- Product Service Configuration Guide
  - Configuration Overview
  - Configuration Files
  - Spring Profiles
  - Environment Variables
  - Configuration Properties
    - Database Configuration
    - Kafka Configuration
    - Server Configuration
    - Actuator Configuration
  - Configuration Classes
  - Conditional Bean Creation
  - Configuration Best Practices


# Product Service Configuration Guide

# Configuration Overview

The Product Service follows the twelve-factor app methodology for configuration, which advocates for strict separation of configuration from code. This approach allows the application to be deployed across different environments without code changes.

# Configuration Files

The service uses YAML configuration files with Spring profiles to manage environment-specific settings:

| File | Purpose |
|---|---|
| `application.yml` | Default configuration for local development |
| `application-postgres.yml` | PostgreSQL-specific configuration |
| `application-docker.yml` | Configuration for containerized deployment |

# Spring Profiles

The application uses Spring profiles to activate environment-specific configurations:

1. **Default Profile**: Used for local development with H2 database
2. **postgres**: Activates PostgreSQL configuration
3. **docker**: Used for containerized deployment with Docker Compose

# Environment Variables

When running in Docker, the following environment variables can be customized:

| Environment Variable | Description | Default Value |
|---|---|---|
| `SPRING_PROFILES_ACTIVE` | Active Spring profiles | `docker` |
| `SPRING_DATASOURCE_URL` | JDBC URL | `jdbc:postgresql://db:5432/product_db` |
| `SPRING_DATASOURCE_USERNAME` | Database username | `postgres` |
| `SPRING_DATASOURCE_PASSWORD` | Database password | `postgres` |
| `SPRING_KAFKA_BOOTSTRAP-SERVERS` | Kafka server address | `kafka:9092` |
| `SPRING_KAFKA_ENABLED` | Enable/disable Kafka | `true` |

# Configuration Properties

## Database Configuration

```
spring:
  datasource:
    url: jdbc:postgresql://db:5432/product_db
    username: postgres
    password: postgres
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: update
    properties:
```

```
    hibernate:
      dialect: org.hibernate.dialect.PostgreSQLDialect
```

# Kafka Configuration

```
spring:
  kafka:
    enabled: true
    bootstrap-servers: kafka:9092
    topics:
      product-created: product-created
      product-updated: product-updated
      product-deleted: product-deleted
      inventory-updated: inventory-updated
      product-events: product-events
      product-analytics: product-analytics
    producer:
      client-id: product-service-producer
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
      properties:
        spring.json.trusted.packages:
com.ecommerce.product.event,com.ecommerce.product.kafka.event
        spring.json.type.mapping: product-
event:com.ecommerce.product.event.ProductEvent,enhanced-
event:com.ecommerce.product.kafka.event.EnhancedProductEvent
      acks: all
      retries: 3
      batch-size: 16384
      buffer-memory: 33554432
      linger-ms: 10
      compression-type: snappy
```

# Server Configuration

```
server:
  port: 8080
  servlet:
    context-path: /api
```

# Actuator Configuration

```
management:
  endpoints:
    web:
      exposure:
```

```
        include: health,info,metrics,prometheus
    endpoint:
      health:
        show-details: always
        probes:
          enabled: true
        group:
          readiness:
            include: db, kafka
```

# Configuration Classes

The application uses several Java configuration classes to manage beans and component setup:

| Class | Purpose |
|---|---|
| `KafkaConfig` | Configures Kafka producers and topics |
| `KafkaProducerConfig` | Enhanced Kafka producer configuration for specific event types |
| `KafkaTopicConfig` | Topic configuration and creation |
| `EnhancedKafkaConfig` | Advanced Kafka configuration with additional features |
| `DatabaseConfig` | Database connection configuration |

# Conditional Bean Creation

The application uses Spring's conditional bean creation to selectively activate components based on environment:

```
@Configuration
@Conditional(KafkaCondition.class)
public class KafkaConfig {
    // Kafka configuration
}
```

The `KafkaCondition` class checks the `spring.kafka.enabled` property to determine whether Kafka beans should be created:

```
public class KafkaCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        String kafkaEnabled =
```

```
        context.getEnvironment().getProperty("spring.kafka.enabled");
        return kafkaEnabled == null || Boolean.parseBoolean(kafkaEnabled);
    }
}
```

# Configuration Best Practices

1. **Never hard-code credentials** - Use environment variables for all sensitive data
2. **Use profiles for different environments** - Dev, test, staging, production
3. **Keep configuration DRY** - Use YAML anchors and aliases for repeated values
4. **Validate configuration at startup** - Fail fast if required properties are missing
5. **Document all configuration options** - Comment all properties for clarity