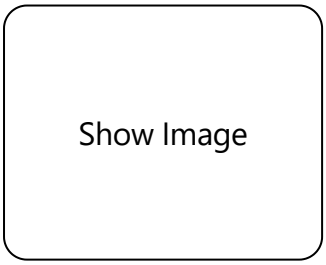
E-Commerce Microservices Application Documentation

Table of Contents

Architecture Overview

This e-commerce application is built using a modern microservices architecture with Spring Boot 3 and Spring Cloud. The system is designed to be scalable, resilient, and maintainable while providing a complete solution for online shopping operations.



The architecture includes:

- **Service Separation**: Independent services for each business domain
- **API Gateway**: Single entry point for all client requests
- **Service Discovery**: Dynamic service registration and discovery with Eureka
- **Centralized Configuration**: Config Server for managing application properties
- **Event-Driven Communication**: Asynchronous messaging via Kafka
- **Distributed Tracing**: Request tracking across services with Zipkin

Key Components

- **Public Network Zone**: Contains the API Gateway accessible to clients
- **Private Network Zone**: Contains all microservices for internal communication
- **Support Services**: Configuration server, service discovery, message broker

Core Services

Config Server

- **Purpose**: Centralizes configuration for all services
- **Technology**: Spring Cloud Config Server
- **Port**: 8888
- **Configuration Storage**: Native file system (YAML files)
- **Features**: Externalized configuration, configuration versioning

Discovery Service (Eureka)

- **Purpose**: Service registration and discovery
- **Technology**: Netflix Eureka Server
- **Port**: 8761
- **Features**: Health monitoring, service registry, load balancing support

API Gateway

- **Purpose**: Request routing, API composition
- **Technology**: Spring Cloud Gateway
- **Port**: 8222
- **Features**: Dynamic routing, request filtering, load balancing

Customer Service

- **Purpose**: Manages customer information
- **Technology**: Spring Boot, Spring Data MongoDB
- **Port**: 8090
- **Database**: MongoDB
- **Features**: Customer registration, profile management, address management
- **API Endpoints**:
  - `POST /api/v1/customers`: Create new customer
  - `PUT /api/v1/customers`: Update customer
  - `GET /api/v1/customers`: Get all customers

- `GET /api/v1/customers/{customer-id}`: Get customer by ID

Product Service

- **Purpose**: Manages product catalog
- **Technology**: Spring Boot, Spring Data JPA, Flyway
- **Port**: 8050
- **Database**: PostgreSQL
- **Features**: Product listing, category management, inventory tracking
- **API Endpoints**:
  - `POST /api/v1/products`: Create new product
  - `GET /api/v1/products`: Get all products
  - `GET /api/v1/products/{product-id}`: Get product by ID
  - `POST /api/v1/products/purchase`: Purchase products

Order Service

- **Purpose**: Handles order processing
- **Technology**: Spring Boot, Spring Data JPA, OpenFeign, Kafka
- **Port**: 8070
- **Database**: PostgreSQL
- **Features**: Order creation, order tracking, order line management
- **API Endpoints**:
  - `POST /api/v1/orders`: Create new order
  - `GET /api/v1/orders`: Get all orders
  - `GET /api/v1/orders/{order-id}`: Get order by ID
  - `GET /api/v1/order-lines/order/{order-id}`: Get order lines by order ID

Payment Service

- **Purpose**: Processes payments
- **Technology**: Spring Boot, Spring Data JPA, Kafka
- **Port**: 8060
- **Database**: PostgreSQL
- **Features**: Payment processing, payment method handling, transaction recording
- **API Endpoints**:
  - `POST /api/v1/payments`: Process payment

Notification Service

- **Purpose**: Sends notifications to customers
- **Technology**: Spring Boot, Spring Data MongoDB, Kafka, JavaMail, Thymeleaf
- **Port**: 8040
- **Database**: MongoDB
- **Features**: Email notifications, template-based messaging, notification history
- **Kafka Topics**:
  - `order-topic`: Order confirmation events
  - `payment-topic`: Payment confirmation events

Communication Patterns

The application implements two primary communication patterns:

1. Synchronous Communication (REST)

- **Implementation**: OpenFeign, RestTemplate
- **Use Cases**:
  - Customer validation during order creation
  - Product availability check during purchase
  - Direct payment processing
- **Benefits**: Immediate response, simple implementation
- **Drawbacks**: Increased coupling, potential for cascading failures

2. Asynchronous Communication (Kafka)

- **Implementation**: Spring Kafka
- **Topics**:
  - `order-topic`: Order confirmations
  - `payment-topic`: Payment confirmations
- **Use Cases**:
  - Order notifications
  - Payment confirmations
- **Benefits**: Loose coupling, improved resilience, better scalability
- **Drawbacks**: Eventual consistency, more complex implementation

Data Storage Strategy

The application uses a polyglot persistence approach:

MongoDB (Document Database)

- **Services**: Customer Service, Notification Service
- **Reasons**:
  - Flexible schema for customer profiles
  - Document-oriented data for notifications
  - Simpler querying for read-heavy operations

PostgreSQL (Relational Database)

- **Services**: Product Service, Order Service, Payment Service
- **Reasons**:
  - ACID compliance for financial transactions
  - Structured relationships between entities (orders, order lines)
  - Strong consistency guarantees

Infrastructure Components

All infrastructure is containerized using Docker and orchestrated with docker-compose:

Databases

- **PostgreSQL**: Relational database for orders, payments, and products
  - Port: 5432
  - Admin Interface: pgAdmin (port 5050)
- **MongoDB**: Document database for customers and notifications
  - Port: 27017
  - Admin Interface: Mongo Express (port 8081)

Message Broker

- **Kafka**: Event streaming platform
  - Port: 9092
  - Dependency: Zookeeper (port 2181)

Observability

- **Zipkin**: Distributed tracing system
  - Port: 9411
  - Features: Trace collection, visualization, latency analysis

Development Tools

- **MailDev**: SMTP testing server
  - UI Port: 1080
  - SMTP Port: 1025

Business Flow

Customer Registration Flow

1. Client sends customer details to API Gateway
2. Gateway routes request to Customer Service
3. Customer Service validates and stores customer data
4. Customer ID is returned to client

Order Creation Flow

1. Client submits order with:
   - Customer ID
   - Product IDs and quantities
   - Payment method
2. API Gateway routes to Order Service
3. Order Service:
   - Validates customer via Customer Service
   - Checks product availability via Product Service
   - Creates order entity with order lines
   - Requests payment via Payment Service
   - Publishes order confirmation event to Kafka
4. Notification Service:
   - Consumes order confirmation event
   - Sends order confirmation email
   - Records notification in database

Payment Processing Flow

1. Order Service initiates payment request
2. Payment Service:
   - Processes payment (simulation)
   - Records payment transaction

- Publishes payment confirmation event to Kafka

3. Notification Service:
    - Consumes payment confirmation event
    - Sends payment confirmation email
    - Records notification in database

Technical Features

Exception Handling

- Centralized exception handlers in each service
- Consistent error response format
- HTTP status code mapping for different error scenarios

Validation

- Bean Validation (Jakarta Validation)
- Custom validation for business rules
- Validation error responses with field-level details

Data Transfer Objects (DTOs)

- Request/Response separation
- Validation annotations on DTOs
- Mappers for entity conversion

Database Migrations

- Flyway for relational database schema evolution
- Version-controlled migrations
- Baseline data seeding

Distributed Tracing

- Micrometer with Brave implementation
- Zipkin for trace collection and visualization
- Consistent trace and span IDs across services

Email Templates

- Thymeleaf for HTML email templates
- Responsive design for all devices

- Customizable templates based on notification type

Deployment Guide

Prerequisites

- Docker and Docker Compose
- Java 17 or later
- Maven

Local Deployment Steps

1. **Clone the repository**

   ```bash
   git clone <repository-url>
   cd ecommerce-microservices
   ```

2. **Start infrastructure components**

   ```bash
   docker-compose up -d
   ```

3. **Build and run services in order**

   ```bash
   # Config Server first
   cd services/config-server
   ./mvnw spring-boot:run

   # Then Discovery Service
   cd ../discovery
   ./mvnw spring-boot:run

   # Then all other services
   ```

Production Considerations

- Use container orchestration (Kubernetes)
- Implement CI/CD pipelines
- Configure proper resource limits
- Set up monitoring and alerting
- Use persistent volume claims for databases

Service Details

Customer Service

Domain Model

- `Customer`: Core entity representing registered users
  - Fields: id, firstname, lastname, email, address
- `Address`: Value object for customer addresses
  - Fields: street, houseNumber, zipCode

API Endpoints

- `POST /api/v1/customers`: Create customer
- `PUT /api/v1/customers`: Update customer
- `GET /api/v1/customers`: Get all customers
- `GET /api/v1/customers/{customer-id}`: Get customer by ID
- `GET /api/v1/customers/exists/{customer-id}`: Check if customer exists
- `DELETE /api/v1/customers/{customer-id}`: Delete customer

Product Service

Domain Model

- `Product`: Core entity representing items for sale
  - Fields: id, name, description, availableQuantity, price, category
- `Category`: Entity for product categorization
  - Fields: id, name, description, products

API Endpoints

- `POST /api/v1/products`: Create product
- `GET /api/v1/products`: Get all products
- `GET /api/v1/products/{product-id}`: Get product by ID
- `POST /api/v1/products/purchase`: Process product purchase

Order Service

Domain Model

- `Order`: Core entity representing customer orders
  - Fields: id, reference, totalAmount, paymentMethod, customerId, orderLines
- `OrderLine`: Entity representing order items

- Fields: id, order, productId, quantity

API Endpoints

- `POST /api/v1/orders`: Create order
- `GET /api/v1/orders`: Get all orders
- `GET /api/v1/orders/{order-id}`: Get order by ID
- `GET /api/v1/order-lines/order/{order-id}`: Get order lines by order ID

Payment Service

Domain Model

- `Payment`: Core entity representing payment transactions
  - Fields: id, amount, paymentMethod, orderId, createdDate, lastModifiedDate

API Endpoints

- `POST /api/v1/payments`: Process payment

Notification Service

Domain Model

- `Notification`: Core entity representing sent notifications
  - Fields: id, type, notificationDate, orderConfirmation, paymentConfirmation

Kafka Consumers

- `payment-topic`: Handles payment confirmation events
- `order-topic`: Handles order confirmation events

Email Templates

- `order-confirmation.html`: Order details template
- `payment-confirmation.html`: Payment success template

Security Considerations

While not fully implemented in the current version, these security aspects should be addressed:

Authentication and Authorization

- OAuth2/JWT-based authentication
- Role-based access control

- API key management for service-to-service communication

Data Protection

- Data encryption at rest
- Transport layer security (TLS)
- PII data handling compliance

API Security

- Rate limiting
- Request validation
- CORS configuration

Monitoring and Observability

The application includes:

Distributed Tracing

- Zipkin for trace collection and visualization
- Micrometer for instrumentation
- Trace context propagation across services

Health Monitoring

- Spring Boot Actuator endpoints
- Health check APIs
- Service registry health status

Future Monitoring Recommendations

- Prometheus for metrics collection
- Grafana for dashboards
- ELK stack for log aggregation

Future Enhancements

Potential improvements for future versions:

Technical Enhancements

- Circuit breaker implementation (Resilience4j)
- CQRS pattern implementation
- Saga pattern for distributed transactions

- API versioning strategy

Business Features

- User authentication and accounts
- Product reviews and ratings
- Order status tracking
- Recommendation engine
- Shopping cart persistence

DevOps Improvements

- CI/CD pipeline setup
- Infrastructure as Code (Terraform)
- Automated testing strategy
- Blue/green deployment support