

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.

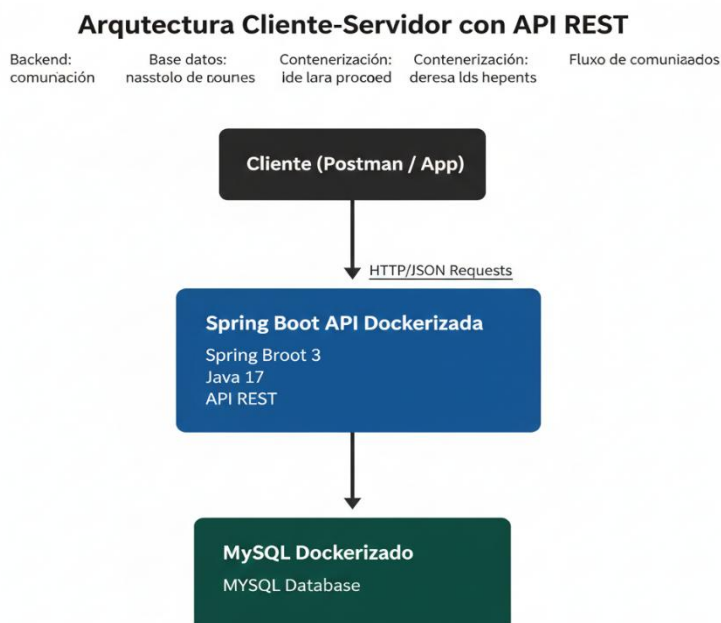
La API está conectada a una base de datos MySQL que corre dentro de un contenedor Docker, garantizando portabilidad y facilidad de despliegue.

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

```

=> => transferring context: 2B
=> [1/3] FROM docker.io/library/eclipse-temurin:17-jdk-alpine@sha256:eaf56b7430cee6c93871106367715e2675192093d8f67dbbde07136f7cfae60
=> => resolve docker.io/library/eclipse-temurin:17-jdk-alpine@sha256:eaf56b7430cee6c93871106367715e2675192093d8f67dbbde07136f7cfae60
=> [internal] load build context
=> => transferring context: 53.21MB
=> CACHED [2/3] WORKDIR /app
=> [3/3] COPY target/test-0.0.1-SNAPSHOT.jar app.jar
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:326b3a8bdeb806710140205fcc794c6b59b45b94c0c7c535df082adb78ede46a
=> => exporting config sha256:e8eac18ac1a2deffbfbcfd2111285d617cd60d8b3c4b346ff3c1de0dd8f5535a
=> => exporting attestation manifest sha256:0b107fc504831a10a74b442203c4ffc2b7391fa8a75e83f83d7b04182b0fadd9
=> => exporting manifest list sha256:f214b839bba6783b5ee39ef08cbafd149cb66e44f913e0654a11f73145073c15
=> => naming to docker.io/kevin/book-api:1.0
=> => unpacking to docker.io/kevin/book-api:1.0

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/0ku3jek3pkttnqmcrodex739l
PS C:\Users\Usuario\Documents\Septimo\Distribuidas\SEGUNDO_P\test>
  
```

Le desplegamos nuestro proyecto en el Docker

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
<input type="checkbox"/>	○ test2	7bf481b245c2	nginx	8082:80	0%	1 month ago	
<input type="checkbox"/>	○ angry_elgamal	e908c4a80064	httpd	8081:80	0%	1 month ago	
<input type="checkbox"/>	○ stupefied_jennings	60913c369255	nginx	8082:80	0%	1 month ago	
<input type="checkbox"/>	● book-api	2e415a107267	kevin/book-api:1.0	8001:8001	0.3%	25 minutes ago	

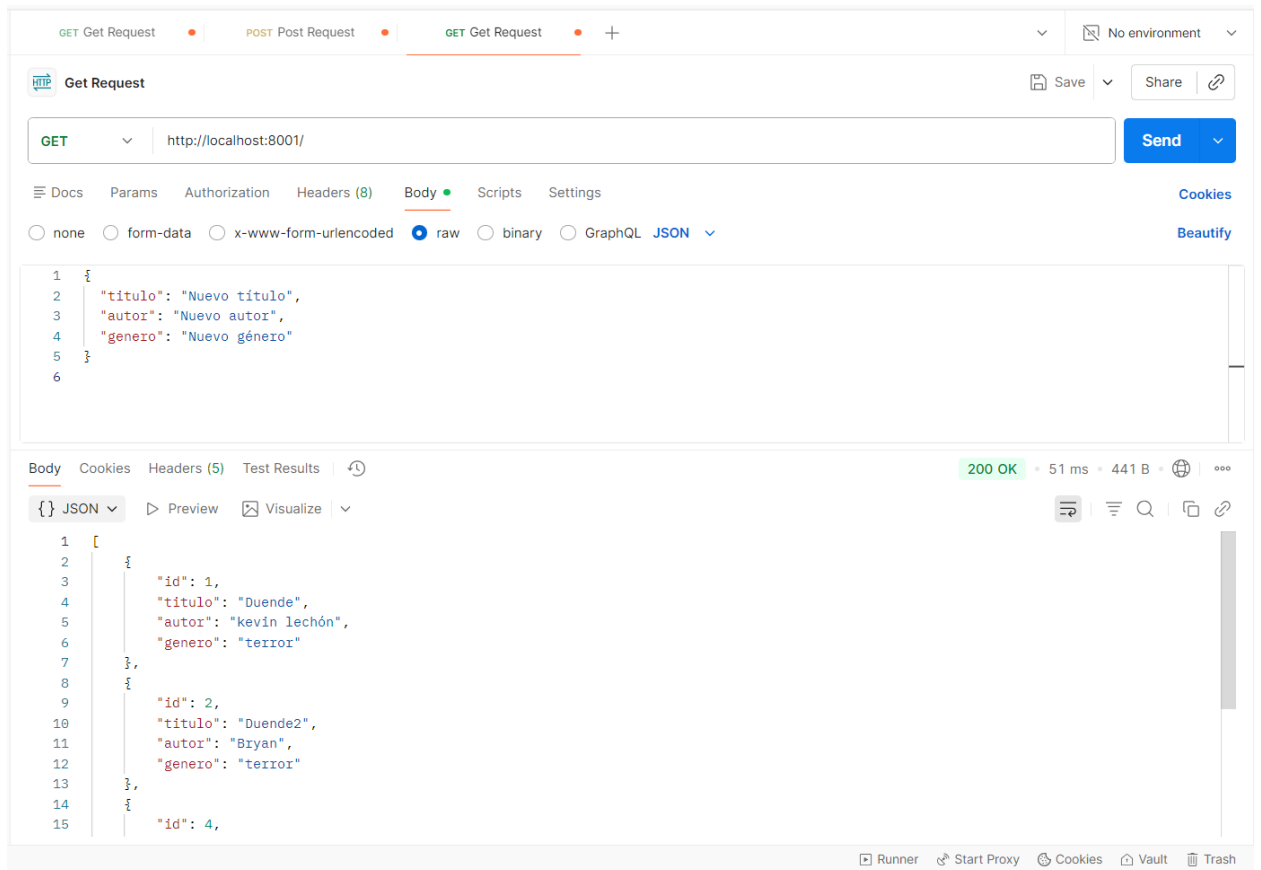
<input type="checkbox"/>	● book-api	2e415a107267	kevin/book-api:1.0	8001:8001	0.23%	25 minutes ago	
--------------------------	------------	--------------	------------------------------------	---------------------------	-------	----------------	--

Mandamos Docker ps para ver que los dos contenedores esten alzados

```
C:\Users\Usuario>docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
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.



The screenshot shows the Postman interface with a GET request to `http://localhost:8001/`. The response is a JSON array of book objects, each with an `id`, `titulo`, `autor`, and `genero`.

```
1 {
2   "titulo": "Nuevo título",
3   "autor": "Nuevo autor",
4   "genero": "Nuevo género"
5 }
6
```

The response status is **200 OK** with a response time of 51 ms and a body size of 441 B. The response body is displayed in JSON format:

```
1 [
2   {
3     "id": 1,
4     "titulo": "Duende",
5     "autor": "kevin lechón",
6     "genero": "terror"
7   },
8   {
9     "id": 2,
10    "titulo": "Duende2",
11    "autor": "Bryan",
12    "genero": "terror"
13  },
14  {
15    "id": 4,
```

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

POST http://localhost:8001/

Docs Params Authorization Headers (8) **Body** Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JS

```
1 {
2   "titulo": "Capibara",
3   "autor": "Camila Paredes",
4   "genero": "Terror"
5 }
6
```

Body Cookies Headers (5) Test Results ↺

{ } JSON ▾ ▶ Preview 🖼 Visualize ▾

```
1 {
2   "id": 6,
3   "titulo": "Capibara",
4   "autor": "Camila Paredes",
5   "genero": "Terror"
6 }
```

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

PUT http://localhost:8001/6

Docs Params Authorization Headers (8) **Body** Scripts Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL

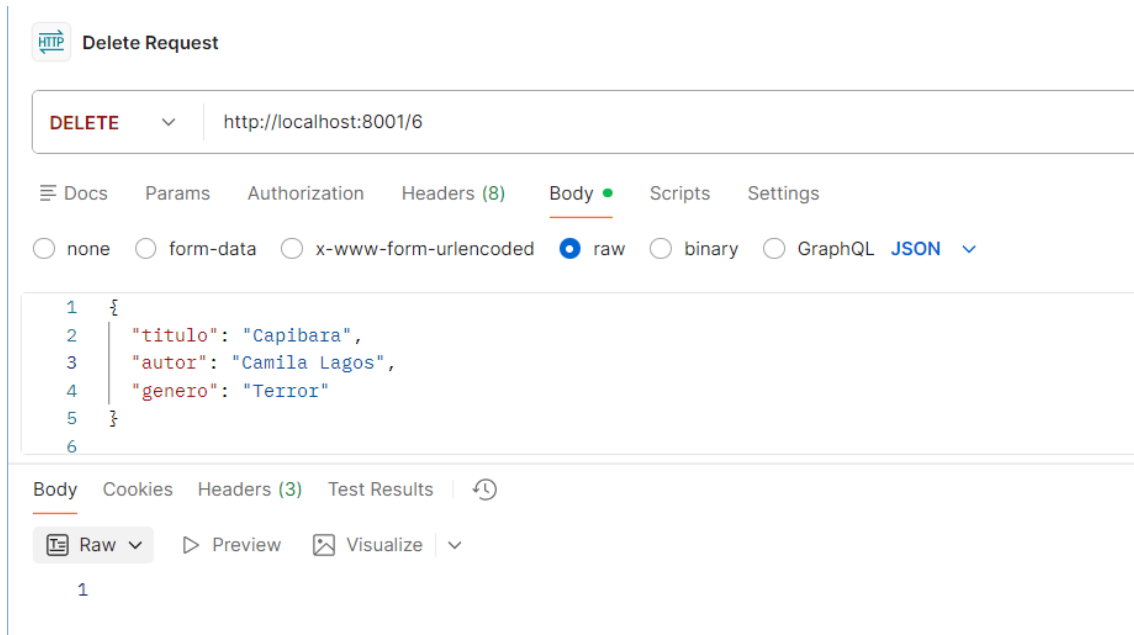
```
1 {
2   "titulo": "Capibara",
3   "autor": "Camila Lagos",
4   "genero": "Terror"
5 }
6
```

Body Cookies Headers (5) Test Results ↺

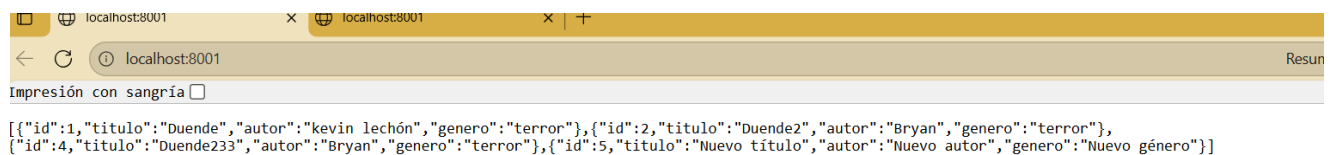
{ } JSON ▾ ▶ Preview 🖼 Visualize ▾

```
1 {
2   "id": 6,
3   "titulo": "Capibara",
4   "autor": "Camila Lagos",
5   "genero": "Terror"
6 }
```

- **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.