Spring Boot Messaging

In diesem Modul

- Warum Messaging? Modelle (Queue vs. Topic) und typische Use Cases
- JMS mit Spring: Producer/Listener, Message Converter
- AMQP/RabbitMQ: Exchanges/Bindings, Producer/Consumer, DLX/DLQ
- Kafka: Topics/Partitionen, Producer/Consumer Groups, Serdes, Fehlerbehandlung
- Reliability: Acks, Confirms, Retry/Backoff, Dead Letter
- Spring Cloud Stream
- Schema Registry (Avro, Protobuf)
- Saga Pattern & Distributed Transactions
- Transactional Outbox (Polling & CDC/Debezium)
- Idempotenz**
- Übungen

Wiederholung: Warum Messaging?

- **Asynchrone Kommunikation:** Sender und Empfänger müssen nicht gleichzeitig verfügbar sein.
- **Entkopplung:** Services kennen sich nicht direkt, kommunizieren über Nachrichtenkanäle.
- **Resilienz:** Bei Ausfall eines Empfängers gehen Nachrichten nicht verloren (werden gepuffert).
- **Skalierbarkeit:** Einfaches Hinzufügen weiterer Consumer für erhöhten Durchsatz.
- **Event-Driven Architectures (EDA):** Basis für moderne verteilte Systeme.

Wiederholung: Messaging-Modelle

1. **Point-to-Point (Queues)**
    - Nachricht wird an eine Queue gesendet.
    - **Nur ein Consumer** empfängt und verarbeitet die Nachricht.
    - Ideal für Work-Distribution und Lastverteilung.
2. **Publish/Subscribe (Topics / Exchanges)**
    - Nachricht wird an ein Topic (oder Exchange) gesendet.
    - **Alle Subscriber**, die das Topic abonniert haben, erhalten eine Kopie der Nachricht.
    - Ideal für Benachrichtigungen und Event-Broadcasting.

Wann nutzt man Messaging?

- **Bestellabwicklung:** Bestellung aufgeben (async zu Payment, Shipping, Notification).
- **Benachrichtigungen:** E-Mails, SMS, Push-Nachrichten versenden.
- **Daten-Integration:** Synchronisierung von Daten zwischen Systemen.
- **Batch-Verarbeitung:** Lange laufende Aufgaben auslagern.
- **Circuit Breaker / Bulkhead Pattern:** Erhöhung der Systemstabilität.

JMS (Java Message Service) in Spring Boot

Die JMS Spezifikation

- JMS ist eine Standard-API für Messaging in Java.
- Definiert gemeinsame Konzepte: `ConnectionFactory`, `Connection`, `Session`, `MessageProducer`, `MessageConsumer`, `Queue`, `Topic`, `Message`.
- Unabhängig vom konkreten Messaging-Anbieter (ActiveMQ, IBM MQ, TIBCO EMS).

Spring JMS mit ActiveMQ (Beispiel-Broker)

Dependency: `spring-boot-starter-activemq`

Nachrichten Senden (`JmsTemplate`)

```
@Service
public class OrderProducer {

    private final JmsTemplate jmsTemplate;

    public OrderProducer(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void sendOrder(Order order) {
        System.out.println("Sending order: " + order.getId());
        // Konvertiert das Objekt automatisch in eine JMS Message (z.B. TextMessage, ObjectMessage)
        jmsTemplate.convertAndSend("orderQueue", order);
    }

    public void sendOrderStatus(String status) {
        System.out.println("Sending status update: " + status);
        jmsTemplate.convertAndSend("orderTopic", status);
    }
}
```

Nachrichten Empfangen (`@JmsListener`)

```
@Component
public class OrderConsumer {

    @JmsListener(destination = "orderQueue")
    public void receiveOrder(Order order) {
        System.out.println("Received order: " + order.getId() + " - Processing...");
        // Hier: Logik zur Verarbeitung der Bestellung
    }

    @JmsListener(destination = "orderTopic", containerFactory = "jmsTopicFactory")
    public void receiveOrderStatus(String status) {
        System.out.println("Received status update on topic: " + status);
    }
}
```

Message Converters

- Wandeln Java-Objekte in `javax.jms.Message` und umgekehrt.
- Spring Boot konfiguriert standardmäßig den `MappingJackson2MessageConverter` für JSON.

```
@Configuration
public class JmsConfig {

    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT); // JSON als TextMessage
        converter.setTypeIdPropertyName("_type"); // Typ-Information für Deserialisierung
        return converter;
    }
}
```

Idempotenz

- Wichtig, da Nachrichten in verteilten Systemen **mehrfach zugestellt** werden können ("at-least-once" Delivery).
- Eine Operation ist idempotent, wenn sie mehrmals ausgeführt werden kann, ohne zusätzliche Seiteneffekte zu erzeugen.
- **Strategien:**
  - Eindeutige Message-ID verfolgen.
  - Status-Management (nur bei Status "pending" verarbeiten).
  - Database Unique Constraints.

AMQP in Spring Boot

Das AMQP-Modell

- Ein offener Standard für Messaging.
- Flexibler und mächtiger als JMS, da das Routing-Modell entkoppelt ist.
- Wichtige Konzepte:
  - **Producer:** Sendet Nachrichten.
  - **Exchange:** Empfängt Nachrichten vom Producer und leitet sie an Queues weiter.
  - **Binding:** Eine Regel, die eine Queue an einen Exchange bindet.
  - **Queue:** Speichert Nachrichten, bis sie von einem Consumer abgeholt werden.
  - **Consumer:** Empfängt Nachrichten von einer Queue.

Exchange Types

1. **Direct Exchange:**
   - Nachricht geht an Queues, deren Binding Key *exakt* dem Routing Key der Nachricht entspricht.
   - Ideal für 1:1 oder 1:N Weiterleitung, wenn der Key bekannt ist.
2. **Topic Exchange:**
   - Nachricht geht an Queues, deren Binding Key einem Wildcard-Muster des Routing Keys entspricht.
   - `*`: Ersetzt genau ein Wort.
   - `#`: Ersetzt null oder mehr Worte.
   - Ideal für Pub/Sub mit feingranularer Filterung.

1. **Fanout Exchange:**
    - Nachricht geht an *alle* Queues, die an diesen Exchange gebunden sind (Routing Key wird ignoriert).
    - Ideal für Broadcasting.
2. **Headers Exchange:**
    - Leitet basierend auf den Headern der Nachricht weiter (seltener verwendet).

Spring AMQP mit RabbitMQ

Dependency

`spring-boot-starter-amqp`

Konfiguration (Minimal)

```yaml
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

Nachrichten Senden (RabbitTemplate)

```java
@Service
public class MessageProducer {

    private final RabbitTemplate rabbitTemplate;

    public MessageProducer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    public void sendMessage(String exchange, String routingKey, Object message) {
        System.out.println("Sending to exchange " + exchange + " with routingKey " + routingKey);
        rabbitTemplate.convertAndSend(exchange, routingKey, message);
    }

    public void publishEvent(Object event) {
        // Beispiel: Fanout Exchange für Events
        rabbitTemplate.convertAndSend("events.fanout", "", event);
    }
}
```

Nachrichten Empfangen (@RabbitListener)

```java
@Component
public class MessageConsumer {

    @RabbitListener(queues = "myQueue")
    public void receiveMessage(String message) {
        System.out.println("Received from myQueue: " + message);
    }

    @RabbitListener(queues = "logQueue")
    public void receiveLog(LogMessage log) {
        System.out.println("Received Log: " + log.getLevel() + " - " + log.getContent());
        // Hier könnte eine manuelle Acknowledge-Logik stattfinden
        // channel.basicAck(deliveryTag, false);
    }
}
```

Automatische Erstellung von Exchanges, Queues und Bindings

Spring AMQP kann diese bei Anwendungsstart automatisch erstellen.

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue myQueue() {
        return new Queue("myQueue", true); // Name, durable
    }

    @Bean
    public TopicExchange topicExchange() {
        return new TopicExchange("logExchange");
    }

    @Bean
    public Binding binding(Queue myQueue, TopicExchange topicExchange) {
        return BindingBuilder.bind(myQueue).to(topicExchange).with("*.critical.#"); // Routing Key Muster
    }

    @Bean // Fanout Exchange für Events
    public FanoutExchange eventsFanoutExchange() {
        return new FanoutExchange("events.fanout");
    }
}
```

Reliable Messaging & Dead-Letter Queues

Publisher Confirms & Returns

- **Confirms:** Der Broker bestätigt dem Publisher, dass er die Nachricht erhalten hat.
- **Returns:** Der Broker benachrichtigt den Publisher, wenn eine Nachricht an keinen Consumer zugestellt werden konnte.
- Wichtig für "at-least-once" oder "exactly-once" Semantik (mit Idempotenz).

```
// Konfiguration im RabbitTemplate
// rabbitTemplate.setConfirmCallback(...)
// rabbitTemplate.setReturnCallback(...)
```

Consumer Acknowledgements

Wie ein Consumer dem Broker mitteilt, dass die Nachricht erfolgreich verarbeitet wurde.

1. `AUTO` **(Default in Spring Boot):** Automatisch bei erfolgreicher Methodenausführung.
2. `MANUAL`**:** Consumer muss explizit `channel.basicAck()` oder `channel.basicNack()` aufrufen.
    - Wichtig bei komplexer Verarbeitung, die fehlschlagen könnte.

Dead-Letter Exchanges (DLX)

- Nachrichten, die nicht verarbeitet werden können (z.B. wegen Exceptions, NACKs, TTL-Ablauf), werden an einen speziellen Exchange (DLX) gesendet.
- Von dort können sie in eine **Dead-Letter Queue (DLQ)** geleitet werden.
- Wichtig für Fehlerbehandlung und Auditing.

Konfiguration einer Queue mit DLX

```java
@Bean
public Queue processingQueue() {
    return QueueBuilder.durable("processingQueue")
            .withArgument("x-dead-letter-exchange", "dlxExchange")
            .withArgument("x-dead-letter-routing-key", "processing.dlq")
            .build();
}

@Bean
public DirectExchange dlxExchange() {
    return new DirectExchange("dlxExchange");
}

@Bean
public Queue dlq() {
    return new Queue("processing.dlq");
}

@Bean
public Binding dlqBinding(Queue dlq, DirectExchange dlxExchange) {
    return BindingBuilder.bind(dlq).to(dlxExchange).with("processing.dlq");
}
```

Spring Boot und Kafka

Die "Log-zentrierte" Architektur

- Kafka ist ein **verteiltes Streaming-Plattform**, kein klassischer Message Broker.
- Speichert Nachrichten in einem **Commit Log** (Topic).
- Nachrichten werden nicht "konsumiert" und gelöscht, sondern bleiben für eine konfigurierbare Zeit erhalten.

Kafka Kernkonzepte

- **Broker:** Server, der Topics verwaltet.
- **Topic:** Logischer Kanal für Nachrichten.
- **Partition:** Ein Topic ist in Partitionen unterteilt (Skalierung, Parallelisierung).
- **Producer:** Schreibt Nachrichten in Topics/Partitionen.
- **Consumer:** Liest Nachrichten aus Topics/Partitionen.
- **Consumer Group:** Eine Gruppe von Consumern, die gemeinsam ein Topic verarbeitet. Jede Nachricht in einer Partition wird nur an *einen* Consumer *innerhalb der Gruppe* zugestellt.

Apache Kafka in Spring

Dependency

`spring-kafka`

Konfiguration (Minimal)

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092 # Adressen der Kafka Broker
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.springframework.kafka.support.serializer.JsonSerializer
    consumer:
      group-id: my-service-group
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
      auto-offset-reset: latest # Wo starten, wenn keine Offset gefunden
```

Nachrichten Senden (KafkaTemplate)

```java
@Service
public class EventProducer {

    private final KafkaTemplate<String, Object> kafkaTemplate;

    public EventProducer(KafkaTemplate<String, Object> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void publishUserCreatedEvent(UserCreatedEvent event) {
        System.out.println("Publishing UserCreatedEvent: " + event.getUserId());
        // Key wird für Partitioning genutzt (alle Nachrichten mit gleichem Key landen in gleicher Partition)
        kafkaTemplate.send("user-events-topic", event.getUserId().toString(), event);
    }
}
```

Nachrichten Empfangen (@KafkaListener)

```
@Component
public class UserEventListener {

    @KafkaListener(topics = "user-events-topic", groupId = "user-processor-group")
    public void listen(UserCreatedEvent event, @Header(KafkaHeaders.RECEIVED_PARTITION) int partition) {
        System.out.println("Received UserCreatedEvent for user " + event.getUserId() +
                           " from partition " + partition);
        // ... Logik zur Verarbeitung des Events
    }
}
```

Serde (Serializer/Deserializer)

- Kafka Nachrichten sind Byte-Arrays.
- Producer muss Objekte serialisieren, Consumer deserialisieren.
- Spring Kafka bietet `JsonSerializer` / `JsonDeserializer` für JSON.

Fehlerbehandlung

- **Consumer Group Offsets:** Kafka merkt sich pro Consumer Group den letzten verarbeiteten Offset.
- **Retry-Mechanismen:** Bei Fehlern die Nachricht erneut versuchen.
- **Dead-Letter Topics (DLT):** Nachrichten, die dauerhaft nicht verarbeitet werden können, an ein spezielles Error-Topic senden.
    - Spring Kafka bietet `DeadLetterPublishingRecoverer`.

Spring Cloud Stream

Was ist Spring Cloud Stream?

Eine **Abstraktionsschicht** über Messaging-Systeme (Kafka, RabbitMQ, etc.).

- **Binder:** Adapter für verschiedene Broker (Kafka Binder, Rabbit Binder, etc.)
- **Functional Programming Model:** Producer/Consumer als `Supplier` / `Consumer` / `Function`
- **Broker-Unabhängigkeit:** Code bleibt gleich, nur Konfiguration ändert sich

Warum Spring Cloud Stream?

Aspekt	Direkt (Spring Kafka/AMQP)	Spring Cloud Stream
Boilerplate	Mehr	Weniger
Broker-Wechsel	Code-Änderung	Nur Config
Testing	Aufwändiger	Test Binder verfügbar
Lernkurve	Niedriger	Höher (Abstraktion)

Empfehlung: Cloud Stream für Multi-Broker oder wenn Portabilität wichtig ist.

Dependency

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream</artifactId>
</dependency>
<!-- Binder für Kafka -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

Functional Model: Consumer

```
@Configuration
public class StreamConfig {

    @Bean
    public Consumer<OrderCreatedEvent> orderProcessor() {
        return event -> {
            System.out.println("Processing order: " + event.getOrderId());
            // Business-Logik hier
        };
    }
}
```

Konfiguration:

```
spring.cloud.stream.bindings.orderProcessor-in-0.destination: order-events
```

Functional Model: Supplier (Producer)

```java
@Bean
public Supplier<Flux<OrderStatusEvent>> orderStatusProducer() {
    return () -> Flux.interval(Duration.ofSeconds(5))
        .map(i -> new OrderStatusEvent("order-" + i, "SHIPPED"));
}
```

```
spring.cloud.stream.bindings.orderStatusProducer-out-0.destination: order-status
```

Oder imperativ mit `StreamBridge`:

```java
@Autowired StreamBridge streamBridge;

public void sendEvent(OrderCreatedEvent event) {
    streamBridge.send("order-events", event);
}
```

Functional Model: Function (Processor)

Empfängt Input, transformiert, sendet Output.

```
@Bean
public Function<OrderCreatedEvent, OrderEnrichedEvent> enrichOrder() {
    return event -> {
        // Anreicherung mit zusätzlichen Daten
        return new OrderEnrichedEvent(
            event.getOrderId(),
            event.getAmount(),
            calculateTax(event.getAmount())
        );
    };
}
```

```
spring.cloud.stream:
  bindings:
    enrichOrder-in-0.destination: raw-orders
    enrichOrder-out-0.destination: enriched-orders
```

Schema Registry

Das Problem: Schema Evolution

Nachrichten-Schemas ändern sich über Zeit:

- Neue Felder hinzugefügt
- Felder entfernt oder umbenannt
- Typen geändert

Ohne Schema Registry: Deserialisierung schlägt fehl → Downtime.

Schema Registry Konzept

Ein zentraler Server speichert und versioniert Schemas.

1. **Producer** registriert Schema vor dem Senden
2. **Nachricht** enthält Schema-ID (nicht das ganze Schema)
3. **Consumer** lädt Schema von Registry und deserialisiert

Tools: Confluent Schema Registry, AWS Glue, Apicurio

Unterstützte Formate

Format	Beschreibung	Vorteil
Avro	Binär, Schema-basiert	Kompakt, Schema Evolution
Protobuf	Binär, Google-Standard	Performance, Typed
JSON Schema	Text-basiert	Lesbar, weit verbreitet

Empfehlung: Avro für High-Throughput, JSON Schema für Debugging.

Avro Schema Beispiel

order.avsc:

```
{
  "type": "record",
  "name": "Order",
  "namespace": "com.example.events",
  "fields": [
    {"name": "orderId", "type": "string"},
    {"name": "amount", "type": "double"},
    {"name": "currency", "type": "string", "default": "EUR"},
    {"name": "metadata", "type": ["null", "string"], "default": null}
  ]
}
```

`default` ermöglicht Backward Compatibility!

Spring Kafka mit Schema Registry

```yaml
spring:
  kafka:
    properties:
      schema.registry.url: http://localhost:8081
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: io.confluent.kafka.serializers.KafkaAvroSerializer
    consumer:
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: io.confluent.kafka.serializers.KafkaAvroDeserializer
      properties:
        specific.avro.reader: true
```

Compatibility Modes

Schema Registry prüft Kompatibilität beim Registrieren:

Mode	Beschreibung
BACKWARD	Neue Consumer können alte Nachrichten lesen
FORWARD	Alte Consumer können neue Nachrichten lesen
FULL	Beides (Empfohlen!)
NONE	Keine Prüfung (gefährlich)

```
# Compatibility setzen
curl -X PUT http://localhost:8081/config/order-events-value \
  -H "Content-Type: application/json" \
  -d '{"compatibility": "FULL"}'
```

Saga Pattern & Distributed Transactions

Das Problem: Verteilte Transaktionen

In Microservices können wir keine klassischen ACID-Transaktionen über Service-Grenzen nutzen.

- **JTA/XA:** Funktioniert nicht gut mit REST (zustandslos, Blocking, keine Protokoll-Unterstützung)
- **Alternative:** Das **Saga Pattern** – eine Folge von lokalen Transaktionen mit Kompensationslogik

> *Saga ist tatsächlich kein Akronym. Es steht einfach nur für eine lange Geschichte.*

Saga-Ansatz 1: Choreography (Event-Driven)

Jeder Service entscheidet selbst, was zu tun ist. Es gibt keinen zentralen Koordinator.

- **Ablauf:** `OrderService` → Event: `OrderCreated` → `InventoryService` → Event: `GoodsReserved` → `PaymentService`
- *Pro:* Lose Kopplung, keine zentrale Logik
- *Con:* Unübersichtlich ("Wer hört auf wen?"), zyklische Abhängigkeiten schwer zu erkennen

Passt gut zu: Kafka, RabbitMQ mit Topics/Fanout

Saga-Ansatz 2: Orchestration (Command-Driven)

Ein zentraler "Conductor" (Klasse oder Service) kennt den gesamten Ablauf und sagt den Teilnehmern, was sie tun sollen.

- **Ablauf:** Orchestrator ruft `Inventory.reserve()` auf. Bei Erfolg ruft er `Payment.charge()` auf.
- *Pro:* Klarer Ablauf, einfache Fehlerbehandlung, zentraler Zustand
- *Con:* Orchestrator kann zum "Gott-Service" werden (zu viel Logik)

Tools: Camunda, Temporal, eigene State Machine

Saga Orchestration: Naive Implementierung

```
@Service
public class OrderSagaOrchestrator {
    @Autowired private OrderRepository orderRepo;
    @Autowired private RestClient inventoryClient;
    @Autowired private RestClient paymentClient;

    public void placeOrder(Order order) {
        orderRepo.save(order); // 1. Local TX

        try {
            // 2. Remote Steps (Commands)
            inventoryClient.post().uri("/reserve").body(order).retrieve();
            paymentClient.post().uri("/charge").body(order).retrieve();
            order.setStatus(OrderStatus.CONFIRMED);
            orderRepo.save(order);
        } catch (Exception e) {
            // KOMPENSATION - Problem: Was wenn dieser Call fehlschlägt?
            inventoryClient.post().uri("/release").body(order).retrieve();
            order.setStatus(OrderStatus.FAILED);
            orderRepo.save(order);
        }
    }
}
```

⚠️ **Achtung:** Fragil! Bei Crash im `catch`-Block → inkonsistenter Zustand.

Das Dual-Write Problem

Die naive Saga-Implementierung hat ein fundamentales Problem: **Dual Write**.

- Wir schreiben in die **Datenbank** (Order speichern)
- UND senden **Events/HTTP-Calls** (Inventory, Payment)
- Was passiert, wenn das System zwischen diesen Schritten abstürzt?

→ Das **Outbox Pattern** löst dieses Problem elegant.

Transactional Outbox Pattern

Das Outbox Pattern

Problem: Wie garantiere ich, dass sowohl die DB-Änderung als auch das Event veröffentlicht werden?

Lösung: Wir schreiben das Event in eine **Outbox-Tabelle** in derselben Transaktion wie die Geschäftsdaten.

Ablauf

1. Business-Logik speichert Daten + Event in `outbox`-Tabelle (gleiche TX)
2. Ein separater Prozess (Polling oder CDC) liest die Outbox und publiziert Events
3. Nach erfolgreicher Publikation wird der Outbox-Eintrag gelöscht/markiert

Outbox Entity

```java
@Entity
@Table(name = "outbox")
public class OutboxEvent {
    @Id @GeneratedValue
    private Long id;
    private String aggregateType;  // z.B. "Order"
    private String aggregateId;    // z.B. "12345"
    private String eventType;      // z.B. "OrderCreated"

    @Column(columnDefinition = "TEXT")
    private String payload;        // JSON

    private Instant createdAt;
    private boolean published;
}
```

Outbox: Speichern in einer Transaktion

```java
@Service
public class OrderService {

    @Transactional
    public Order createOrder(Order order) {
        // 1. Business-Daten speichern
        Order saved = orderRepository.save(order);

        // 2. Event in Outbox schreiben (gleiche Transaktion!)
        OutboxEvent event = new OutboxEvent();
        event.setAggregateType("Order");
        event.setAggregateId(saved.getId().toString());
        event.setEventType("OrderCreated");
        event.setPayload(objectMapper.writeValueAsString(saved));
        outboxRepository.save(event);

        return saved;
    }
}
```

Outbox: Publisher (Polling-Variante)

```
@Component
public class OutboxPublisher {

    @Scheduled(fixedDelay = 1000)
    @Transactional
    public void publishPendingEvents() {
        List<OutboxEvent> events = outboxRepository.findByPublishedFalse();

        for (OutboxEvent event : events) {
            try {
                kafkaTemplate.send("domain-events", event.getAggregateId(), event.getPayload());
                event.setPublished(true);  // Oder: outboxRepository.delete(event);
            } catch (Exception e) {
                log.warn("Event {} konnte nicht publiziert werden", event.getId());
                // Retry beim nächsten Durchlauf
            }
        }
    }
}
```

Idempotenz

Warum Idempotenz?

In verteilten Systemen können Nachrichten **mehrfach zugestellt** werden ("at-least-once" delivery).

- **Problem:** `releaseInventory()` wird zweimal aufgerufen → Bestand wird doppelt erhöht
- **Lösung:** Idempotency Key tracken

Eine Operation ist **idempotent**, wenn sie mehrmals ausgeführt werden kann, ohne zusätzliche Seiteneffekte zu erzeugen.

Idempotenz: Implementierung

```java
@Service
public class InventoryService {

    @Transactional
    public void releaseInventory(String orderId, String idempotencyKey) {
        // Prüfen, ob diese Operation bereits durchgeführt wurde
        if (processedOperationRepository.existsByKey(idempotencyKey)) {
            log.info("Operation {} bereits verarbeitet, überspringe", idempotencyKey);
            return;
        }

        // Geschäftslogik ausführen
        Inventory inv = inventoryRepository.findByOrderId(orderId);
        inv.release();
        inventoryRepository.save(inv);

        // Operation als verarbeitet markieren
        processedOperationRepository.save(new ProcessedOperation(idempotencyKey));
    }
}
```

Idempotency Key Strategien

Strategie	Beispiel	Vorteil
Message-ID	`msg.getMessageId()`	Broker liefert ID
Composite Key	`orderId + "_release"`	Deterministisch
Client-Generated	UUID im Header	Volle Kontrolle

Wichtig: Der Key muss die *Operation* identifizieren, nicht nur die Nachricht!

Transactional Outbox mit Debezium (CDC)

Von Polling zu CDC

Die Polling-Variante des Outbox Patterns hat Nachteile:

- Latenz (je nach Polling-Intervall)
- Zusätzliche DB-Last

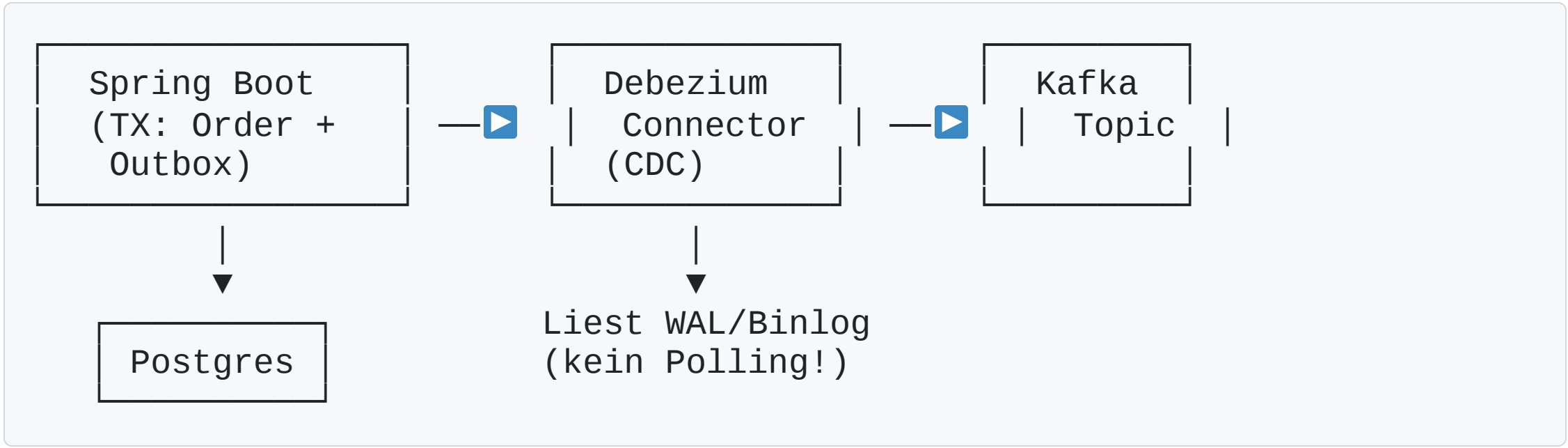
Alternative: Change Data Capture (CDC) mit **Debezium**.

Change Data Capture (CDC)

Statt Polling der Outbox-Tabelle: **Debezium** liest das Datenbank-Log (WAL/Binlog).

- **Vorteil:** Keine zusätzliche Last auf der DB
- **Vorteil:** Near-Realtime (Millisekunden)
- **Nachteil:** Komplexeres Setup (Debezium Connector)

Debezium Outbox Architektur



Debezium Outbox Transformer

Debezium bietet einen speziellen **Outbox Event Router**:

```
{
  "name": "outbox-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "transforms": "outbox",
    "transforms.outbox.type": "io.debezium.transforms.outbox.EventRouter",
    "transforms.outbox.table.field.event.key": "aggregate_id",
    "transforms.outbox.table.field.event.type": "event_type",
    "transforms.outbox.table.field.event.payload": "payload",
    "transforms.outbox.route.topic.replacement": "${routedByValue}"
  }
}
```

Das Outbox-Event wird automatisch ins richtige Topic geroutet!