findOne(id: number) {
    return `This action returns a ${id} user`;
  }

  update(id: number, updateUserDto: UpdateUserDto) {
    return `This action updates a ${id} user`;
  }

  remove(id: number) {
    return `This action removes a ${id} user`;
  }
}
```

Voilà!! A partir de los archivos generados ya se pueden adaptar los endpoints del controlador, crear el código de los servicios y adaptar las entidades y los DTOs al caso de base de datos del proyecto.



Más información sobre **CRUD generator** en la [documentación oficial](https://docs.nestjs.com/recipes/crud-generator)
(<https://docs.nestjs.com/recipes/crud-generator>)

Generador del código de la entidad

`typeorm-model-generator` es un paquete NodeJS para trabajar con TypeORM que genera los modelos a partir de las tablas existentes en una base de datos. Actualmente soporta los siguientes DBMS:

- Microsoft SQL Server
- PostgreSQL
- MySQL

- MariaDB
- Oracle Database
- SQLite



`typeorm-model-generator` incorpora el driver para todas las bases de datos soportadas excepto para Oracle. Para Oracle se debe tener instalado Oracle Instant Client en el equipo donde se vaya a ejecutar `typeorm-model-generator`.

`typeorm-model-generator` se podrá invocar directamente mediante `npx` (<https://www.npmjs.com/package/npx>).



`npx` es una herramienta que permite ejecutar paquetes de binarios `npm`. `npx` queda instalado en versiones posteriores a `npm 5.2`.

A continuación se muestran unos parámetros habituales que utilizaremos al generar los modelos con `typeorm-model-generator`

- `-h` : Host de la base de datos
- `-d` : Nombre de la base de datos
- `-u` : Usuario
- `-x` : Contraseña
- `-e` : DBMS (p.e. `mysql`, `oracle`, `mssql`, `pgsql`, ...)
- `-o` : (opcional) Ruta en la que guardar los archivos generados
- `-p` : (opcional) Puerto

A continuación se muestra un ejemplo de uso en Oracle

```
$ npx typeorm-model-generator -h localhost -d myDatabase -u myUser -x myPassword -e oracleBASH  
-o ./reservations -p 1527
```

Esto generará un archivo de entidad para cada tabla encontrada en la base de datos indicada incluyendo la definición de cada uno de los campos de la tabla.



Usa los archivos generados para adaptar el contenido de los archivos de entidades generados con *CRUD generator* del apartado anterior Generador de archivos para una entidad.

Los archivos generados por `typeorm-model-generator` también se pueden usar para personalizar los DTO generados por *CRUD generator*. Para adaptar los DTOs normalmente quitaremos algunos de los campos de los modelos creados por *CRUD generator*.

A continuación se muestra un ejemplo de archivo creado por ``

```
import { Column, Entity, Index, JoinColumn,ManyToOne } from "typeorm";                                TYPESCRIPT
import { RstCalendarios } from "../RstCalendarios";

@Index("RST_DIAS_PK", ["yDia"], { unique: true })
@Entity("RST_DIAS") 1
export class RstDias {
  @Column("varchar2", { name: "T_OBSERVACIONES", nullable: true, length: 100 }) 2
    tObservaciones: string | null;

  @Column("varchar2", { name: "L_RESERVABLE", nullable: true, length: 1 })
    lReservable: string | null;

  @Column("date", { name: "F_CALEDARIO", nullable: true })
    fCalendario: Date | null;

  @Column("number", { primary: true, name: "Y_DIA" })
    yDia: number;

  @ManyToOne(() => RstCalendarios, (rstCalendarios) => rstCalendarios.rstDias) 3
  @JoinColumn([{ name: "Y_CALEDARIO", referencedColumnName: "yCalendario" }])
    yCalendario: RstCalendarios;
}
```

- 1 Entidad con el nombre de la tabla con la que se corresponde
- 2 Definición de cada una de las columnas con sus tipos de datos, restricciones, ...
- 3 Anotaciones para relaciones

Last updated 2021-01-27 00:42:50 CET