E-commerce Order Management System (OMS)

1. System Overview

The **E-commerce Order Management System (OMS)** is a microservices-based platform designed to handle the complete lifecycle of an online order, from creation to fulfillment, including inventory validation, payment processing, and shipping. The microservices architecture ensures scalability, fault isolation, and maintainability while supporting both synchronous (REST) and asynchronous (message-driven) communication.

Goals:

- Seamless handling of e-commerce orders.
- High availability and responsiveness during peak loads.
- Flexibility to integrate multiple payment gateways and logistics providers.
- Event-driven notifications and analytics support.

---

2. Microservices Architecture

2.1 Core Services

Service	Responsibilities	Database Type	Communication
Order Service	Create, validate, confirm, update orders; manage order status; publish domain events.	MySQL/PostgreSQL	REST, Feign client, Kafka (events)
Inventory Service	Track product stock, update availability, prevent overselling.	MySQL	REST, Kafka (async updates)
Product Service	Manage product catalog, pricing, descriptions, and metadata.	MongoDB	REST
Customer Service	Handles registration, authentication, profile, and order history.	MySQL/PostgreSQL	REST, JWT security

Service	Responsibilities	Database Type	Communication
Payment Service	Process payments, integrate with gateways (Credit Card, PayPal, Razorpay, UPI).	MySQL	REST, async events
Shipping & Logistics Service	Manage shipment, track deliveries, calculate estimated arrival.	MySQL	REST, async events
Notification Service	Send emails, SMS, or push notifications based on domain events.	MongoDB/Redis	Kafka (events)
Analytics Service *(optional)*	Collect and analyze sales and operational metrics.	PostgreSQL/BigQuery	Event-driven

2.2 Supporting Infrastructure

- **API Gateway (Spring Cloud Gateway)**: Entry point for all client requests, routing to appropriate microservices and handling cross-cutting concerns (authentication, rate limiting).
- **Service Discovery (Eureka)**: Enables automatic registration and discovery of microservices.
- **Message Broker (Kafka/RabbitMQ)**: Facilitates asynchronous inter-service communication for events like OrderCreated or InventoryLow.
- **Circuit Breakers & Resilience (Resilience4j)**: Prevent cascading failures by handling service timeouts and retries.
- **Centralized Configuration (Spring Cloud Config)**: Consistent, dynamic configuration management across microservices.
- **Observability**: Logging (Loki), Metrics (Prometheus), Tracing (Grafana Tempo).

---

3. Domain-Driven Design (DDD) Elements

3.1 Bounded Contexts

- **Customer Context**: Authentication, user profiles.
- **Product Context**: Product catalog and pricing.

- **Order Context**: Order creation, validation, and fulfillment.

3.2 Aggregates and Value Objects

- **Order Aggregate**: Order, OrderItem, PaymentInfo.
- **Product Aggregate**: Product, Inventory.
- **Customer Aggregate**: Customer, Address, Email.
- **Value Objects**: Money, Email, CustomerId, ProductId, OrderId.

3.3 Domain Events

- OrderCreatedEvent
- PaymentProcessedEvent
- InventoryUpdatedEvent
- OrderShippedEvent

---

4. API Design

4.1 Resource-Based Endpoints

- **Order Service**
    - POST /api/orders: Create new order
    - GET /api/orders/{id}: Retrieve order details
    - PATCH /api/orders/{id}/status: Update order status
- **Product Service**
    - GET /api/products: List all products
    - POST /api/products: Add product
    - GET /api/products/{id}: Get product detail
- **Inventory Service**
    - GET /api/inventory/{productId}: Check stock quantity
    - PUT /api/inventory/{productId}: Adjust stock level
- **Customer Service**
    - POST /api/customers: Register user
    - GET /api/customers/{id}: Get customer profile

4.2 Integration via Feign Client

```
@FeignClient(name = "inventory-service")
public interface InventoryClient {
  @GetMapping("/api/inventory/{productId}")
  Inventory getInventory(@PathVariable Long productId);
}
```

---

5. Database Schema (Simplified)

Customers Table

- customer_id (PK), name, email, password_hash, address.

Products Table

- product_id (PK), name, description, price, category_id, created_at.

Orders Table

- order_id (PK), customer_id (FK), status, total_amount, order_date.

Order_Items Table

- order_item_id (PK), order_id (FK), product_id (FK), quantity, price.

Inventory Table

- inventory_id (PK), product_id (FK), quantity, location_id.

Payments Table

- payment_id (PK), order_id (FK), amount, payment_method, status, payment_date.

---

6. Communication Patterns

- **Synchronous**: REST + OpenFeign for inter-service calls where immediate feedback is required (inventory validation, product details).
- **Asynchronous/Event-Driven**: Kafka for notifying services of state changes (order creation, shipment, payment confirmation).
- **API Gateway** handles routing and authentication (Keycloak integration for OAuth2).

---

7. Resilience, Security, and Observability

- **Resilience**
    - Circuit Breaker via Resilience4j
    - Retry mechanisms with exponential backoff
- **Security**
    - OAuth2 with Keycloak
    - JWT tokens for microservice authentication
    - HTTPS enforced at API Gateway
- **Observability**
    - Logging: centralized via Loki
    - Metrics: Prometheus + Micrometer
    - Tracing: Grafana Tempo

---

8. Deployment Considerations

- **Containerization**: Docker for each microservice.
- **Orchestration**: Kubernetes (K8s) for scaling, load balancing, and self-healing.
- **Configuration Management**: Spring Cloud Config for environment-based configuration.
- **Database Migration**: Flyway/Liquibase for schema changes.
- **Monitoring & Alerts**: Grafana dashboards with alerting for high-latency or failed transactions.

---

9. Testing Strategy

- **Unit Tests**: For domain logic and value objects.
- **Integration Tests**: Feign client interactions and database operations.
- **End-to-End Tests**: Whole order lifecycle including asynchronous events.
- **Test Environment**: Use TestContainers for ephemeral databases (MySQL, MongoDB) to mimic production setups.

---

10. Sample Order Creation Flow

1. Customer submits order via front-end.
2. API Gateway routes request to **Order Service**.
3. **Order Service** validates inventory via **Inventory Service** (Feign client).
4. Payment initiated via **Payment Service**.
5. Order status updated from PENDING → CONFIRMED.
6. Domain Event OrderCreatedEvent published to Kafka.
7. **Notification Service** sends email/SMS.
8. **Shipping Service** schedules shipment.

---

11. Advantages of This Design

- **Scalability**: Microservices can be scaled independently.
- **Fault Tolerance**: Circuit breakers prevent cascading failures.
- **Maintainability**: Independent codebases, aggregated by bounded contexts.
- **Extensibility**: New features like promotions, 3PL integrations, or subscription services can be added without disrupting the system.
- **Observability & Monitoring**: Full visibility into microservice interactions and performance.

References

- Spring Boot Microservices Example
- GitHub: Spring Boot Microservices OMS
- DDD-based E-commerce OMS with Spring Boot
- Spring Commerce Microservices Patterns

This design provides a robust foundation for a production-ready, microservices-based e-commerce order management system using Spring Boot.

Source(s):
1. https://dev.to/devcorner/spring-boot-ddd-e-commerce-order-management-system-detailed-walkthrough-12ie
2. https://github.com/Ali-Modassir/SpringBoot-Microservices-Order-Management-System
3. https://www.geeksforgeeks.org/springboot/java-spring-boot-microservices-example-step-by-step-guide/
4. https://www.sourcecodeexamples.net/2024/05/spring-boot-microservices-e-commerce-project.html