DB 연동과 RabbitMQ의 Transaction 처리

큐의 트랜잭션 처리

튜토리얼 Step 9. DB 연동 메시지 큐의 Transaction 처리

RabbitMQ는 **트랜잭션 메시징**을 지원하여 메시지 전송, 큐 수신 및 Ack를 처리 할 수 있음. (튜토리얼 Step 7 - `SimpleRabbitListenerContainerFactory` 참고)

- 메시지 전송 중 **데이터 일관성**을 보장하고, 실패 시 메시지를 재전송하거나 복구할 수 있다.

트랜잭션 메시징 개념

RabbitMQ에서의 트랜잭션 메시징은 메시지를 큐에 보내는 동안 문제가 발생하면 메시지 상태를 롤백하거나 커밋하여 메시지가 성공적으로 처리되었음을 보장

트랜잭션 종류

1. **프로듀서 트랜잭션**
    - 메시지를 MQ에 전송하기 전 트랜잭션 시작
    - 메시지를 큐에 게시한 후 트랜잭션 커밋 or 롤백
    - 실패 시 메시지가 큐로 producing 되지 않음
    - 단, 속도 문제가 있으며 비동기라는 특징에 부합하지 않음
2. **컨슈머 트랜잭션**
    - 메시지를 수신한 후 트랜잭션 시작 - RabbitMQ 브로커에 도달했음을 알림
    - 메시지를 처리한 뒤 커밋

- 실패 시 메시지를 다시 처리하거나 DLQ로 이동
- 대규모 메시지 처리 방식에 권장

트랜잭션 개발 플로우

amqp-client에 의존성 (starter-amqp에 포함)

1. `channel.txSelect()`:
   - 트랜잭션 시작. 이후 모든 메시지 전송 작업은 이 트랜잭션 내에서 실행.
2. `channel.txCommit()`:
   - 트랜잭션 커밋. 모든 작업이 성공적으로 수행되었음을 RabbitMQ에 알림.
3. `channel.txRollback()`:
   - 트랜잭션 롤백. 트랜잭션 내 작업을 무효화하고 변경사항을 되돌림.

→ setChannelTransacted(true)와 @Transactional 로 처리 가능

트랜잭션 메시징 Package Structure

```
src/main/java/net/harunote/hellomessagequeue git:[tutorial-step9
hellomessagequeue
 ├── HelloController.java
 ├── HelloMessageQueueApplication.java
 └── step09
       ├── MessageConsumer.java
     ├── MessageProducer.java
       ├── RabbitMQConfig.java
     ├── StockEntity.java
     ├── StockRepository.java
     └── TransactionController.java
```

개발 프로세스

1. application.yml에 DB 연동을 위한 속성 설정 (H2)

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.H2Dialect
        format_sql: true
    show-sql: true
  h2:
    console:
      enabled: true
      path: /h2-console
```

2. build.gradle 에 디펜던시 추가

```
implementation 'org.springframework.boot:spring-boot-starter-
runtimeOnly 'com.h2database:h2' // H2 데이터베이스 의존성
```

3. StockEntity와 Repository 작성

```
import jakarta.persistence.*;
import lombok.ToString;
import java.time.LocalDateTime;

@Entity
@ToString
public class StockEntity {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String userId;
    private int stock;

    private boolean processed;
    private LocalDateTime createdAt;
        private LocalDateTime updatedAt;
    // Getters and Setters
}
```

```
public interface StockRepository extends JpaRepository<StockE
}
```

4. RabbitMQConfig에서 큐 네임 설정 및 RabbitTemplate의 setChannelTransacted 속성 활성화 및 Entity를 메시지로 전송하기 위해 JSON 직렬화 (Jackson2JsonMessageConverter) 를 이용하여 JSON Converter 설정

```
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.ConnectionF
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2Jso
import org.springframework.amqp.support.converter.MessageConv
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {

    @Bean
    public Queue queue() {
        return new Queue("transactionQueue", true);
    }
```

```
    @Bean
    public MessageConverter messageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory co
        RabbitTemplate rabbitTemplate = new RabbitTemplate(co
        rabbitTemplate.setMessageConverter(messageConverter);
        rabbitTemplate.setChannelTransacted(true);  // 트랜잭션

        return rabbitTemplate;
    }

}
```

5. Producer에 @transactional 어노테이션 선언 후 txSelect(), txCommit(), txRollback() 처리
6. Controller에 StockEntity와 Testcase를 받아서 Producer 호출

```
@PostMapping()
public ResponseEntity<String> sendMessage(@RequestBody StockE
                                          @RequestParam(requi
        // do something

}
```

7. cURL 호출 테스트, 파라미터 테스트(fail or success)

```
curl -X POST "http://localhost:8080/api/message?testCase=succ
     -H "Content-Type: application/json" \
     -d '{
           "userId": "testUser",
```

```
            "stock": 100
          }'
```

트랜잭션 메시징의 한계

1. **성능 오버헤드** :
    - 트랜잭션은 추가적인 작업(예: 디스크 동기화)을 요구하므로 성능이 떨어질수있음
    - **대량 메시지 처리**가 필요한 경우 부적합.
2. **복잡성**:
    - 트랜잭션 처리를 잘못 구성하면 메시지가 중복 처리되거나 손실될 가능성 증가.
3. 분산 트랜잭션을 완벽하게 보장하지는 않는다는 것.
   데이터베이스 작업과 메시지 전송 사이의 완벽한 원자성을 위해서는 추가적인 메커니즘(예: 2 Phase Commit 또는 보상 트랜잭션)이 필요함
4. **AMQP Confirm Select 추천**:
    - RabbitMQ에서는 트랜잭션 대신 **Publisher Confirms**(AMQP Confirm Select) 방식이 더 효율적

튜토리얼 Step 10. Publisher Confirms를 이용한 트랜잭션 처리

Publisher Confirms

메시지가 **브로커의 Exchange에 도달했는지** 확인하는 메커니즘.

- 메시지가 RabbitMQ 브로커에 성공적으로 도달했는지 확인하기 위해 사용되는 경량화된 메커니즘으로 서로 다른 이기종간의 신뢰성 보정, 대사 기법인 TCC와 비슷한 방법

이는 트랜잭션보다 성능이 뛰어나며, 메시지가 브로커에 안전하게 저장 되었는지 확인하는 데 적합하고 가볍다는 특징이 있으므로, 시스템 부담이 적고 빠르다.

특징

1. **가볍고 빠름**:
   - 개별 메시지 단위로 전송 후 브로커로부터 확인(acknowledgment)을 받는다
   - 트랜잭션처럼 모든 메시지 작업을 롤백하지 않고, 실패한 메시지만 다시 처리.
2. **응답 처리 방식**:
   - Ack : 브로커가 메시지를 성공적으로 처리했음을 알림.
   - Nack : 브로커가 메시지 처리에 실패했음을 알림.
3. **활용 사례**:
   - 대규모 메시지 전송.
   - 메시지 손실 방지가 중요한 시스템(예: 결제 시스템, 알림 시스템).

application.yml 설정

```
rabbitmq:
  host: localhost
  port: 5672
  username: guestuser
  password: guestuser
  publisher-confirm-type: correlated # Enable Publisher Confir
```

테스트 시나리오

- 정상적인 케이스

```
curl -X POST "http://localhost:8080/api/message?testCase=fals
     -H "Content-Type: application/json" \
     -d '{"userId":"USER1","stock":10}'
```

- 트랜잭션 실패 - RabbitMQ 전송 실패

```
curl -X POST "http://localhost:8080/api/message?testCase=true
     -H "Content-Type: application/json" \
```

```
      -d '{"userId":"USER1","stock":10}'
```

- 트랜잭션 실패 - DB 저장 실패

```
curl -X POST "http://localhost:8080/api/message?testCase=fals
     -H "Content-Type: application/json" \
     -d '{"userId":"","stock":10}'
```

단, 메시지 중복 가능성이 있고 지속성에 대한 비용이 존재할 수 있다.

- Nack 발생 시 재전송 과정에서 메시지가 중복될 수 있음.
- 중복 메시지 처리를 별도로 구현
- PERSISTENT 설정시 약간 성능이 저하될 수 있음
- txSelect 로 트랜잭션을 시작하는 RabbitMQ Transactions 방식 보다 (이 경우는 메시지 원자성을 명시적으로 보장) 메시지를 비동기적으로 전송하며
  RabbitMQ 서버에서 메시지가 정상적으로 저장되었는지 ack/nack로 확인하고 응답 역시 비동기로 수신하기 때문에 처리량이 높다.
- 실제 서버 환경 마다 차이는 있으나 RabbitMQ Transactions 방식보다 두배 이상 전송 성능이 높다.

어떤 상황에서 쓰는가?

- 높은 처리량이 필요한 경우
- 개별 메시지의 신뢰성이 중요한 경우
- 성능이 중요한 대규모 시스템

TCC 기법을 활용한 데이터의 보정(Recognition)과 대사(Reconciliation)

메시지 큐를 가공하여 다른 시스템에 전달 하였을 때, 이는 비동기이기 때문에 트랜잭션으로 묶을수가 없다. 일반적인 케이스에서 물리적으로 코드도 다른 서버에 있고 트랜잭션 처리를 한다는게 큐의 비동기 사상과 맞지 않기 때문에 보통 이 경우에는 양쪽(전송자와 수신자) 서버쪽에

서 데이터가 맞는지 대사 (recognition, data integrity and consistency) 작업을 통해 양쪽의 데이터가 일치하는지 여부를 확인하는 단계를 둔다.

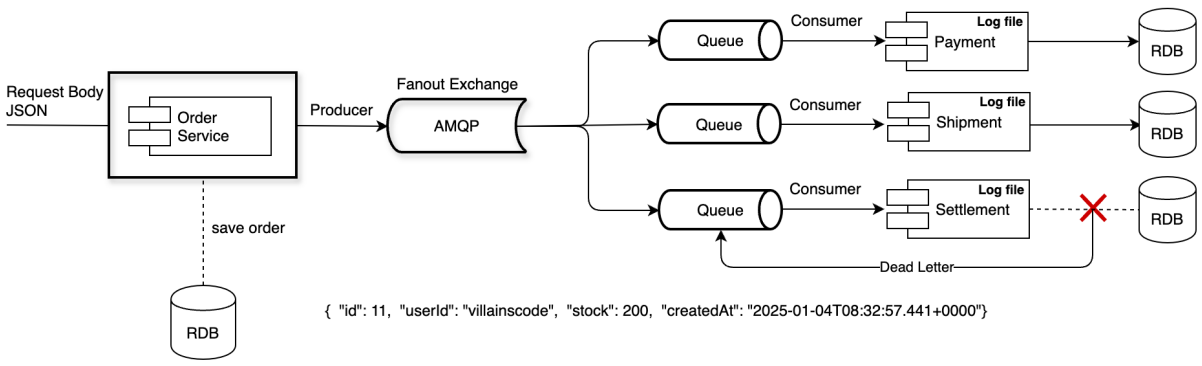
시간 배치작업이나 일별 배치등으로 양측의 데이터를 재확인하는 대사 작업을 한 뒤 틀어지는 데이터가 있다면 재보정을 해주는 후처리를 해주는 방식이다.

이 방식과 비슷하게 분산 서버간의 트랜잭션 처리를 TCC(Try-Confirm/Cancel) 로 만들어서 취소/확인과 확정의 단계를 두는 방식으로 처리하면 상당부분의 트랜잭션은 비교적 어렵지 않게 처리할 수 있다. 취소와 확인, 보정이라는 안정장치와 절차들을 통해 데이터를 확정짓는 패턴인 것이다.

TCC 를 이용한 구현 (Try-Confirm/Cancel) 패턴은 서로 다른 두개의 도메인에 각각 요청한 결과에 대해서, 정상적으로 처리 된 경우의 응답이 오면 둘다 확정 하는 단계를 거치도록 하고 중간에 에러가 발생한 경우 다른 하나에 대해서 취소 처리 프로세스를 호출하는게 패턴의 핵심이다.

1) TCC의 진행 프로세스

- 이 경우 직접적인 트랜잭션 레이어가 아니라 방어/보상 로직을 통해 확정(confirm) 혹은 취소(cancel) 하는 방식이다.
- 전송되는 데이터의 원본 (JSON) 을 저장하고 호출이 정상종료일 경우 확정, 비정상일 경우 취소 처리 한다.
- consumer가 confirm (확정) 혹은 exception 시 cancel 을 한곳의 queue에 발행한다
- 각 경계에서 전달되는 데이터들은 JSON으로 저장하고 이를 이용하여 로직을 처리하고 상태를 확정 처리 한다.
- 타임아웃이나 예외가 발생하면 취소(cancel) 상태를 요청 혹은 Expire 필드를 두어 처리한다.



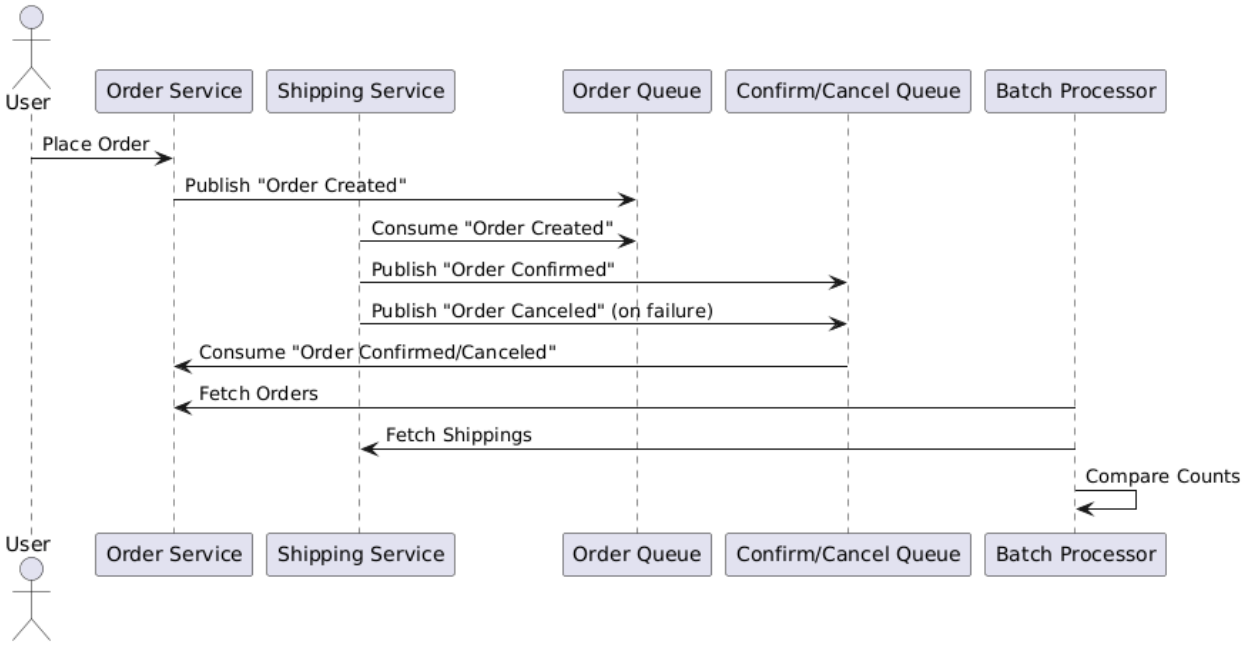
2) 주문/배송 간의 시나리오 예시

- **Order 도메인**: 주문 상태를 초기 상태로 설정하고 메시지를 전송.
- **Consumer(Shipping 도메인)**: 주문 상태를 CONFIRM(처리 성공) 또는 CANCEL(처리 실패)로 변경하여 메시지 발행.
- **최종 큐**: CONFIRM/CANCEL 상태 메시지는 Order가 바라보는 단일 큐로 수신.
- **배치 처리**: Order와 Shipping 도메인에서 시간 단위로 메시지 카운트를 대조하여 보정.

Message 레이어를 통한 Retry와 보정(후처리)

각 도메인으로 연결되는 경계 로직에서 Message Queue로 데이터를 Publish하고 각 도메인은 이 Queue를 바라보고 Message를 Consume 한다.

이 단계에서 Message 솔루션들은 토픽에 대한 손실을 방지하기 위해 자체적인 저장소를 유지해야 한다. 이 저장소를 통해 미 처리된 Message 에 대해서 retry를 시도하여 최종적으로 요청된 토픽이 처리 될 수 있도록 하며, 전송측과 구독측의 데이터 보정을 위해 주기적으로 Sync하는 추가 로직이 존재해야 한다.


User
Order Service
Shipping Service
Order Queue
Confirm/Cancel Queue
Batch Processor
Place Order
Publish "Order Created"
Consume "Order Created"
Publish "Order Confirmed"
Publish "Order Canceled" (on failure)
Consume "Order Confirmed/Canceled"
Fetch Orders
Fetch Shippings
Compare Counts
User
Order Service
Shipping Service
Order Queue
Confirm/Cancel Queue
Batch Processor