**Architecture Document for E-Commerce System in Spring Boot**

Table of Contents

[**Architecture Document for E-Commerce System in Spring Boot** 1](#_Toc174910469)

**1. Introduction**

**1.1. Purpose**

The purpose of this document is to design and implement a microservices-based e-commerce platform using Spring Boot. The system will adhere to microservices principles, leveraging patterns, asynchronous communication, testing strategies, and DevOps practices. This document outlines the architecture, modules, and their interactions to ensure a scalable, maintainable, and resilient system.

**1.2. Scope**

This architecture document covers the design and implementation of the following services:

* Product Catalog Service
* Order Processing Service
* Customer Service
* Inventory Service
* Payment Service

**2. System Overview**

**2.1. High-Level Architecture**

The e-commerce system is designed as a collection of microservices, each responsible for a specific domain. The architecture follows the principles of Domain-Driven Design (DDD), ensuring that each service is decoupled and independently deployable.

**2.2. Microservices Overview**

* **Product Catalog Service**: Manages product information, categories, and search functionality.
* **Order Processing Service**: Handles order creation, tracking, and management.
* **Customer Service**: Manages customer information, authentication, and profiles.
* **Inventory Service**: Tracks product stock levels, manages inventory updates.
* **Payment Service**: Processes payments, manages transactions and integration with payment gateways.

**3. Architecture Components**

**3.1. Microservices Design**

Each microservice is designed as an independent Spring Boot application with its own database, following the Database per Service pattern. The services communicate via REST APIs (webflux) and asynchronous messaging using Kafka.

**3.2. API Gateway**

An API Gateway (Spring Cloud Gateway) acts as the entry point for all client requests, routing them to the appropriate micro services. It also handles cross-cutting concerns like authentication, rate limiting, and logging.

**3.3. Service Discovery**

Eureka is used for service discovery, allowing micro services to dynamically discover each other and enabling load balancing.

**3.4. Configuration Management**

Spring Cloud Config Server is used for centralized configuration management, storing configurations in a version-controlled repository.

**3.5. Asynchronous Communication**

For decoupling services and ensuring scalability, asynchronous communication is implemented using Kafka. This is particularly useful for operations like order processing and inventory updates, where services can operate independently without waiting for responses.

**3.6. Circuit Breaker**

Resilience4j is integrated into the system to implement the Circuit Breaker pattern, preventing cascading failures and ensuring system reliability.

**3.7. Database**

Each micro service manages its own MySQL database, ensuring data isolation and independence.

**3.8. Security**

OAuth 2.0 and JWT are used for securing the microservices, ensuring that only authenticated users and services can access the system.

**3.9. Logging and Monitoring**

Prometheus and Grafana are used for centralized logging and monitoring. This helps in tracking service health, performance, and debugging issues.

**4. Service Details**

**4.1. Product Catalog Service**

* **Responsibilities**: Manage product information, categories, search, and filtering.
* **Technologies**: Spring Boot, Reactive Crud, MySQL

**4.2. Order Processing Service**

* **Responsibilities**: Handle order creation, status tracking, and order history.
* **Technologies**: Spring Boot, Reactive Crud, MySQL

**4.3. Customer Service**

* **Responsibilities**: Manage customer profiles, authentication, and user data.
* **Technologies**: Spring Boot, Reactive Crud, MySQL

**4.4. Inventory Service**

* **Responsibilities**: Track inventory levels, manage stock updates, and prevent overselling.
* **Technologies**: Spring Boot, Reactive Crud, MySQL

**4.5. Payment Service**

* **Responsibilities**: Handle payment processing, manage transactions, and integrate with payment gateways.
* **Technologies**: Spring Boot, Reactive Crud, MySQL

**5. Asynchronous Communication**

**5.1. Message Broker**

Kafka is used as the message broker to enable asynchronous communication between services. Key events include:

* Order Created
* Inventory Updated
* Payment Processed

**5.2. Event-Driven Design**

Services publish and subscribe to events, ensuring loose coupling and high scalability.

**6. Testing Strategies**

**6.1. Unit Testing**

* **Tools**: JUnit, Mockito
* **Focus**: Testing individual components and services.

**6.2. Integration Testing**

* **Tools**: Spring Boot Test, TestContainers
* **Focus**: Testing interactions between microservices and external dependencies.

**6.3. End-to-End Testing**

* **Tools**: Cucumber, Selenium
* **Focus**: Testing the entire user journey through the system.

**7. DevOps Practices**

**7.1. CI/CD Pipeline**

* **Tools**: Jenkins,GitHub Actions
* **Pipeline**: Automates building, testing, and deploying microservices.

**7.2. Containerization**

* **Tools**: Docker, Kubernetes
* **Setup**: Each microservice is containerized and managed using Kubernetes for orchestration.

**7.3. Monitoring and Logging**

* **Tools**: Prometheus, Grafana, ELK Stack
* **Purpose**: Continuous monitoring of service health, performance, and centralized logging.

**8. Conclusion**

This architecture document outlines the design and implementation strategy for a micro services-based e-commerce system using Spring Boot. By adhering to micro services principles and best practices, the system is designed to be scalable, maintainable, and resilient, capable of handling the dynamic needs of an e-commerce platform.