COMP41720 Distributed Systems Lab4 Microservices E-Commerce System

Ze Li, 24203409

Abstract—A microservice-based e-commerce system with three services and an API gateway.
Source code: https://github.com/zeli8888/COMP41720-Distributed-Systems/tree/main/lab4

Index Terms—Microservice, API gateway, Rest, Grpc, Kafka, Docker, Kubernetes, Spring Boot

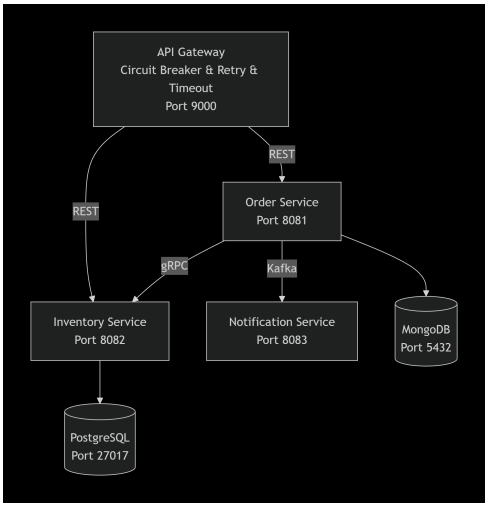
1 Introduction

This lab implements a distributed e-commerce system using microservices architecture to demonstrate fundamental principles of distributed systems. The application consists of three distinct microservices communicating through various patterns and deployed on Kubernetes. The technologies chosen include Java 17 with Spring Boot, Docker for containerization, and Kubernetes for orchestration, addressing the module's recommendation to use modern Java versions.

Basic Functionality: The system handles order processing with inventory management and asynchronous notifications, simulating a real-world e-commerce workflow where orders are created, inventory is updated, and customers receive email confirmations.

2 System Design & Setup

2.1 Architecture Diagram



2.2 Service Boundaries

- API Gateway (Port 9000): Single entry point with resilience patterns
- Order Service (Port 8081): Handles order creation and management
- Inventory Service (Port 8082): Manages product stock levels
- Notification Service (Port 8083): Processes email notifications asynchronously

2.3 Docker Containerization

Each service is containerized with optimized Dockerfiles using mvn spring-boot:build-image plugin

```
export DOCKER_ACCOUNT=YOUR_DOCKER_ACCOUNT
export DOCKER_PASSWORD=YOUR_DOCKER_PASSWORD
mvn spring-boot:build-image -DskipTests
```

2.4 Kubernetes Deployment

The system uses Kubernetes manifests including:

- Deployments for database, each microservice and kafka ecosystem
- Services for internal service discovery and communication
- ConfigMaps for application configuration
- Secrets for sensitive data (e.g. Mailtrap credentials)
- PersistentVolumes for database data persistence
- Namespace isolation (microservices-lab)

2.5 Setup and Run Instructions

Detailed guidance and **demos** can be found at readme.md

Prerequisites: Git, Minikube, kubectl, Docker

1) Pull Git repository from https://github.com/zeli8888/COMP41720-Distributed-Systems.git
2) (Optional) Configure your Mailtrap to receive emails from Notification service
    a) register at https://mailtrap.io/
    b) create a new sandbox project
    c) replace mailtrap username and password at ./k8s/secrets.yaml with yours encoded with base64

```
# to encode with base64
echo -n "your_username_or_password" | base64
```

3) Apply Kubernetes configurations:

```
cd lab4/
# create namepsace first to avoid error
kubectl apply -f k8s/namespace.yaml
kubectl apply -f k8s/
```

4) Verify deployment, wait until all servers are ready:

```
kubectl get pod -n microservices-lab
```

5) Start API gateway proxy and send requests:

```
minikube service api-gateway \
-n microservices-lab
```

6) Open a new terminal and Start Kafka UI proxy: (cluster name: anything, bootstrap servers: broker:29092)

```
minikube service kafka-ui \
-n microservices-lab
```

7) Cleanup

```
kubectl delete -f k8s/
```

2.6 Available Requests

2.6.1 *POST /api/inventory*

- Description: Create new inventory entry
- Request

```
POST /api/inventory
Content-Type: application/json
```

- Request Body

```
{
  "skuCode": "SKU001",
  "quantity": 100
}
```

- Response
  - 201 CREATED: "Inventory created successfully"
  - 409 CONFLICT: "Inventory creation failed"

2.6.2 *GET /api/inventory*

- Description: Retrieve inventory levels for multiple SKU codes
- Request

```
GET /api/inventory
Content-Type: application/json
```

- Request Body

```
["SKU001", "SKU002", "SKU003"]
```

- Response
  - 200 OK: Returns list of integers representing quantities

```
[100, 50, 200]
```

2.6.3 *POST /api/order*

- Description: Create a new order
- Request

```
POST /api/order
Content-Type: application/json
```

- Request Body

```
{
  "orderNumber": "ORD-001",
  "skuCode": "SKU001",
  "price": 29.99,
  "quantity": 2,
  "userDetails": {
    "userId": "user123",
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Doe"
  }
}
```

- Response
  - 201 CREATED: "Order placement completed successfully"
  - 409 CONFLICT: "Order placement failed"

2.6.4 *GET /api/order/{userId}*

- Description: Retrieve all orders for a specific user
- Request

```
GET /api/order/user123
```

- Response
  - 200 OK: Returns list of orders

```
[
  {
    "orderNumber": "ORD-001",
    "skuCode": "SKU001",
    "quantity": 2,
    "userId": "user123",
    "price": 29.99
  }
]
```

3 Architectural Analysis & Justification

3.1 Microservice Granularity

3.1.1 *Granularity Disintegrators Considered*

1) Code Volatility: Order processing and inventory management have different change frequencies
2) Scalability Needs: Notification service requires horizontal scaling during peak loads

3) Fault Tolerance: Notification failures shouldn't prevent order creation
4) Technology Heterogeneity: Different database requirements (MongoDB vs PostgreSQL). Document structure fits nested and flexible order items. SQL database offers strong consistency and fast complex query for stock management.

3.1.2 Decision-making Justification

1) Implemented eventual consistency for non-critical data paths (e.g., order notifications via Kafka) to offer higher scalability and failure tolorerance.
2) Maintaining strong consistency for inventory updates through synchronous gRPC calls to prevent overselling and reduce latency.
3) Implemented independent database for Order service and Inventory service to satisfy business needs and ensure loose coupling.

3.1.3 Boundary Rationale

Services were split based on business capabilities rather than technical layers, following Domain-Driven Design principles. Each service owns its data and exposes capabilities through well-defined interfaces.

3.2 Inter-service Communication

3.2.1 Implemented Patterns

1) **Synchronous REST with Resilience**
   - **Usage**: API Gateway → Order/Inventory Services
   - **Trade-offs**
     - **Pros**: Simple, widely understood, easy debugging, HTTP-native tooling
     - **Cons**: Higher latency, temporal coupling, text-based payloads
   - **Justification**:
     - External-facing APIs benefit from REST's ubiquity and tooling ecosystem
     - Resilience4j circuit breakers prevent cascading failures when downstream services are unavailable
     - Retry mechanisms handle transient network issues without burdening clients
     - Timeouts ensure responsive failure recovery in distributed environment
2) **Synchronous gRPC**
   - **Usage**: Order Service → Inventory Service for stock updates
   - **Trade-offs**
     - **Pros**: High performance, strong typing
     - **Cons**: More complex setup, harder to debug, requires protocol buffers
   - **Justification**:
     - Inventory checks are performance-critical path requiring sub-millisecond latency
     - Strong typing prevents data mismatch errors in inventory operations
     - Internal service-to-service communication doesn't require REST's human-readability
3) **Asynchronous Messaging (Kafka)**
   - **Usage**: Order Service → Notification Service
   - **Trade-offs**
     - **Pros**: Decoupling, fault tolerance, scalability, replay capability
     - **Cons**: Eventual consistency, message ordering complexity, operational overhead
   - **Justification**:
     - Email notifications are non-blocking and don't affect core order processing
     - Kafka's persistence ensures no notification loss during service outages
     - Enables future extension (e.g., analytics, audit services) without modifying Order Service
     - Handles notification backlog during peak loads without impacting order placement latency

3.2.2 Architectural Implications

- **Coupling**: Hybrid approach balances loose coupling (Kafka) with strong consistency requirements (gRPC)
- **Latency**: Critical paths optimized with gRPC, while async patterns handle background tasks
- **Error Handling**: Layered strategy - circuit breakers for availability, retries for transient failures
- **Extensibility**: Event-driven architecture allows new services to consume order events without API changes
- **Operational Complexity**: Requires monitoring multiple communication patterns but provides appropriate tools for each use case

3.3 API Gateway Implementation

The API Gateway serves as:

1) Single Entry Point: Unified API surface for clients
2) Request Routing: Intelligent routing to backend services
3) Resilience Enforcement: Centralized circuit breaking and retry logic
4) Cross-cutting Concerns: Future extension for authentication, logging, rate limiting

Benefits: Simplified client interactions, centralized resilience patterns, and separation of operational concerns from business logic.

3.4 Key Architectural Decisions (ADRs)

3.4.1 ADR 1: Polyglot Persistence Strategy

- Context: Different services have varying data storage requirements for orders and inventory.
- Decision: Use MongoDB for Order Service (document storage) and PostgreSQL for Inventory Service (relational integrity).

- Consequences
  - Positive: Optimal data models for each domain, better performance characteristics. Document structure fits nested and flexible order items. SQL database offers strong consistency and fast complex query for stock management.
  - Negative: Increased operational complexity, cross-database joins impossible
  - Trade-offs: Accepted operational complexity for domain-optimized persistence

3.4.2 ADR 2: Hybrid Communication Patterns

- Context: Need to balance performance, reliability, and decoupling across service interactions.
- Decision: Implement three communication patterns: REST (gateway), gRPC (synchronous internal), Kafka (asynchronous events).
- Consequences
  - Positive: Optimized patterns for different interaction types, reduced coupling
  - Negative: Increased implementation complexity, multiple protocols to maintain
  - Trade-offs: Complexity justified by operational requirements and performance benefits

3.4.3 ADR 3: Resilience-First API Gateway

- Context: Prevent cascading failures and improve client experience in distributed system.
- Decision: Implement circuit breaker, retry mechanisms, and timeouts at API Gateway level.
- Consequences
  - Positive: System stability, graceful degradation, better user experience
  - Negative: Additional latency, configuration complexity
  - Trade-offs: Minimal performance impact for significant reliability gains

4 Conclusion

4.0.1 Key Learnings

- Service Boundary Definition: The most challenging aspect was determining optimal service granularity, balancing cohesion against coordination overhead.
- Communication Complexity: Implementing multiple communication patterns revealed the importance of choosing the right pattern for each interaction type.
- Operational Considerations: Container orchestration with Kubernetes provided insights into deployment complexities and resource management in distributed systems.
- Trade-off Management: Every architectural decision involved balancing competing concerns - there are no perfect solutions, only context-appropriate compromises.

4.0.2 Challenges Encountered

- Debugging Distributed Transactions: Tracing requests across service boundaries required sophisticated logging and monitoring
- Configuration Management: Maintaining consistent configurations across multiple environments and services

4.0.3 Insights Gained

This hands-on experience reinforced that microservices architecture is fundamentally about organizational alignment and managing complexity through bounded contexts. The technical implementation, while challenging, serves the higher goal of enabling independent development, deployment, and scaling of system components. The lab successfully demonstrated that thoughtful architectural decisions, guided by clear understanding of trade-offs, can yield systems that are both robust and adaptable to changing requirements.

The implementation proves that modern cloud-native tools like Docker and Kubernetes, combined with appropriate communication patterns, can effectively address the inherent challenges of distributed systems while providing the flexibility needed for evolutionary architecture.