Laboratorio 4: Desarrollo de una API REST Evolutiva para la Gestión de Blueprints

Juan Miguel Rojas Chaparro, Cristian David Silva Perilla, David Eduardo Salamanca, Felipe Eduardo Calvache
Escuela Colombiana de Ingeniería Julio Garavito
Bogotá, Colombia
Email: tu.correo@mail.escuelaing.edu.co

***Resumen*—Este documento detalla la implementación de una API REST robusta utilizando el framework Spring Boot 3.3.x y Java 21. Se describe la migración de un sistema de persistencia en memoria hacia PostgreSQL, la aplicación de filtros de procesamiento de planos (Redundancia y Undersampling) y la estandarización de respuestas bajo el protocolo HTTP, documentado mediante la especificación OpenAPI.**

***Index Terms*—API REST, Spring Boot, PostgreSQL, OpenAPI, Inyección de Dependencias, Arquitectura de Software.**

I. Introducción

La evolución de sistemas monolíticos a arquitecturas desacopladas requiere interfaces de comunicación estandarizadas. Este laboratorio se enfoca en la creación de servicios REST que gestionen planos (blueprints), asegurando la integridad de los datos y la escalabilidad mediante una arquitectura de capas bien definida.

II. Desarrollo de Actividades

II-A. Familiarización con el Código Base

Inicialmente se analizó la estructura del proyecto base, el cual se encontraba organizado bajo una arquitectura por capas. Se identificaron los siguientes componentes:

- **Model Layer:** Contiene las entidades de dominio `Blueprint` y `Point`, que representan un plano y sus coordenadas respectivamente.
- **Persistence Layer:** Define la interfaz `BlueprintPersistence` y su implementación inicial en memoria mediante `InMemoryBlueprintPersistence`.
- **Service Layer:** Orquesta la lógica de negocio y aplica dinámicamente los filtros configurados.
- **Controller Layer:** Expone los endpoints REST bajo el esquema de versionamiento `/api/v1/blueprints`.

Esta separación permitió garantizar bajo acoplamiento y alta cohesión entre componentes.

III. Migración a Persistencia Relacional con PostgreSQL

Inicialmente, el sistema utilizaba una implementación en memoria (`InMemoryBlueprintPersistence`) basada en estructuras como `ConcurrentHashMap`. Esto implicaba que:

- Los datos se almacenaban únicamente en RAM.
- La información se perdía al reiniciar la aplicación.
- Los datos iniciales eran estáticos y estaban definidos manualmente.

Esta aproximación es adecuada para pruebas iniciales, pero no garantiza durabilidad ni persistencia real en entornos productivos.

III-A. Despliegue de PostgreSQL con Docker

Para garantizar persistencia, se desplegó un contenedor PostgreSQL utilizando Docker Desktop. Esto permitió aislar la base de datos del sistema operativo anfitrión y mantener los datos incluso después de reiniciar la aplicación.

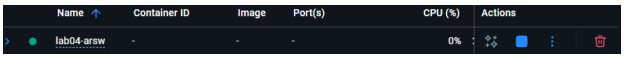

Figura 1: Creación y ejecución del contenedor PostgreSQL en Docker Desktop.

III-B. Orquestación de Infraestructura con Docker Compose

En lugar de un despliegue manual, se utilizó **Docker Compose** para definir y gestionar el entorno de base de datos de forma declarativa. Esta aproximación garantiza que cualquier desarrollador pueda replicar el entorno exacto de persistencia ejecutando un único comando.

El archivo `docker-compose.yml` (ver Listado 2) configura un servicio de PostgreSQL versión 16, exponiendo el puerto estándar 5432 y definiendo variables de entorno para la creación automática de la base de datos y credenciales de acceso.

```
version: '3.8'
services:
  db:
    image: postgres:16
    container_name: blueprints-db
    environment:
      POSTGRES_DB: blueprints
      POSTGRES_USER: blueprints_user
      POSTGRES_PASSWORD: blueprints_pass
    ports:
      - "5432:5432"
    volumes:
      - ./postgres-data:/var/lib/postgresql/data
```

Listing 1: Configuración de Docker Compose para el entorno de persistencia.

La integración se potenció mediante el módulo `spring-boot-docker-compose`, el cual permite que Spring Boot reconozca el archivo al iniciar la aplicación,

gestionando automáticamente la conexión sin necesidad de configurar manualmente el JDBC URL en entornos de desarrollo.

III-C. *Definición del Esquema Relacional*

Se creó el archivo `schema.sql` donde se definieron las tablas:

- `blueprints`
- `blueprint_points`

Se estableció una relación 1:N mediante clave foránea, permitiendo que un blueprint tenga múltiples puntos ordenados por el campo `idx`.

Adicionalmente, se creó el archivo `data.sql` para permitir la carga inicial opcional de datos de prueba, facilitando validaciones durante el desarrollo.

III-D. *Implementación de PostgresBlueprintPersistence*

Se desarrolló la clase:
`/impl/PostgresBlueprintPersistence.java`
Esta clase implementa la interfaz `BlueprintPersistence`, manteniendo el contrato original y respetando el principio de inversión de dependencias.

Se configuró la conexión en `application.properties`:

```
spring.mvc.pathmatch.matching-strategy=ant_path_matcher

spring.datasource.url=jdbc:postgresql://localhost:5432/
    blueprints
spring.datasource.username=blueprints_user
spring.datasource.password=blueprints_pass
spring.datasource.driver-class-name=org.postgresql.Driver
spring.sql.init.mode=always

springdoc.swagger-ui.path=/swagger-ui.html
springdoc.api-docs.path=/v3/api-docs
```

En la siguiente imagen se valida la creación correcta de las tablas utilizando el comando `\dt`, evidenciando que ambas existen y están inicialmente vacías.

```
blueprints=# \dt
                List of relations
 Schema |       Name       | Type  |      Owner
--------+------------------+-------+-----------------
 public | blueprint_points | table | blueprints_user
 public | blueprints       | table | blueprints_user
(2 rows)

blueprints=# SELECT * FROM blueprints;
 author | name
--------+------
(0 rows)

blueprints=# SELECT * FROM blueprint_points;
 id | author | name | idx | x | y
----+--------+------+-----+---+---
(0 rows)
```

Figura 2: Validación de tablas creadas en PostgreSQL `\dt`.

—

IV. Buenas Prácticas de API REST

IV-A. *Versionamiento de Endpoints*

Se modificó el `@RequestMapping` base del controlador a:

`/api/v1/blueprints`

Esto permite versionar la API y mantener compatibilidad futura.

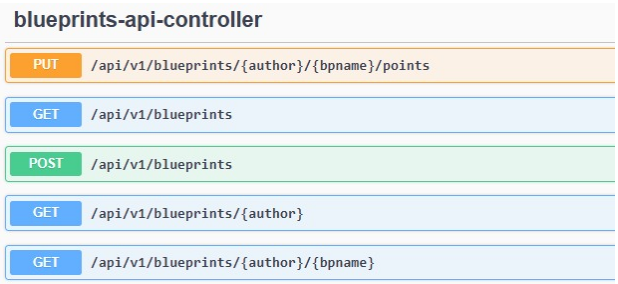

Figura 3: Swagger mostrando la ruta versionada `/api/v1/blueprints`.

IV-B. *Uso Correcto de Códigos HTTP*

Se estandarizaron los siguientes códigos:

- 200 OK para consultas exitosas.
- 201 Created para creación de recursos.
- 202 Accepted para actualizaciones.
- 400 Bad Request para solicitudes inválidas.
- 404 Not Found para recursos inexistentes.

IV-C. *Respuesta Uniforme*

Se implementó la clase:
`/controllers/dto/ApiResponse.java`
junto con `NewBlueprintRequest.java`, permitiendo encapsular todas las respuestas bajo un formato uniforme:

```
public record ApiResponse<T>(
    int code,
    String message,
    T data
) {}
```

—

V. Validación Funcional mediante Pruebas con cURL

V-A. *Base Inicialmente Vacía*

Al iniciar la aplicación conectada a PostgreSQL, la base de datos no contiene registros.

```
felip@WawaDeFelipe MINGW64 ~/OneDrive/ARSW/Lab04-ARSW (feature/dataBase)
$ curl http://localhost:8080/blueprints
[]
```

Figura 4: Consulta inicial mostrando base de datos vacía.

V-B. Creación de Blueprint

Se ejecutó una petición POST que retornó código 201 Created. Posteriormente, al realizar un GET, se evidenció que el registro fue almacenado correctamente.

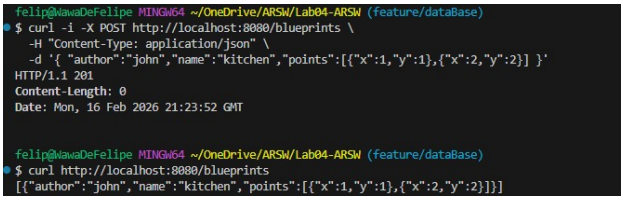

Figura 5: Inserción de blueprint y posterior verificación mediante GET.

V-C. Actualización de Blueprint

Se agregó un nuevo punto mediante PUT, retornando 202 Accepted.

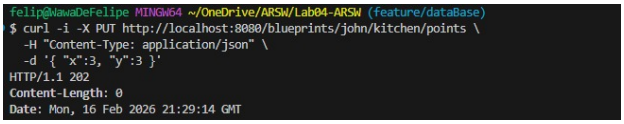

Figura 6: Actualización del blueprint agregando un nuevo punto.

V-D. Verificación Directa en Base de Datos

Se accedió al contenedor PostgreSQL mediante `docker exec` para verificar que los registros fueron efectivamente persistidos.

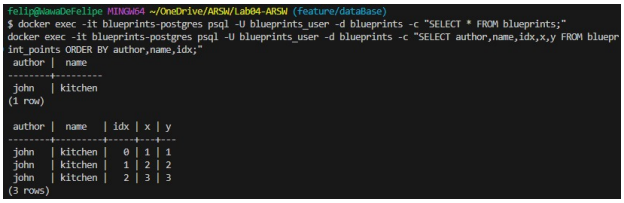

Figura 7: Validación directa en PostgreSQL tras inserción y actualización.

V-E. Validación de Error 404

Se realizó una consulta sobre un recurso inexistente, retornando correctamente 404 Not Found.

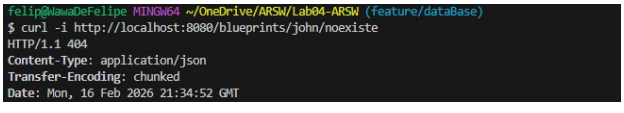

Figura 8: Prueba de recurso inexistente retornando código 404.

V-F. Pruebas Integrales de Endpoints

Finalmente, se validaron todos los endpoints comprobando la correcta correspondencia entre operación y código HTTP.

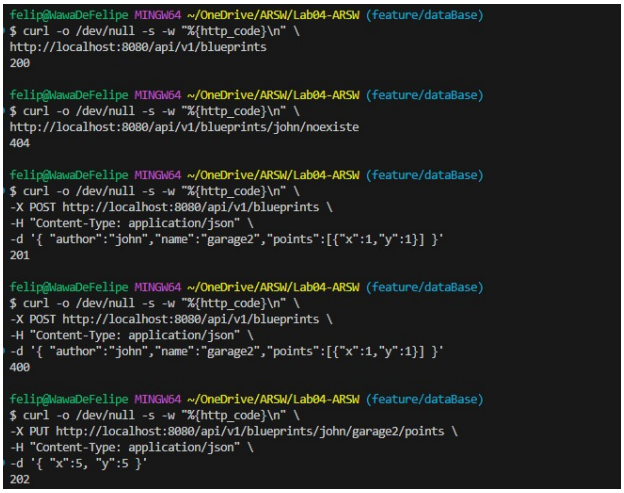

Figura 9: Pruebas completas de los endpoints y sus respectivos códigos HTTP.

VI. OpenAPI / Swagger

Con el objetivo de mejorar la mantenibilidad y comprensión de la API, se integró la especificación OpenAPI mediante la dependencia:

```
springdoc-openapi-starter-webmvc-ui
```

Esta integración permitió generar automáticamente la documentación del servicio REST a partir del código fuente, exponiendo dos recursos principales:

- `/v3/api-docs`: Representa el contrato formal de la API en formato JSON bajo el estándar OpenAPI 3.0.
- `/swagger-ui.html`: Interfaz gráfica interactiva que permite explorar y probar los endpoints sin herramientas externas.

La documentación generada describe de manera estructurada:

- Rutas disponibles.
- Métodos HTTP soportados.
- Parámetros de entrada.
- Esquemas de datos.
- Códigos de respuesta.

VI-A. Uso de Anotaciones OpenAPI

Para enriquecer la documentación automática, se emplearon las anotaciones:

- `@Operation`
- `@ApiResponses`
- `@Tag`

La anotación `@Operation` permite definir un resumen y una descripción detallada de cada endpoint, facilitando la comprensión funcional del recurso expuesto. Esto mejora la claridad para desarrolladores externos que consuman la API.

Por su parte, `@ApiResponses` permite documentar explícitamente los códigos HTTP que puede retornar cada operación, incluyendo escenarios exitosos y de error. Esto

garantiza coherencia entre la implementación real y el contrato publicado, evitando ambigüedades en la integración con clientes.

Finalmente, `@Tag` permite agrupar los endpoints bajo una categoría lógica dentro de Swagger UI, mejorando la organización visual de la documentación.

VI-B. Importancia Arquitectónica

La incorporación de OpenAPI no solo mejora la experiencia de desarrollo, sino que fortalece la arquitectura del sistema al establecer un contrato formal entre productor y consumidor del servicio. Esto facilita:

- Integración con clientes frontend.
- Generación automática de SDKs.
- Validación de cumplimiento REST.
- Evolución controlada del API.

De esta manera, la documentación deja de ser un elemento externo y se convierte en parte integral del ciclo de vida del software.

VI-C. Filtros de Procesamiento de Blueprints

Se implementaron filtros dinámicos aplicables a los planos antes de su retorno:

- **RedundancyFilter:** Elimina puntos consecutivos duplicados.
- **UndersamplingFilter:** Conserva uno de cada dos puntos.

La activación de los filtros se realizó mediante perfiles de Spring, permitiendo modificar el comportamiento del sistema sin alterar la lógica central.

Este enfoque demuestra extensibilidad y configuración basada en entorno.

VII. Resultados

La API resultante presenta las siguientes características:

- Persistencia real en base de datos relacional.
- Arquitectura desacoplada y mantenible.
- Versionamiento formal de endpoints.
- Documentación automática alineada con el estándar OpenAPI 3.0.
- Respuestas uniformes bajo un esquema tipado.

El sistema demuestra una transición exitosa desde una implementación en memoria hacia una solución preparada para entornos reales de producción.

VIII. Análisis de Filtros Implementados

Se evaluaron dos estrategias de filtrado para los puntos de los planos:

1. **RedundancyFilter:** Optimiza el ancho de banda eliminando puntos adyacentes idénticos.
2. **UndersamplingFilter:** Útil para previsualizaciones rápidas, reduciendo la densidad de puntos al 50 %.

IX. Resultados y Evidencias

IX-A. Validación de Contenedores

Se verificó la correcta orquestación de los servicios mediante la inspección de logs en Docker Desktop y comandos de terminal. Como se observa en la Figura 10, el contenedor `blueprints-db` se encuentra en estado *running* y aceptando conexiones en el puerto 5432, lo que permitió la ejecución exitosa de los scripts `schema.sql` y `data.sql` durante el arranque.

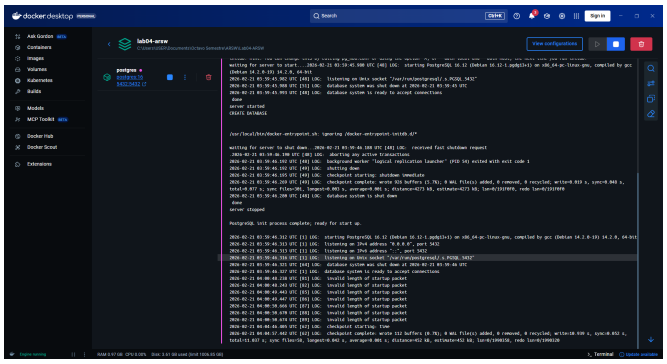

Figura 10: Estado del contenedor PostgreSQL gestionado por Docker Compose.

X. Conclusiones

1) La inversión de control (IoC) permitió migrar la base de datos sin modificar una sola línea del controlador. 2) El uso de códigos de estado HTTP (201, 202, 404) es vital para la semántica de la API. 3) OpenAPI reduce drásticamente el tiempo de integración con el frontend.

Referencias

[1] Spring Boot Reference Documentation, v3.3.x, 2024.

[2] Fielding, R., .Architectural Styles and the Design of Network-based Software Architectures,"2000.