**📘 XYZ E-Commerce Platform – Engineering Design Document**

**🧾 Title & Metadata**

**Project Name**: XYZ E-Commerce Platform **Document Type**: Technical Design Specification **Version**: 1.0 **Date**: July 10, 2025 **Author**: Architecture Team (via Microsoft Copilot)

**📑 Table of Contents**

1. Executive Summary
2. Business Scenario
3. Project Objectives
4. Architecture Vision
5. Technology Stack
6. High-Level Architecture
7. Microservices Overview
8. High-Level Design (HLD)
9. Low-Level Design (LLD)
10. Database Design
11. API Contracts
12. Inter-Service Communication
13. Configuration Management
14. Security Design
15. Error Handling Strategy
16. Testing Strategy
17. Deployment Plan
18. Multithreading Strategy
19. Design Patterns Used
20. Common Library Design
21. Monitoring & Logging
22. Scalability & Resilience
23. Roadmap
24. Risks & Assumptions
25. Glossary
26. Appendix A: Diagrams
27. Appendix B: Code Samples
28. Appendix C: SQL Scripts
29. Appendix D: Docker & CI/CD
30. Approval Page

**✅ 1. Executive Summary**

XYZ aims to build an e-commerce platform using Spring Boot microservices architecture with Java 17, designed to be scalable, reliable, and maintainable. This document outlines the technical foundation for developing an MVP version with five services and modular components.

**📘 2. Business Scenario**

Company XYZ seeks to transition its retail platform online, enabling self-service purchases, multi-channel integration, and distributed operations. This system leverages microservices to allow independent evolution of features and resilience across components.

**🎯 3. Project Objectives**

* Deliver a working MVP
* Follow API-first architecture
* Design five independent microservices
* Use relational databases for data segregation
* Implement communication via REST
* Establish foundational testing and deployment pipelines

**🧠 4. Architecture Vision**

**Key Principles:**

* **Microservice Isolation**
* **API-First Interfaces**
* **One DB per Service**
* **Clear Service Contracts**
* **Modular Logging**

Architecture adheres to cloud-native principles with a view to scale and automate.

**🧰 5. Technology Stack**

| **Layer** | **Technology** |
| --- | --- |
| Language | Java 17 |
| Framework | Spring Boot |
| Database | H2, MySQL, PostgreSQL |
| Communication | REST, FeignClient |
| Testing | JUnit 5, Mockito |
| Deployment | Docker |
| Configuration | application.properties |
| CI/CD | GitHub Actions (Future) |

**🧱 6. High-Level Architecture**

plaintext

[ User / Admin Interface ]

↓

+--------------------+

| API Gateway |

+--------------------+

↓ ↓ ↓

[User Service] [Product Service] [Order Service]

↓ ↓ ↓

[ MySQL/H2 ] [ MySQL ] [ PostgreSQL ]

↓

[ Payment Service ]

↓

[ Card Validation ]

All services are loosely coupled with individual databases.

**🧠 Layered Architecture**

Each microservice follows **3-tier layered architecture**:

[ Controller Layer ]

↓

[ Service Layer ]

↓

[ Repository Layer ]

↓

[ Database (per service) ]

**🧱 Microservices Breakdown**

| **Microservice** | **Port** | **DB Engine** | **Key Responsibilities** |
| --- | --- | --- | --- |
| user-service | 8081 | MySQL | Register, login, profile |
| product-catalog-service | 8082 | MySQL | Product listing, filtering, pricing |
| order-service | 8083 | PostgreSQL | Order placement, status updates, notification |
| payment-service | 8084 | MySQL | Initiate payments, call card validation service |
| card-validation-service | 8085 | H2 (In-memory) | Validate card numbers via GET endpoint |
| common-lib | — | — | Shared logging (LoggerUtil), reusable components |

**🧩 7. Microservices Overview**

| **Microservice** | **Functionality** |
| --- | --- |
| User Service | Registration, authentication, profile |
| Product Catalog | Product details, pricing, availability |
| Order Service | Order lifecycle, status updates |
| Payment Service | Card validation + gateway integration |
| Card Validation | Basic GET validation of card info |
|  |  |

**🏗️ 8. High-Level Design (HLD)**

Each microservice follows standard layered architecture:

* **Controller Layer**
* **Service Layer (business logic)**
* **Repository Layer (Spring Data JPA)**

Service contracts defined via OpenAPI (Swagger) and REST conventions.

**🔧 9. Low-Level Design (LLD)**

Each service contains:

* @Entity classes
* @Repository interfaces
* @Service logic
* @RestController APIs
* Exception handling via @ControllerAdvice

Details are available in Appendix B (Code Samples).

**🗄️ 10. Database Design**

| **Microservice** | **DB Engine** | **Reason** |
| --- | --- | --- |
| User Service | MySQL | Structured user data |
| Product Catalog | MySQL | Fast querying on categories |
| Order Service | PostgreSQL | Complex joins, reliability |
| Payment Service | MySQL | Transaction-heavy inserts |
| Card Validation | H2 (Dev) | Lightweight checks |
|  |  |  |

**🌐 11. API Contracts**

**User Service**

**POST /users**

json

{

"name": "Alice",

"email": "alice@xyz.com",

"password": "secure123"

}

**GET /users/{id}** Returns full user object with status 200 OK

**🔁 12. Inter-Service Communication**

FeignClient (preferred method):

java

@FeignClient(name = "card-service", url = "${card.service.url}")

public interface CardClient {

@GetMapping("/validate/{cardNumber}")

Boolean validate(@PathVariable String cardNumber);

}

RestTemplate (alternative):

java

restTemplate.getForObject("http://localhost:8082/validate/" + cardNum, Boolean.class);

**⚙️ 13. Configuration Management**

**application.properties**

properties

server.port=8081

spring.datasource.url=jdbc:mysql://localhost:3306/user\_db

spring.datasource.username=root

spring.datasource.password=secret

card.service.url=http://localhost:8082/validate

**Java usage**

java

@Value("${card.service.url}")

private String cardServiceUrl;

**🔐 14. Security Design**

* Passwords hashed using BCrypt
* JWT and OAuth2 planned in next phase

java

BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

String encrypted = encoder.encode("secure123");

**🚫 15. Error Handling Strategy**

Global error handler:

java

@RestControllerAdvice

public class ExceptionHandler {

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<ApiError> handleNotFound(UserNotFoundException ex) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(new ApiError(...));

}

}

**🧪 16. Testing Strategy**

**Controller Testing**

java

@WebMvcTest(UserController.class)

class UserTest {

@MockBean UserService service;

@Autowired MockMvc mvc;

@Test

void testGetUser() throws Exception {

mvc.perform(get("/users/1")).andExpect(status().isOk());

}

}

**Service Testing**

java

@SpringBootTest

class UserServiceTest {

@Mock UserRepository repo;

@InjectMocks UserService service;

}

**🚢 17. Deployment Plan**

**Dockerfile**

dockerfile

FROM openjdk:17

COPY target/user-service.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

EXPOSE 8080

**docker-compose.yml**

yaml

services:

user-service:

build: ./user-service

ports: ["8080:8080"]

db:

image: mysql:8

environment:

MYSQL\_ROOT\_PASSWORD: root

MYSQL\_DATABASE: user\_db

**🔄 18. Multithreading Strategy**

Used in Order Service:

java

public void handleOrder(Order order) {

placeOrder(order); // Main thread

new Thread(() -> sendNotification(order)).start(); // Async

}

Using executor:

java

@Bean

public Executor executor() {

ThreadPoolTaskExecutor e = new ThreadPoolTaskExecutor();

e.setCorePoolSize(5);

e.setMaxPoolSize(10);

e.setQueueCapacity(100);

e.setThreadNamePrefix("Async-");

e.initialize();

return e;

}

**🧩 19. Design Patterns Used**

| **Pattern** | **Purpose** | **Used In** |
| --- | --- | --- |
| Builder | Object creation | User profiles |
| DAO & Repo | Data separation | All services |
| Observer | Event notifications | Order → Payment |

**🛠️ 20. Common Library Design**

* common-logger module
* Contains SLF4J wrapper
* Injected via dependency into all services

ava

@Slf4j

public class LoggerUtil {

public static void log(String msg) {

log.info("Log: {}", msg);

}

}

**📊 21. Monitoring & Logging**

* Spring Boot Actuator
* Future: ELK stack (ElasticSearch, Logstash, Kibana)

properties

management.endpoints.web.exposure.include=health,info,metrics

**⚙️ 22. Scalability & Resilience**

* Redis caching layer for frequently accessed data
* Circuit Breaker