29 DE NOVIEMBRE DE 2025

API RESFUL

Taller01

KEVIN SEBASTIAN LECHON CHURUCHUMBI
UNIVERSIDAD DE LAS FUERZAS ARMADAS “ESPE”
Sangolquí - Ecuador

Reporte Ejecutivo Técnico – API Book

1. Descripción general del sistema

El sistema consiste en una API RESTful desarrollada en Java 17 con Spring Boot, que gestiona la entidad Book. Permite realizar operaciones de CRUD completo: crear, listar, buscar por ID, actualizar y eliminar libros.

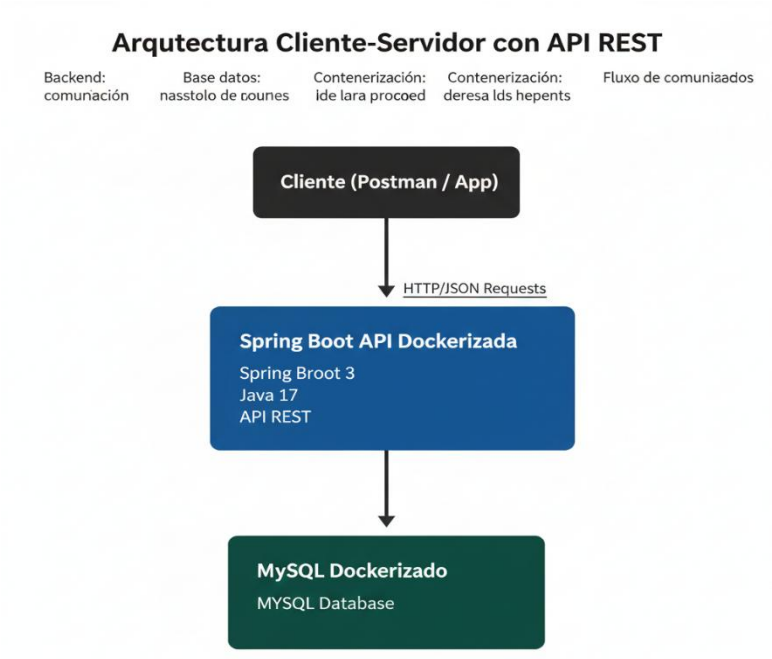
La API está conectada a una base de datos MySQL que corre dentro de un contenedor Docker, garantizando portabilidad y facilidad de despliegue.

El propósito principal es ofrecer un servicio backend que pueda integrarse con cualquier aplicación cliente, ya sea web o móvil.

2. Arquitectura utilizada

- Arquitectura: Cliente – Servidor con API REST.
- Backend: Spring Boot 3, Java 17.
- Base de datos: MySQL en contenedor Docker.
- Contenerización: Docker para la API y la base de datos, sin usar Docker Compose.
- Flujo de comunicación: Los clientes hacen peticiones HTTP (GET, POST, PUT, DELETE) y la API responde en formato JSON.

Diagrama conceptual:



3. Diseño REST aplicado

Endpoints principales implementados:

MÉTODO	URL	DESCRIPCIÓN	REQUEST BODY
GET	/books	Listar todos los libros	-
GET	/books/{id}	Buscar libro por ID	-
POST	/books	Crear un libro	{"titulo":"...", "autor":"...", "genero":"..."}
PUT	/books/{id}	Actualizar un libro	{"titulo":"...", "autor":"...", "genero":"..."}
DELETE	/books/{id}	Eliminar un libro	-

Formato de respuesta: JSON, ejemplo:

```
[
  {"id":1,"titulo":"Duende","autor":"Kevin Lechón","genero":"terror"},
  {"id":2,"titulo":"Duende2","autor":"Bryan","genero":"terror"}
]
```

4. Código relevante y explicaciones

Book.java (Entidad):

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String titulo;
    private String autor;
    private String genero;
}
```

Explicación:

- `@Entity`: Marca la clase como una entidad JPA que se mapea a una tabla en la base de datos.
- `@Id`: Define el atributo `id` como clave primaria de la tabla.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: Indica que la base de datos generará automáticamente los valores del ID (auto-incremento).
- Los atributos `titulo`, `autor` y `genero` representan las columnas de la tabla y se almacenan como datos de cada libro.

Esta clase representa la **estructura de los datos** que manejamos en la AP

BookRepository.java:

```
public interface BookRepository extends JpaRepository<Book, Long> {}
```

Explicación:

- `JpaRepository<Book, Long>`: Hereda métodos de JPA para operaciones CRUD básicas sin necesidad de implementarlas manualmente.
- `Book`: Tipo de entidad que maneja el repositorio.
- `Long`: Tipo del ID de la entidad.

Ventaja: Gracias a JPA, podemos usar métodos como `save()`, `findAll()`, `findById()`, `delete()` directamente, simplificando la capa de acceso a datos.

BookController.java (CRUD):

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private BookRepository repository;

    @GetMapping
    public List<Book> getAll() { return repository.findAll(); }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getById(@PathVariable Long id) {
        return repository.findById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public Book create(@RequestBody Book book) { return
repository.save(book); }

    @PutMapping("/{id}")
    public ResponseEntity<Book> update(@PathVariable Long id,
@RequestBody Book book) {
        return repository.findById(id)
            .map(b -> {
                b.setTitulo(book.getTitulo());
                b.setAutor(book.getAutor());
                b.setGenero(book.getGenero());
                return ResponseEntity.ok(repository.save(b));
            }).orElse(ResponseEntity.notFound().build());
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        return repository.findById(id)
            .map(b -> { repository.delete(b); return
ResponseEntity.noContent().build(); })
            .orElse(ResponseEntity.notFound().build());
    }
}
```

Explicación:

- `@RestController`: Indica que la clase manejará **peticiones HTTP** y retornará **JSON**.
- `@RequestMapping("/books")`: Define la ruta base de la API.
- `@Autowired`: Inyecta automáticamente el repositorio para acceder a la base de datos.

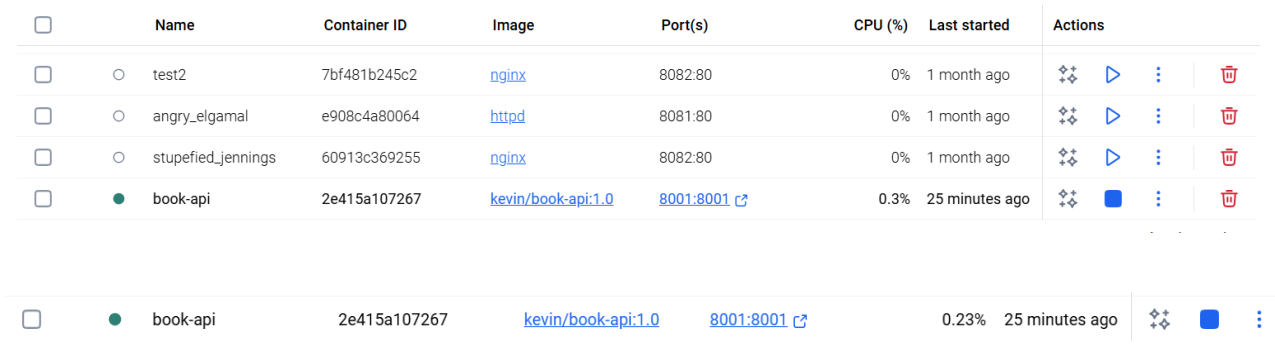
Endpoints:

1. `GET /books` → Lista todos los libros (`repository.findAll()`).
2. `GET /books/{id}` → Busca un libro por ID, devuelve `404 Not Found` si no existe.
3. `POST /books` → Crea un nuevo libro con los datos del cuerpo de la petición.
4. `PUT /books/{id}` → Actualiza un libro existente; primero verifica si existe.
5. `DELETE /books/{id}` → Elimina un libro por ID; retorna `204 No Content` si se elimina correctamente.

Ventaja: Cada endpoint maneja tanto el flujo exitoso como los errores (libro no encontrado), cumpliendo principios RESTful.

6. Evidencias de Docker

Le desplegamos nuestro proyecto en el Docker

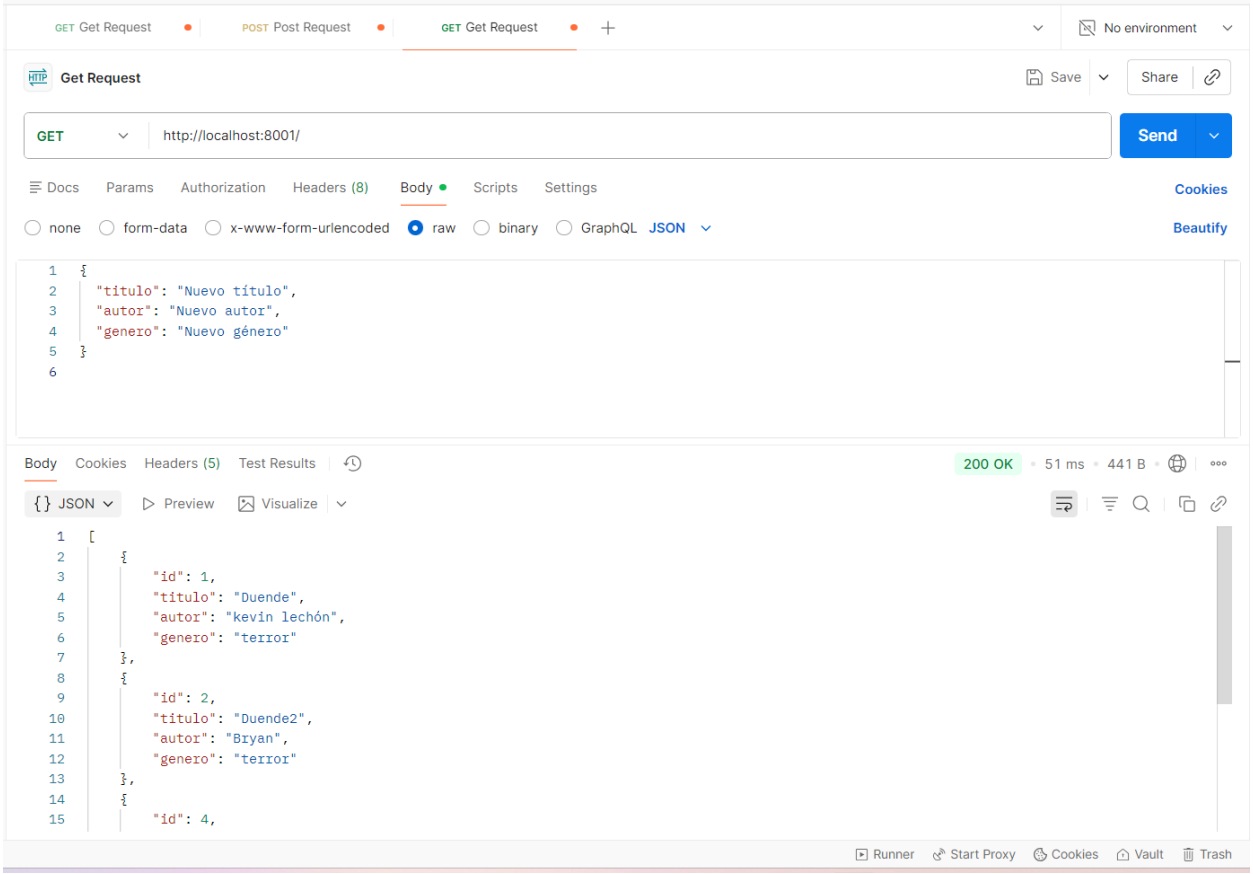


Mandamos Docker ps para ver que los dos contenedores esten alzados

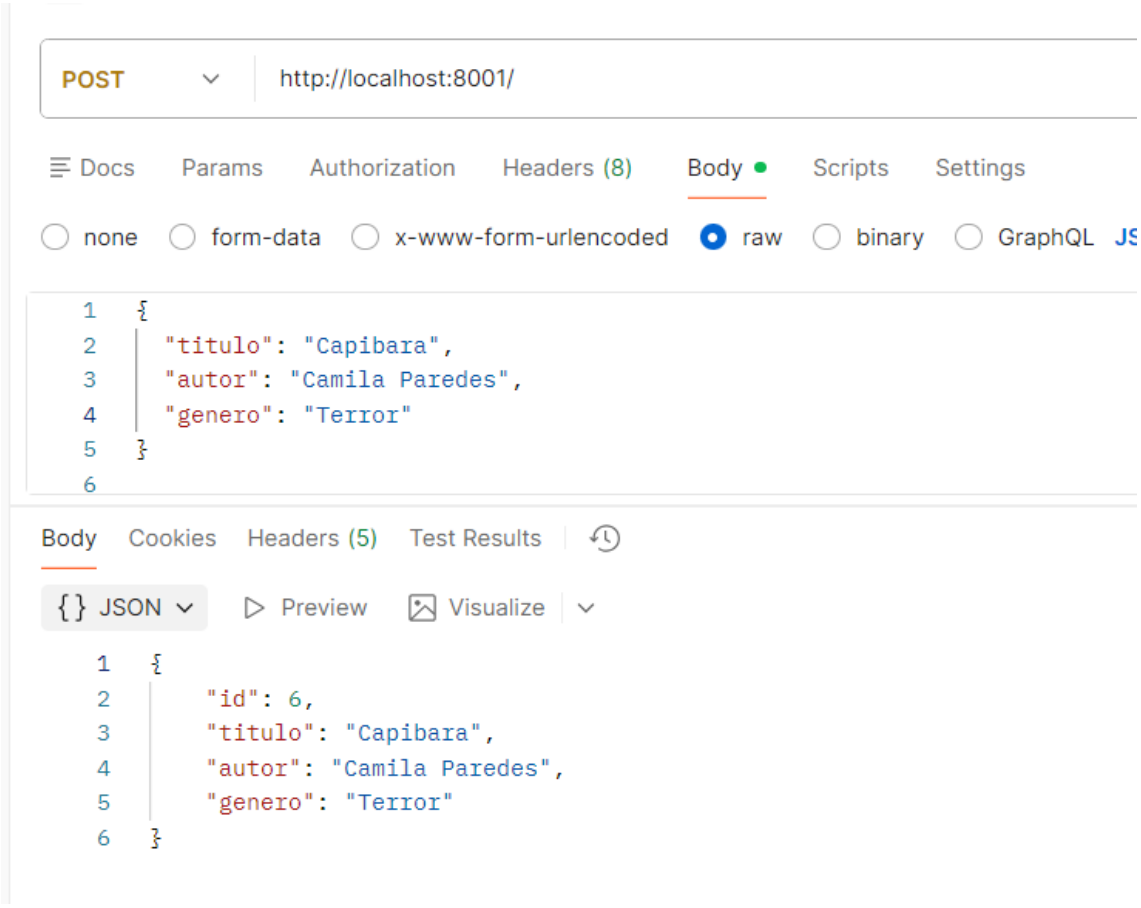
```
C:\Users\Usuario>docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED          STATUS              PORTS
                     NAMES
2e415a107267   kevin/book-api:1.0   "java -jar app.jar"      26 minutes ago   Up 26 minutes       0.0.0.0:8001->8001/tcp,
 [::]:8001->8001/tcp   book-api
aaa848d98501   mysql:8.0            "docker-entrypoint.s…"   3 days ago       Up About an hour    0.0.0.0:3307->3306/tcp,
 [::]:3307->3306/tcp   mysql-sisdb2025
```

6. **Evidencias de pruebas con Postman**

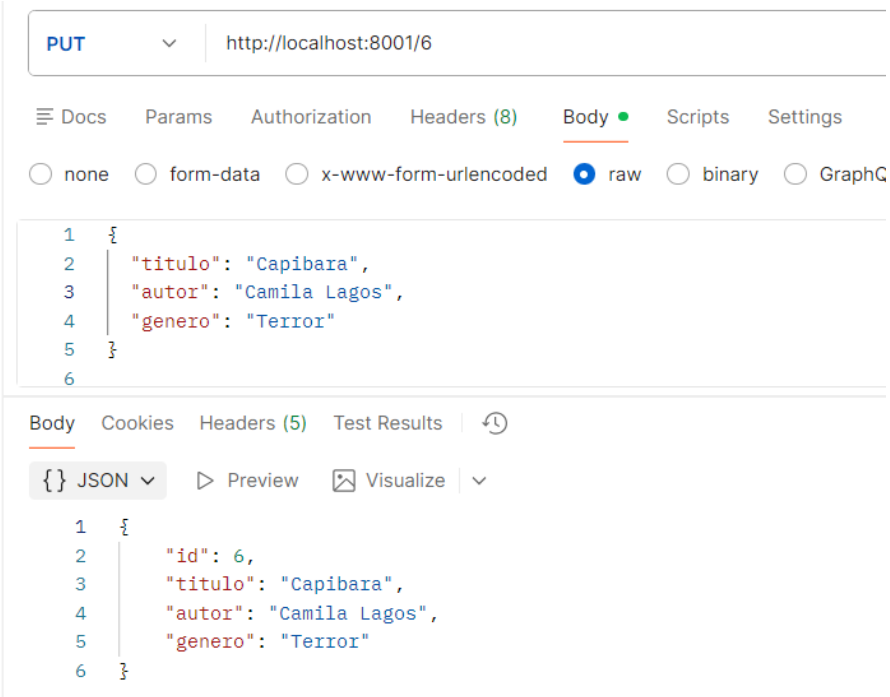
- **GET /books:** Devuelve todos los libros.



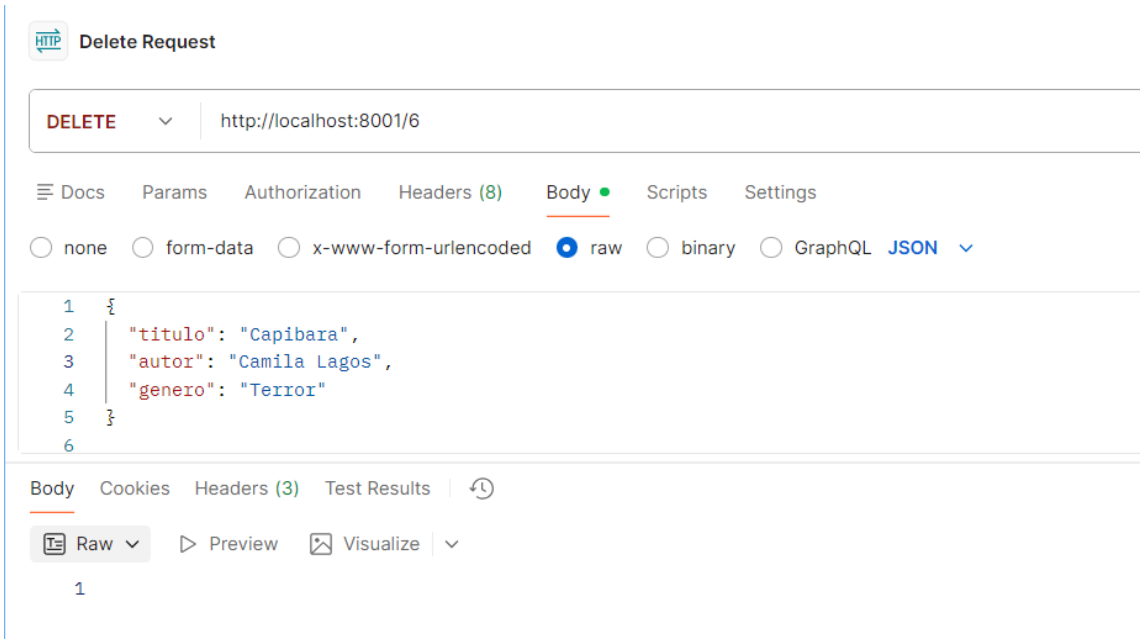
- **POST /books:** Inserta un nuevo libro.



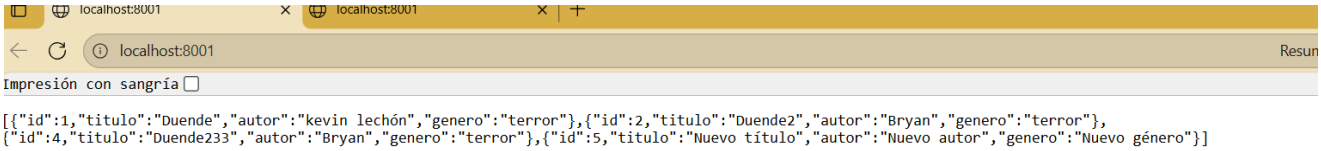
- **PUT /books/{id}:** Actualiza libro existente.



- **DELETE /books/{id}:** Elimina libro por ID.



En este apartado podemos ver nuestra Api en web



7. Pasos para ejecutar la aplicación

1. **Levantar MySQL:**

docker run -d -p 3307:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=abcd -e MYSQL_DATABASE=sisdb2025 mysql:8

2. **Construir el JAR:**

mvn clean package -DskipTests

3. **Construir imagen Docker de la API:**

docker build -t kevin/book-api:1.0 .

4. **Ejecutar contenedor de la API:**

docker run -d -p 8001:8001 --name book-api kevin/book-api:1.0

Conclusiones

- La API RESTful para la entidad Book permite crear, listar, actualizar, buscar por ID y eliminar registros, cumpliendo con los principios de diseño REST y garantizando la gestión completa de los datos.
- La aplicación Spring Boot se pudo contenerizar con Docker, lo que asegura portabilidad y facilidad de despliegue en distintos entornos sin depender de configuraciones locales.
- La conexión con MySQL en contenedor Docker funcionó correctamente, y las pruebas mediante Postman permitieron verificar tanto casos exitosos como errores, asegurando la confiabilidad de la API.

Recomendaciones

- Implementar variables de entorno o un archivo de configuración seguro para las credenciales de la base de datos, evitando exponer información sensible en el código.
- Usar Swagger o OpenAPI para documentar los endpoints y versionar la API, facilitando el mantenimiento y la integración con otros sistemas.
- Incorporar pruebas unitarias y de integración automatizadas, así como herramientas de monitoreo para detectar errores o caídas del servicio en tiempo real.