Lab 2 — From Monolith to SOAP & REST Webservice in Java

Instructor	**Hamza Gbada**
Course	Service And API
Lab Duration	4 hours
Language	Java 17+

Overview

The company **"DataMart"** manages its product inventory and wants to progressively modernize its system. In this lab you will migrate from a **monolithic Java console application** to a full **Service-Oriented Architecture** using both **SOAP** and **REST** webservices, and then apply **three fundamental design patterns** to improve the codebase.

You will implement **four successive versions** of the system:

1. **Monolithic Console Application** (No SOA — plain Java + JDBC/JPA)
2. **SOAP Webservice** (JAX-WS / Apache CXF + PostgreSQL)
3. **RESTful Webservice** (Spring Boot + JPA/Hibernate)
4. **Design Patterns** applied to the inventory service (Singleton, Factory, Observer)

By the end of this lab you will be able to:

- Explain the limitations of a monolithic architecture.
- Expose a service contract using WSDL (SOAP).
- Design a RESTful API following HTTP semantics.
- Apply and recognize three GoF design patterns in a real service.

Environment Setup

Required Tools

Tool	Version	Purpose
JDK (OpenJDK)	17 +	Java compiler & runtime
Maven **or** Gradle	3.9+ / 8+	Build & dependency management
IntelliJ IDEA **or** VS Code	latest	IDE
Docker & Docker Compose	20+	Container runtime
PostgreSQL	15	Relational database
Postman **or** SoapUI	latest	API testing
curl / HTTPie	any	Quick CLI testing

Docker Compose — PostgreSQL

Save this as `docker-compose.yml` at the root of each part:

```
version: '3.8'
services:
  db:
    image: postgres:15
    container_name: inventory_db
    environment:
      POSTGRES_USER: inventory
      POSTGRES_PASSWORD: inventory
      POSTGRES_DB: inventory
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Start the database with:

```
docker compose up -d
```

> ⚠ **Note:** All four parts share the same PostgreSQL database container. Use the JDBC URL: `jdbc:postgresql://localhost:5432/inventory` (user: `inventory`, password: `inventory`)

Data Model

Every part of this lab uses the same **Product** entity:

Field	Type	Constraints
`id`	`int`	**Primary key**, unique
`name`	`String`	Not null, not empty
`quantity`	`int`	≥ 0
`price`	`double`	≥ 0.0

Part 1 — Monolithic Console Application (No SOA)

Goal

Implement all inventory logic in a single Java project — no network communication, no service separation.

Scenario

DataMart starts with a plain Java application. All business logic, database access, and user interaction are bundled together in the same codebase.

Maven Dependencies (`pom.xml` excerpt)

```
<dependencies>
    <!-- PostgreSQL JDBC driver -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
    </dependency>

    <!-- Hibernate ORM (JPA) -->
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.4.Final</version>
    </dependency>

    <!-- Hibernate Validator (bean validation) -->
    <dependency>
        <groupId>org.hibernate.validator</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>8.0.1.Final</version>
    </dependency>

    <!-- Jakarta Persistence API -->
    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>3.1.0</version>
    </dependency>
</dependencies>
```

Recommended Project Structure

```
part1-monolith/
├── src/
│   └── main/
│       ├── java/com/datamart/
│       │   ├── model/
│       │   │   └── Product.java         ← JPA entity
│       │   ├── repository/
│       │   │   └── ProductRepository.java ← CRUD via EntityManager
│       │   ├── service/
│       │   │   └── InventoryService.java ← business logic
│       │   └── Main.java                ← console menu entry point
│       └── resources/
│           └── META-INF/
│               └── persistence.xml      ← JPA configuration
├── docker-compose.yml
└── pom.xml
```

Tasks

1. **Configure JPA** — Create `persistence.xml` connecting to the PostgreSQL database. Enable `hibernate.hbm2ddl.auto=update` so the `products` table is created automatically.
2. **Product Entity** — Annotate `Product.java` as a JPA `@Entity`. Use Bean Validation annotations (`@NotBlank`, `@Min(0)`) to enforce constraints.
3. **Repository Layer** — Implement `ProductRepository` with the following methods:
    - `void save(Product p)` — persist a new product
    - `Product findById(int id)` — return a product or `null`
    - `List<Product> findAll()` — return all products
    - `void update(Product p)` — merge changes
    - `void delete(int id)` — remove a product
4. **Service Layer** — Implement `InventoryService` that:
    - Calls the repository

- Enforces business rules (no duplicate ID, quantity ≥ 0, price ≥ 0)
- Throws a custom `InventoryException` on validation failures

5. **Console Interface** — Implement a `Main` class with a numbered menu:

```
1. Add Product
2. View Product by ID
3. List All Products
4. Update Product
5. Delete Product
6. Exit
```

   Read input with `Scanner`. Display a friendly message for errors.

6. **Reflection Question** — Answer in your `README.md`:

> *What are the main limitations of this monolithic architecture when the business grows and multiple teams need to use the inventory data?*

Part 2 — SOAP Webservice (JAX-WS / Apache CXF)

Goal

Extract the inventory service from the monolith and expose it as a **SOAP webservice** with a formal **WSDL contract**.

Scenario

Other enterprise systems (ERP, warehouse management) need a machine-readable contract to integrate with the inventory. SOAP provides strict typing and built-in error handling (`SOAP Fault`).

Maven Dependencies

```
<dependencies>
    <!-- Apache CXF JAX-WS -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>4.0.4</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http</artifactId>
        <version>4.0.4</version>
    </dependency>
    <!-- Embedded Jetty server (for standalone hosting) -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http-jetty</artifactId>
        <version>4.0.4</version>
    </dependency>
    <!-- PostgreSQL + Hibernate (same as Part 1) -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.7.3</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.4.Final</version>
    </dependency>
</dependencies>
```

Recommended Project Structure

```
part2-soap/
├── src/main/java/com/datamart/
│   ├── model/
│   │   └── Product.java
│   ├── repository/
│   │   └── ProductRepository.java
│   ├── service/
│   │   ├── InventoryService.java      ← @WebService interface
│   │   └── InventoryServiceImpl.java  ← @WebService implementation
│   └── Main.java                      ← publishes the endpoint
├── docker-compose.yml
└── pom.xml
```

Tasks

1. **Service Interface** — Annotate `InventoryService.java` with `@WebService`. Declare the following operations:

Operation	Input	Output
`createProduct`	`id, name, quantity, price`	success message `String`
`getProduct`	`id`	`Product`
`updateProduct`	`id, name, quantity, price`	success message `String`
`deleteProduct`	`id`	success message `String`
`listProducts`	*(none)*	`List<Product>`

2. **Service Implementation** — Implement the interface in `InventoryServiceImpl.java`. Annotate with `@WebService(endpointInterface = "...")`. Reuse the repository from Part 1. Throw a `WebServiceException` with a descriptive message on validation failures.
3. **Publish the Endpoint** — In `Main.java`, use `Endpoint.publish("http://localhost:8080/inventory", new InventoryServiceImpl())` to expose the service.
4. **Verify WSDL** — Start the service and open the following URL in a browser:

   ```
   http://localhost:8080/inventory?wsdl
   ```

   Confirm the WSDL contains all five operations.
5. **Test with SoapUI or Postman** — Import the WSDL into SoapUI, or craft a raw SOAP envelope in Postman (see Appendix A).
6. **Unit Tests** — Write at least 3 JUnit 5 tests for the service implementation (e.g., create then retrieve, duplicate ID rejection, invalid quantity).
7. **Reflection Question**:

   > *How does the WSDL contract enforce structure? Compare the explicit typing in SOAP with the free-form interface of your monolithic app.*

Appendix A — Testing SOAP with Postman

Step 1 — Create a POST request

- URL: `http://localhost:8080/inventory`
- Method: **POST**
- Header: `Content-Type: text/xml;charset=UTF-8`

Step 2 — Body (raw XML) — *createProduct* example

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:inv="http://service.datamart.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <inv:createProduct>
         <id>1</id>
         <name>Laptop</name>
         <quantity>10</quantity>
         <price>1299.99</price>
      </inv:createProduct>
   </soapenv:Body>
</soapenv:Envelope>
```

Step 3 — Other operations

Adapt the body, replacing `createProduct` with `getProduct`, `updateProduct`, `deleteProduct`, or `listProducts` and supplying the relevant parameters.

Part 3 — RESTful Webservice (Spring Boot)

Goal

Refactor the SOAP service into a **RESTful API** using Spring Boot, following standard HTTP conventions.

Scenario

Modern web and mobile clients expect a lightweight JSON API. REST is easier to consume, self-documenting with OpenAPI/Swagger, and better suited for public-facing services.

Maven Dependencies (`pom.xml`)

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.5</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <!-- OpenAPI / Swagger UI -->
    <dependency>
        <groupId>org.springdoc</groupId>
        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
        <version>2.5.0</version>
    </dependency>
    <!-- Testing -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/inventory
spring.datasource.username=inventory
spring.datasource.password=inventory
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
springdoc.swagger-ui.path=/swagger-ui.html
```

Recommended Project Structure

```
part3-rest/
├── src/
│   ├── main/java/com/datamart/
│   │   ├── model/
│   │   │   └── Product.java
│   │   ├── repository/
│   │   │   └── ProductRepository.java   ← extends JpaRepository
│   │   ├── service/
│   │   │   └── InventoryService.java
│   │   ├── controller/
│   │   │   └── ProductController.java   ← @RestController
│   │   ├── dto/
│   │   │   ├── ProductRequest.java      ← input DTO with validation
│   │   │   └── ProductResponse.java     ← output DTO
│   │   ├── exception/
│   │   │   ├── ProductNotFoundException.java
│   │   │   └── GlobalExceptionHandler.java ← @ControllerAdvice
│   │   └── InventoryApplication.java
│   └── main/resources/
│       └── application.properties
├── docker-compose.yml
└── pom.xml
```

REST API Contract

Method	Endpoint	Description	Request Body	Response
`GET`	`/api/products`	List all products	—	`200 OK` + JSON
`GET`	`/api/products/{id}`	Get product by ID	—	`200` or `404`
`POST`	`/api/products`	Create a new product	`ProductRequest`	`201 Created`
`PUT`	`/api/products/{id}`	Update existing product	`ProductRequest`	`200` or `404`
`DELETE`	`/api/products/{id}`	Delete a product	—	`204 No Content`

Tasks

1. **Product Entity** — Reuse the JPA `@Entity` from Part 1 (same table).
2. **Repository** — Create `ProductRepository` that extends `JpaRepository<Product, Integer>`.
3. **DTOs** — Define `ProductRequest` (with `@NotBlank`, `@Min(0)` validation) and `ProductResponse`.
4. **Service Layer** — Implement `InventoryService` to:
   - Delegate CRUD to the repository
   - Throw `ProductNotFoundException` for missing IDs
   - Map between entities and DTOs
5. **REST Controller** — Implement `ProductController` with `@RestController`. Map each HTTP method/endpoint to the corresponding service call. Return correct HTTP status codes (`201`, `200`, `404`, `204`).
6. **Global Exception Handler** — Use `@ControllerAdvice` to return JSON error bodies for `ProductNotFoundException` (`404`) and `MethodArgumentNotValidException` (`400`).
7. **OpenAPI Documentation** — Access interactive docs at:

   ```
   http://localhost:8080/swagger-ui.html
   ```

   Verify all five endpoints appear with correct schemas.
8. **Integration Tests** — Write at least 3 integration tests using `@SpringBootTest` + `MockMvc`.
9. **Dockerize** — Write a `Dockerfile` for the Spring Boot application and add it to `docker-compose.yml`. Ensure the API and database containers can communicate.
10. **Reflection Question**:

> *Compare SOAP and REST in terms of: contract strictness, payload format, error handling, tooling, and ease of client integration.*

Appendix B — Quick curl Testing

```
# Create a product
curl -X POST http://localhost:8080/api/products \
  -H "Content-Type: application/json" \
  -d '{"id":1, "name":"Laptop", "quantity":10, "price":1299.99}'

# List all products
curl http://localhost:8080/api/products

# Get one product
curl http://localhost:8080/api/products/1

# Update a product
curl -X PUT http://localhost:8080/api/products/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"Gaming Laptop", "quantity":5, "price":1499.99}'

# Delete a product
curl -X DELETE http://localhost:8080/api/products/1
```

Part 4 — Design Patterns

Goal

Apply three fundamental **GoF (Gang of Four) design patterns** to improve the architecture of the inventory system.

4.1 — Singleton Pattern

Concept

The **Singleton** guarantees that a class has **only one instance** throughout the application's lifecycle, and provides a **global access point** to it.

```
┌──────────────────────────┐
│       Singleton          │
├──────────────────────────┤
│ - instance: Singleton    │
├──────────────────────────┤
│ - Singleton()            │ ← private constructor
│ + getInstance(): Singleton│ ← lazy or eager init
│ + doSomething(): void    │
└──────────────────────────┘
```

Why use it here?

A **database connection pool** (or a JDBC connection) should be shared across the application. Creating a new connection on every request is expensive and unsustainable.

Your Task

1. Create a class `DatabaseConnectionManager` that:
    - Has a **private static** field holding the single instance.
    - Has a **private constructor** that initializes a `DataSource` (connection pool) using HikariCP or a single `Connection`.
    - Exposes a **thread-safe `getInstance()`** static method (use `double-checked locking` or the `initialization-on-demand holder` idiom).
    - Exposes a `getConnection()` method.
2. Modify `ProductRepository` to obtain connections from `DatabaseConnectionManager.getInstance()`.
3. Write a JUnit test that:

- Calls `getInstance()` from two different threads.
- Asserts that **both references point to the same object**.

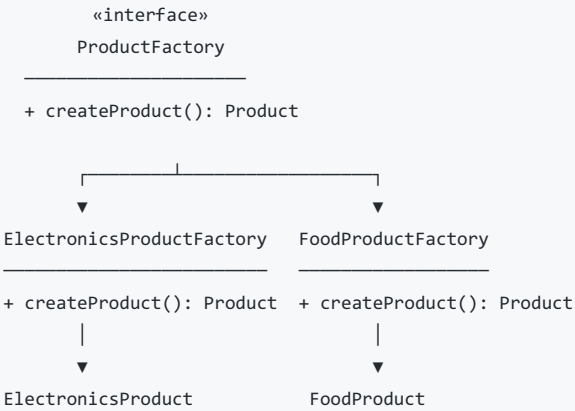
4. Reflection question:

> *What is the difference between eager initialization and lazy initialization for a Singleton? When should you prefer one over the other?*

4.2 — Factory Method Pattern

Concept

The **Factory Method** defines an interface for creating objects, but lets **subclasses decide which class to instantiate**. It promotes the *Open/Closed Principle*: open for extension, closed for modification.



Why use it here?

DataMart sells multiple **categories** of products (Electronics, Food, Clothing). Each category may have different default validation rules, tax rates, or shelf-life constraints. A factory isolates product construction from business logic.

Your Task

1. Create a `ProductFactory` interface with method:

```
Product createProduct(int id, String name, int quantity, double price);
```

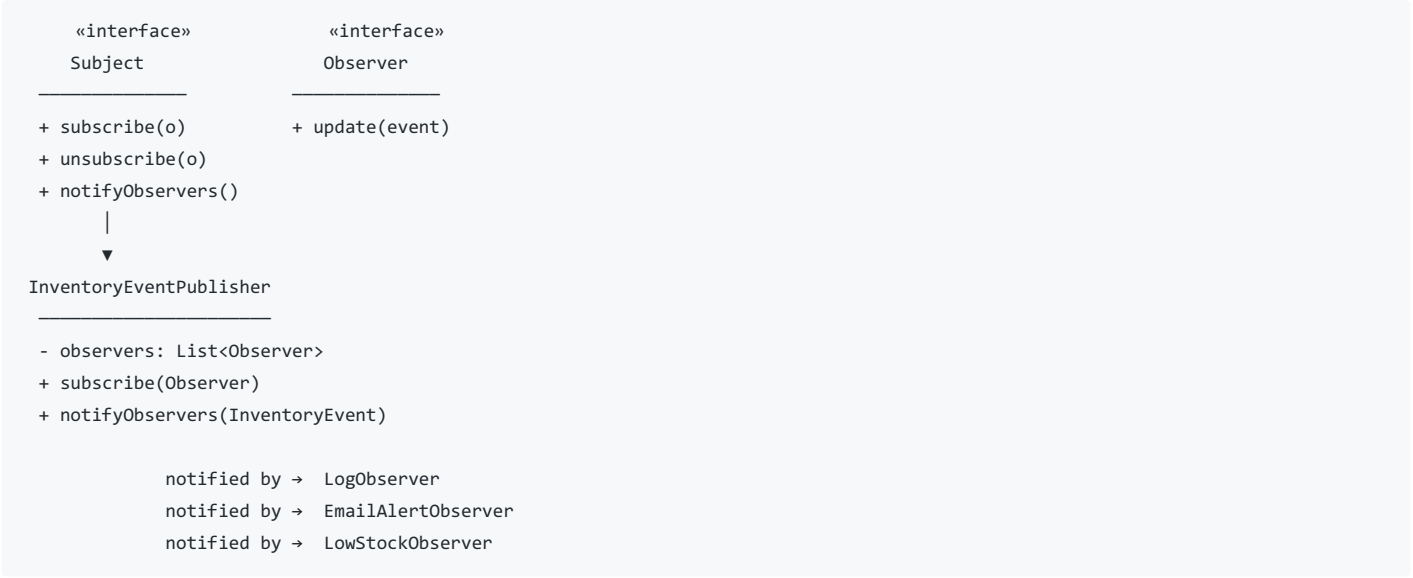
2. Implement at least **two concrete factories**:
   - `ElectronicsProductFactory` — wraps the product with `category = "ELECTRONICS"` and enforces `price > 0`.
   - `FoodProductFactory` — wraps the product with `category = "FOOD"`, enforces `quantity > 0`, and sets a `expiryDays` field.
3. Add a `category` field (and `expiryDays` for food) to `Product` or use **subclasses** (`ElectronicsProduct`, `FoodProduct extends Product`).
4. Update the REST controller's `POST /api/products` endpoint to accept a `category` field in the request. Instantiate the correct factory based on that field. Use a `FactoryRegistry` (a `Map<String, ProductFactory>`) to look up the right factory.
5. Write JUnit tests verifying that:
   - Requesting category `"ELECTRONICS"` returns an `ElectronicsProduct`.
   - Requesting category `"FOOD"` returns a `FoodProduct` with correct defaults.
   - An unknown category throws an `IllegalArgumentException`.
6. Reflection question:

> *Compare the Factory Method pattern with direct instantiation using `new`. What are the advantages of the factory in terms of testability and extensibility?*

4.3 — Observer Pattern

Concept

The **Observer** (also called *Publish-Subscribe* or *Event Listener*) defines a **one-to-many dependency** so that when one object (the *Subject*) changes state, all its **observers** are notified automatically.



Why use it here?

When a product's **quantity drops below a threshold**, the system should:

- Log the event.
- Send an email alert (simulated).
- Trigger a restock notification.

These are **cross-cutting concerns** that should not be mixed into the core service logic.

Your Task

1. Define an `InventoryEvent` record/class carrying:

   ```
   record InventoryEvent(String type, Product product, String message) {}
   ```

   Types: `PRODUCT_CREATED`, `PRODUCT_UPDATED`, `PRODUCT_DELETED`, `LOW_STOCK`.

2. Define the `InventoryObserver` interface:

   ```
   public interface InventoryObserver {
       void onEvent(InventoryEvent event);
   }
   ```

3. Implement three observers:
   - `LogObserver` — prints a formatted log line to `System.out`.
   - `EmailAlertObserver` — prints a simulated email (e.g. "Sending email: Product X is below minimum stock").
   - `LowStockObserver` — triggers only on `LOW_STOCK` events; prints a restock recommendation.
4. Implement `InventoryEventPublisher`:
   - Maintains a `List<InventoryObserver>`.
   - Exposes `subscribe(InventoryObserver)` and `unsubscribe(InventoryObserver)`.
   - Exposes `publish(InventoryEvent)` which iterates observers.
5. Integrate into `InventoryService`:
   - After a `createProduct` or `updateProduct` call, publish a `PRODUCT_CREATED` / `PRODUCT_UPDATED` event.
   - If `quantity < 5`, also publish a `LOW_STOCK` event.
6. Wire everything up in the Spring context using `@Bean` or `@Component`.
7. Write JUnit tests verifying:
   - All registered observers are notified on a `PRODUCT_CREATED` event.
   - Only `LowStockObserver` reacts to `LOW_STOCK`.
   - Unsubscribed observers no longer receive events.
8. Reflection question:

   > *How does the Observer pattern improve decoupling compared to calling logging, email, and restock logic directly inside `InventoryService`? What is the Open/Closed Principle implication here?*

Submission Checklist

For each part, submit:

- [ ] Source code (Maven project) zipped or pushed to a Git repository
- [ ] `docker-compose.yml` + `Dockerfile` (Parts 2–4)
- [ ] `README.md` with:
  - Setup instructions
  - How to run
  - Answers to all reflection questions
- [ ] Test reports (`mvn test` output)
- [ ] Part 2: WSDL screenshot or file
- [ ] Part 3: Swagger UI screenshot

Grading Criteria

Criterion	Points
Part 1 — Monolith (CRUD + JPA + validation)	20
Part 2 — SOAP (contract + WSDL + tests)	20
Part 3 — REST (endpoints + DTOs + Swagger + Docker)	25
Part 4 — Singleton (thread-safety + tests)	10
Part 4 — Factory Method (extensibility + tests)	12
Part 4 — Observer (decoupling + tests)	13
Total	**100**

Good luck! Ask your instructor for clarification on any section.