Product Service Deployment Guide

Deployment Options

The Product Service supports multiple deployment options to accommodate various infrastructure requirements:

1. **Local Development** - Run directly on developer machine
2. **Containerized Development** - Using Docker and Docker Compose
3. **Kubernetes Deployment** - Production-ready orchestration (configuration provided)
4. **Cloud Platform Deployment** - AWS ECS and Azure AKS templates available

Local Deployment

Prerequisites

- Java 17 JDK
- Maven 3.8+
- PostgreSQL 13+ (optional, can use H2 for development)
- Kafka (optional, can be disabled for development)

Steps

1. Clone the repository
2. Configure the application:

```
# Edit application.yml with your local settings
```

3. Build the application:

```
mvn clean package
```

4. Run the application:

```
java -jar target/product-service.jar
```

Docker Deployment

Prerequisites

- Docker 20.10+
- Docker Compose 2.0+

Steps

1. Navigate to the project directory:

```
cd ecommerce-microservices/product-service
```

2. Build and run with Docker Compose:

```
docker-compose up --build
```

3. Access the services:
    - Product Service API: `http://localhost:8080/api`
    - Kafka UI: `http://localhost:8090`
    - PostgreSQL: `localhost:5432`

Docker Compose Components

The `docker-compose.yml` file orchestrates the following services:

1. **Zookeeper**:
    - Image: `confluentinc/cp-zookeeper:7.3.0`
    - Purpose: Manages Kafka cluster state
    - Port: 2181
2. **Kafka**:
    - Image: `confluentinc/cp-kafka:7.3.0`
    - Purpose: Event streaming platform
    - Ports: 9092, 29092
    - Dependencies: Zookeeper
3. **Kafka UI**:
    - Image: `provectuslabs/kafka-ui:latest`
    - Purpose: Web interface for Kafka monitoring
    - Port: 8090
    - Dependencies: Kafka
4. **PostgreSQL**:
    - Image: `postgres:13`

- Purpose: Relational database
- Port: 5432
- Volumes: `postgres_data`

5. **Product Service**:

   - Image: Built from Dockerfile
   - Purpose: Spring Boot microservice
   - Port: 8080
   - Dependencies: PostgreSQL, Kafka

Docker Network

All services are connected to a custom bridge network called `product-network`, allowing containers to communicate using service names as hostnames.

Volume Management

The PostgreSQL data is persisted using a named volume `postgres_data`, ensuring data survival across container restarts.

CI/CD Pipeline Integration

The service includes GitHub Actions workflow configurations for automated CI/CD:

1. **Build and Test**:

   - Triggers on pull requests to main branch
   - Builds the application and runs unit/integration tests
   - Performs static code analysis with SonarQube

2. **Deploy to Development**:

   - Triggers on merges to main branch
   - Builds Docker image
   - Pushes to container registry
   - Deploys to development environment

3. **Deploy to Production**:
    - Triggers on release tags
    - Builds production Docker image
    - Pushes to production container registry
    - Deploys to production environment with approval step

Container Optimization Techniques

1. **Multi-stage Docker builds**:

```
# Build stage
FROM maven:3.9.6-eclipse-temurin-17 as build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline -B
COPY src ./src
RUN mvn package -DskipTests

# Runtime stage
FROM eclipse-temurin:17-jre
COPY --from=build /app/target/product-service.jar product-service.jar
ENTRYPOINT ["java", "-Xms256m", "-Xmx512m", "-jar", "product-service.jar"]
```

2. **JVM Tuning**:
    - Memory allocation settings
    - Garbage collection optimization
    - Container-aware memory limits
3. **Health Checks**:
    - PostgreSQL readiness check
    - Kafka connectivity check
    - Application health probes

Cloud Deployment Considerations

AWS Deployment

- ECS Fargate for containerized deployment
- RDS PostgreSQL for database
- MSK for managed Kafka
- Secrets Manager for credentials
- CloudWatch for monitoring and logging

Azure Deployment

- AKS for Kubernetes orchestration
- Azure Database for PostgreSQL
- Event Hubs for Kafka API compatibility
- Key Vault for secrets management
- Azure Monitor for observability

Google Cloud Deployment

- GKE for Kubernetes orchestration
- Cloud SQL for PostgreSQL
- Pub/Sub for event streaming
- Secret Manager for sensitive data
- Cloud Monitoring for observability

Scaling Considerations

1. **Horizontal Scaling**:
    - Stateless design allows multiple instances
    - Load balancing with sticky sessions not required
    - Database connection pooling configured
2. **Vertical Scaling**:
    - JVM memory allocation configuration
    - Tuned thread pool sizes
    - Optimized database query performance
3. **Database Scaling**:

- Read replicas for query distribution
- Connection pooling configuration
- Index optimization for common queries