ESCUELA COLOMBIANA DE INGENIERÍA JULIO GARAVITO

UNIVERSIDAD

Software Architecture Laboratory

Laboratory No. 4
REST API – Blueprints

Blueprints Java 21 / Spring Boot 3.3.x

Students:

Andersson David Sánchez Méndez

Cristian Santiago Pedraza Rodríguez

Elizabeth Correa Suárez

Juan Sebastian Ortega Muñoz

Instructor: Professor Javier Ivan Toquica Barrera

Course: Software Architecture (ARSW)

Institution: Escuela Colombiana de Ingeniería Julio Garavito

February 2026

Contents

1 Objective

This laboratory focuses on designing and implementing a production-quality REST API using Java 21 and Spring Boot 3.3.x, following layered architecture principles. The *Blueprints* system manages geometric drawings identified by author and name, each composed of a list of 2D points.

The specific goals include:

- **Layered Architecture**: Model, Persistence, Service, and Controller separation.
- **PostgreSQL Migration**: Replace in-memory storage with a real relational database via JPA/Hibernate.
- **REST Best Practices**: Versioned paths (`/api/v1/`), correct HTTP codes, and uniform `ApiResponse<T>` wrappers.
- **OpenAPI/Swagger**: Auto-generated, annotated interactive documentation.
- **Configurable Filters**: Strategy pattern via Spring profiles (redundancy, undersampling).
- **Bonus**: Container image via `spring-boot:build-image` and production metrics with Spring Actuator.

2 Application Architecture

2.1 Package Structure

The project follows a strict **logical layers** pattern, ensuring that each layer depends only on the one below it:

Package	Responsibility
`model`	Domain entities: Blueprint, Point
`persistence`	Interface + InMemory/Postgres impls
`persistence.entity`	JPA entities: BlueprintEntity, PointEntity
`persistence.jpa`	Spring Data JPA repository
`services`	Business logic, filter orchestration
`filters`	Strategy: Identity, Redundancy, Undersampling
`controllers`	REST endpoints, DTO, ApiResponse
`config`	OpenAPI/Swagger configuration

Table 1: Project package structure

2.2 Request / Response Flow

Every API call traverses the following pipeline:

1. **Controller** validates the HTTP request and maps it to a service call.
2. **Service** applies the active filter and delegates to persistence.
3. **Persistence** reads or writes the data store (in-memory or PostgreSQL).
4. The result is wrapped in `ApiResponse<T>` and returned with the appropriate HTTP status.

Key Design Decision

Filters are applied on **all read operations** (`getAll`, `getByAuthor`, `getByName`). Write operations (`save`, `addPoint`) store raw data without filtering.

3 Part 1: Code Base Familiarization

3.1 Model Layer

- **Blueprint**: Identified uniquely by `author + name`. Exposes an unmodifiable view of its point list. Equality and `hashCode` are based only on author and name.
- **Point**: An immutable Java `record` holding `(int x, int y)`.

Listing 1: Immutable Point record

```
public record Point(
    @Schema(description = "X coordinate") int x,
    @Schema(description = "Y coordinate") int y
) { }
```

3.2 Persistence Layer

`BlueprintPersistence` defines the storage contract:

Listing 2: Persistence interface contract

```
public interface BlueprintPersistence {
    void saveBlueprint(Blueprint bp)
        throws BlueprintPersistenceException;
    Blueprint getBlueprint(String author, String name)
        throws BlueprintNotFoundException;
    Set<Blueprint> getBlueprintsByAuthor(String author)
        throws BlueprintNotFoundException;
    Set<Blueprint> getAllBlueprints();
    void addPoint(String author, String name,
        int x, int y) throws BlueprintNotFoundException;
}
```

`InMemoryBlueprintPersistence` implements this interface using a `ConcurrentHashMap` with composite key `"author:name"`. Pre-loaded sample data:

Author	Name	Points
john	house	4
john	garage	3
jane	garden	3

Table 2: Sample data loaded on startup

3.3 Service Layer

`BlueprintsServices` acts as the orchestration layer. It injects both `BlueprintPersistence` and `BlueprintsFilter` via constructor injection, ensuring loose coupling and testability.

3.4 Controller Layer

`BlueprintsAPIController` maps all HTTP endpoints:

Method	Path	Code
GET	`/api/v1/blueprints`	200
GET	`.../{author}`	200/404
GET	`.../{author}/{name}`	200/404
POST	`/api/v1/blueprints`	201/400/403
PUT	`.../{author}/{name}/points`	202/404

Table 3: Controller endpoints and HTTP codes

4 Part 2: Migration to PostgreSQL

4.1 Design Decisions

The migration was implemented without modifying any existing class. Two key Spring annotations drive the behavior:

- `@Profile("postgres")`: The new implementation only loads when this profile is active.
- `@Primary`: When the profile is active, Spring selects `PostgresBlueprintPersistence` over `InMemoryBlueprintPersistence`.

Zero-Impact Migration

The full layered stack (Service, Controller, Filters) required **zero changes**. Only new classes were added in the persistence layer, honoring the Open/Closed Principle.

4.2 JPA Entity Model

Two JPA entities map to the relational schema:

Listing 3: BlueprintEntity – bidirectional mapping

```
@Entity
@Table(name = "blueprints",
  uniqueConstraints = @UniqueConstraint(
    columnNames = {"author","name"}))
public class BlueprintEntity {
    @OneToMany(mappedBy = "blueprint",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.EAGER)
    @OrderColumn(name = "position")
    private List<PointEntity> points = new ArrayList<>();
    ...
}
```

The `@OrderColumn` preserves point insertion order in the database, which is critical for filters like `UndersamplingFilter` that rely on positional indices.

4.3 Auto-Generated Relational Model

Table: blueprints		
`id`	BIGINT	PK, auto-increment
`author`	VARCHAR	NOT NULL
`name`	VARCHAR	NOT NULL
UNIQUE(author, name)		

Table: points		
`id`	BIGINT	PK, auto-increment
`x`	INT	NOT NULL
`y`	INT	NOT NULL
`position`	INT	Order in blueprint
`blueprint_id`	BIGINT	FK → blueprints(id)

Table 4: Auto-generated schema by Hibernate

4.4 Domain ↔ Entity Mapping

Listing 4: toDomain – entity to domain model

```
private Blueprint toDomain(BlueprintEntity e) {
    List<Point> pts = e.getPoints().stream()
        .map(p -> new Point(p.getX(), p.getY()))
        .collect(Collectors.toList());
    return new Blueprint(e.getAuthor(), e.getName(), pts);
}
```

Listing 5: toEntity – domain to JPA entity

```
private BlueprintEntity toEntity(Blueprint bp) {
    BlueprintEntity e = new BlueprintEntity(
        bp.getAuthor(), bp.getName());
    bp.getPoints().forEach(p ->
        e.addPoint(new PointEntity(p.x(), p.y())));
    return e;
}
```

4.5 Docker Setup

Listing 6: Starting PostgreSQL with Docker

```
docker run --name blueprints-db \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=blueprints \
  -p 5432:5432 -d postgres:16
```

Important Note

On Windows, port 5432 may already be in use by a local PostgreSQL installation. If so, use `-p 5433:5432` and update the datasource URL in `application-postgres.properties` accordingly.

4.6 Database Verification

After running with `-Dspring-boot.run.profiles=postgres`, Hibernate auto-creates the schema. The data can be verified directly:

Listing 7: Verifying persisted blueprints

```
SELECT b.author, b.name, p.x, p.y, p.position
FROM blueprints b
JOIN points p ON p.blueprint_id = b.id
ORDER BY b.author, b.name, p.position;
```

```
blueprints=# SELECT * FROM blueprints;
 id | author |  name
----+--------+--------
  1 | john   | house
  2 | john   | garage
  3 | jane   | garden
(3 rows)
```

Figure 1: PostgreSQL query confirming persisted blueprints

```
blueprints=# SELECT b.author, b.name, p.x, p.y, p.position
FROM blueprints b
JOIN points p ON p.blueprint_id = b.id
WHERE b.author = 'john' AND b.name = 'house'
ORDER BY p.position;
 author | name  | x  | y  | position
--------+-------+----+----+----------
 john   | house |  0 |  0 |        0
 john   | house | 10 |  0 |        1
 john   | house | 10 | 10 |        2
 john   | house |  0 | 10 |        3
(4 rows)
```

Figure 2: Blueprint and its points joined from both tables

5 Part 3: REST API Best Practices

5.1 Versioned Base Path

All endpoints were moved from `/blueprints` to `/api/v1/blueprints`, enabling forward compatibility when the API evolves:

Listing 8: Versioned controller mapping

```
@RestController
@RequestMapping("/api/v1/blueprints")
public class BlueprintsAPIController { ... }
```

5.2 Uniform Response Wrapper

Every endpoint returns an `ApiResponse<T>`, a generic record that standardizes the response shape across the entire API:

Listing 9: ApiResponse record

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

Example response for `GET /api/v1/blueprints/john/house`:

Listing 10: Successful 200 response

```
{
  "code": 200,
  "message": "execute ok",
  "data": {
    "author": "john",
    "name": "house",
    "points": [
      {"x":0,"y":0}, {"x":10,"y":0},
      {"x":10,"y":10}, {"x":0,"y":10}
    ]
  }
}
```

5.3 HTTP Status Code Mapping

Code	Scenario
200	Successful read
201	Blueprint created
202	Point added
400	Validation failure (missing fields)
403	Duplicate blueprint
404	Author or blueprint not found

Table 5: HTTP codes used by the API

5.4 Error Response Format

Errors also follow the `ApiResponse<T>` structure with `data: null`:

Listing 11: 404 error response

```
{
  "code": 404,
  "message": "Blueprint not found: john/xyz",
  "data": null
}
```

6 Part 4: OpenAPI / Swagger

6.1 Configuration

The `springdoc-openapi-starter-webmvc-ui` dependency (v2.6.0) was added to `pom.xml`. `OpenApiConfig.java` customizes the metadata:

Listing 12: OpenAPI bean configuration

```
@Bean
public OpenAPI api() {
    return new OpenAPI()
        .info(new Info()
            .title("ARSW Blueprints API")
            .version("v1")
            .description("REST API for blueprint management"))
        .servers(List.of(new Server()
            .url("http://localhost:8080")
            .description("Development Server")));
}
```

6.2 Endpoint Annotations

All controller methods are annotated with:

- `@Operation`: Summary and description.
- `@ApiResponses`: All possible response codes.
- `@Parameter`: Path variable descriptions and examples.
- `@Schema`: Request/response body examples.

6.3 Swagger UI Evidence

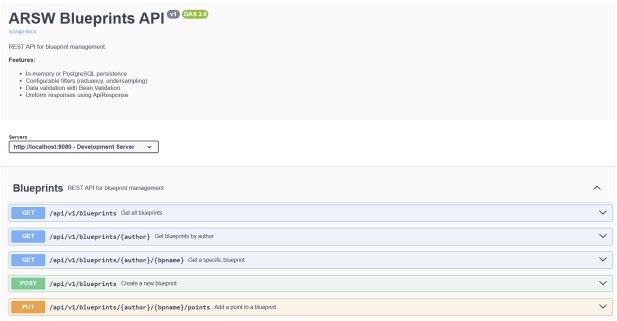

Figure 3: Swagger UI showing all versioned endpoints

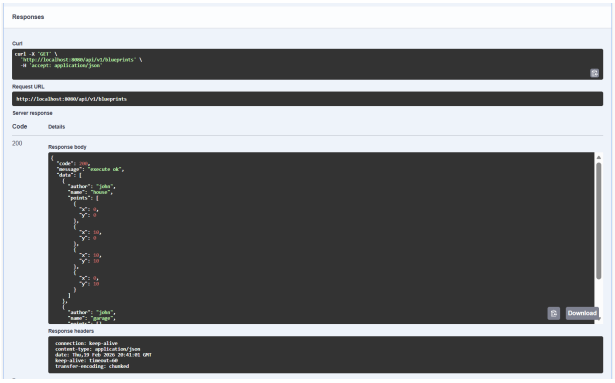

Figure 4: GET /api/v1/blueprints – successful response in Swagger

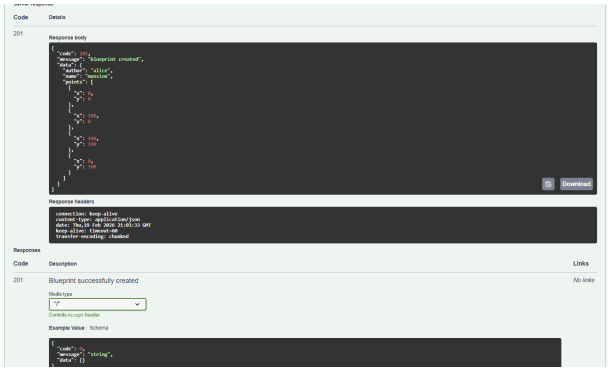

Figure 5: POST – 201 Created response with blueprint data

Figure 6: GET non-existent blueprint – 404 Not Found

7 Part 5: Blueprints Filters

7.1 Design Pattern

Filters follow the **Strategy pattern**: all implement `BlueprintsFilter`, and Spring injects the active one based on the active profile. This avoids any `if/switch` logic in the service layer.

Listing 13: Filter interface

```
public interface BlueprintsFilter {
    Blueprint apply(Blueprint bp);
}
```

7.2 Available Filters

Filter	Profile	Behavior
`IdentityFilter`	`(default)`	Returns blueprint unchanged
`RedundancyFilter`	`redundancy`	Removes consecutive duplicates
`UndersamplingFilter`	`undersampling`	Keeps even-indexed points

Table 6: Available filters and their activation profiles

7.3 RedundancyFilter

Iterates the point list sequentially and discards any point equal to its predecessor:

Listing 14: RedundancyFilter core logic

```
Point prev = null;
for (Point p : in) {
    if (prev == null ||
        !(prev.x()==p.x() && prev.y()==p.y())) {
        out.add(p);
        prev = p;
    }
}
```

Example: `(0,0),(0,0),(1,1),(1,1),(2,2)` → `(0,0),(1,1),(2,2)`

7.4 UndersamplingFilter

Keeps only points at even indices (0, 2, 4, . . .). Blueprints with ≤ 2 points are returned unchanged:

Listing 15: UndersamplingFilter core logic

```
for (int i = 0; i < in.size(); i++) {
    if (i % 2 == 0) out.add(in.get(i));
}
```

Example: `(0,0),(1,1),(2,2),(3,3),(4,4)` → `(0,0),(2,2),(4,4)`

7.5 Profile-Based Activation

The `IdentityFilter` uses a negated profile expression to avoid bean conflicts:

Listing 16: IdentityFilter profile guard

```
@Component
@Profile("!redundancy & !undersampling")
public class IdentityFilter implements BlueprintsFilter {
    @Override
    public Blueprint apply(Blueprint bp) { return bp; }
}
```

7.6 Filter Evidence

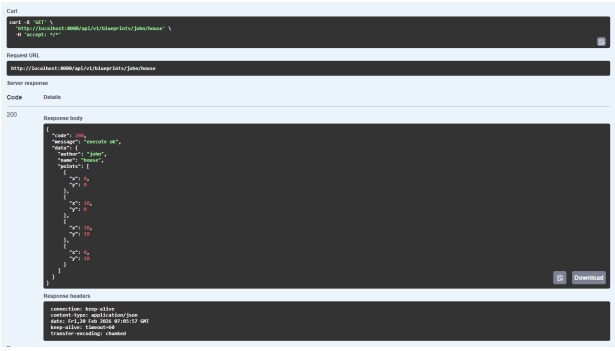

Figure 7: Identity filter: all 4 original points returned

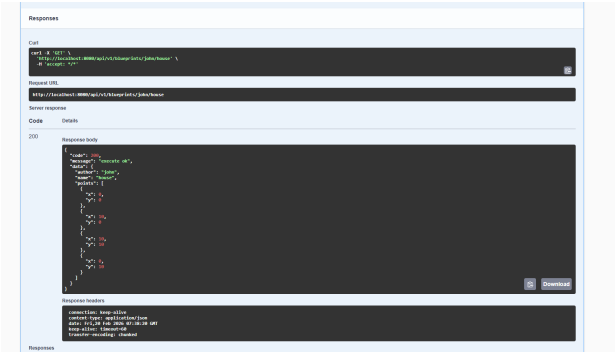

Figure 8: Redundancy filter: consecutive duplicate points removed

Figure 9: Undersampling filter: 4 points reduced to 2 (even indices)

8 Testing Strategy

8.1 Test Suite Overview

Test Class	Layer	#
BlueprintsSmokeTest	Context	1
BlueprintModelTests	Model	7
BlueprintPersistenceTests	InMemory	8
BlueprintServiceTests	Service	9
FiltersTest	Filters	5
BlueprintControllerTests	Controller	9
OpenApiConfigTest	Config	2
BlueprintEntityTest	JPA Entity	4
PointEntityTest	JPA Entity	4
PostgresBlueprintPersistenceTest	Postgres	15
Total		**64**

Table 7: Test suite breakdown

8.2 Notable Test Techniques

MockMvc for Controller Tests – Loads the full Spring context without a real HTTP server, verifying JSON paths and HTTP status codes:

Listing 17: MockMvc controller test example

```
@Test
void testGetBlueprint() throws Exception {
    mockMvc.perform(get("/api/v1/blueprints/john/house"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.code").value(200))
        .andExpect(jsonPath("$.data.author").value("john"));
}
```

Mockito for PostgresPersistence – Uses `@Mock` on `BlueprintJpaRepository` to test the full mapping logic without a database:

Listing 18: ArgumentCaptor verifies toEntity mapping

```
var captor = ArgumentCaptor.forClass(
    BlueprintEntity.class);
verify(repo).save(captor.capture());
BlueprintEntity saved = captor.getValue();
assertEquals(10, saved.getPoints().get(0).getX());
```

8.3 SonarCloud Coverage

JaCoCo was configured without exclusions so that SonarCloud receives coverage data for all layers including JPA entities and the Postgres implementation. Key covered areas:

- All 5 `BlueprintPersistence` methods in Postgres impl.
- `toDomain` and `toEntity` private mapping helpers.
- All getters/setters in `BlueprintEntity` and `PointEntity`.
- `OpenApiConfig.api()` bean creation and server configuration.

9 Bonus Features

9.1 Spring Boot Actuator

Production-ready metrics and health endpoints were enabled by adding `spring-boot-starter-actuator` and configuring:

Listing 19: Actuator properties

```
management.endpoints.web.exposure.include=
    health,info,metrics,env
management.endpoint.health.show-details=always
info.app.name=ARSW Blueprints API
info.app.version=1.0.0
```

Available endpoints at runtime:

Endpoint	URL
Health	`/actuator/health`
Info	`/actuator/info`
Metrics	`/actuator/metrics`
Env	`/actuator/env`

Table 8: Exposed Actuator endpoints

9.2 Container Image

Spring Boot's built-in Cloud Native Buildpacks support was used to generate an OCI image without a manual Dockerfile:

Listing 20: Building the container image

```
mvn spring-boot:build-image -DskipTests
```

The plugin was configured in `pom.xml`:

Listing 21: Image name configuration

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <name>arsw-blueprints-api:${project.version}</name>
    </image>
  </configuration>
</plugin>
```

Running the container with a specific profile:

Listing 22: Running with undersampling filter

```
docker run -p 8080:8080 \
  -e SPRING_PROFILES_ACTIVE=undersampling \
  arsw-blueprints-api:1.0.0
```

Aspect	Dockerfile	build-image
Requires Dockerfile	Yes	No
Layered image	Manual	Automatic
JVM optimization	Manual	Auto-configured

Table 9: Dockerfile vs spring-boot:build-image

10 Conclusions

Key Findings

1. **Layered Architecture Pays Off**: The persistence migration required zero changes to the service and controller layers, demonstrating the value of interface-based design.
2. **Spring Profiles for Environment Switching**: A single `@Profile` annotation switches the entire persistence stack cleanly, eliminating configuration conditionals in business code.
3. **Strategy Pattern for Extensibility**: Adding a new filter requires only a new class implementing `BlueprintsFilter`, with no changes to existing code.
4. **Uniform Responses Improve API Consistency**: `ApiResponse<T>` ensures every endpoint — success or error — shares the same JSON contract, simplifying client-side handling.
5. **OpenAPI as Living Documentation**: Annotating endpoints with `@Operation` and `@ApiResponse` generates documentation that is always in sync with the code.
6. **Testability by Design**: Constructor injection and interface-based persistence allowed Mockito to fully test the Postgres implementation without a real database.

10.1 Lessons Learned

- `@Profile("!a & !b")` is the correct way to make a default bean that yields to alternatives.
- `@OrderColumn` is essential when point order matters downstream.
- JaCoCo exclusions hide real coverage gaps from SonarCloud — prefer writing tests over adding exclusions.
- `spring.autoconfigure.exclude` must be cleared in the postgres profile or JPA won't start.

11 References

1. Spring Boot Reference Documentation (v3.3.x). https://docs.spring.io/spring-boot/docs/3.3.x/reference/html/
2. Springdoc OpenAPI Documentation. https://springdoc.org/
3. JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco/trunk/doc/
4. SonarCloud Documentation. https://docs.sonarcloud.io/
5. Martin, R.C. (2017). *Clean Architecture*. Prentice Hall.
6. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.