Chapter 10

이벤트 중심 애플리케이션과 함수

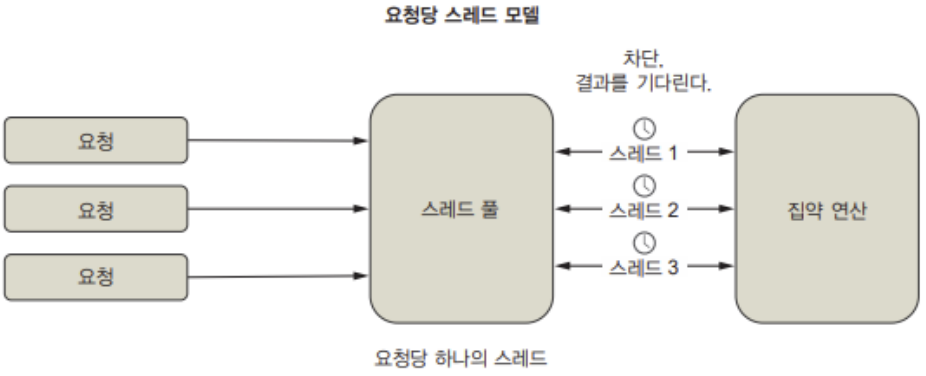


그림 8.1 요청당 스레드 모델에서 각 요청은 전용 스레드에 의해 처리된다.

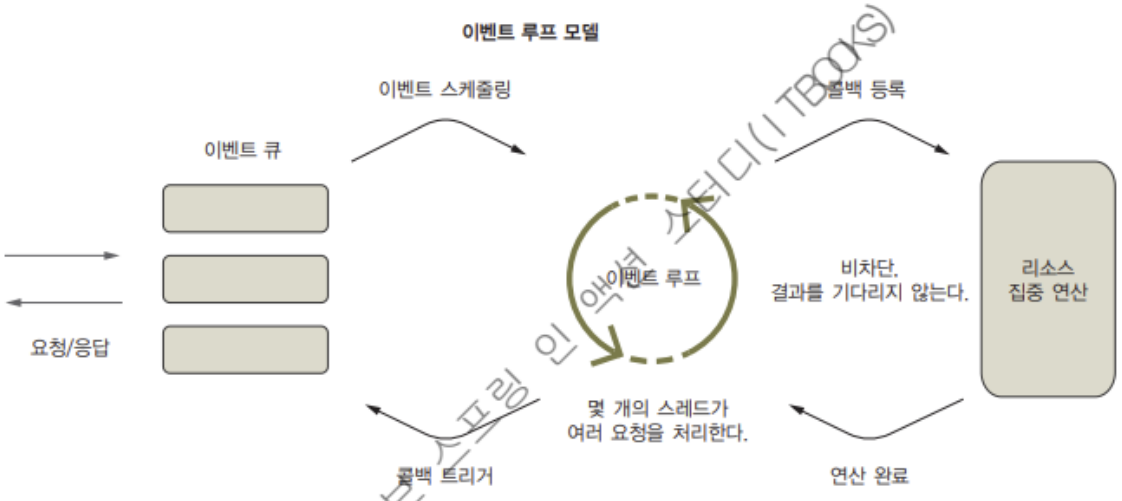


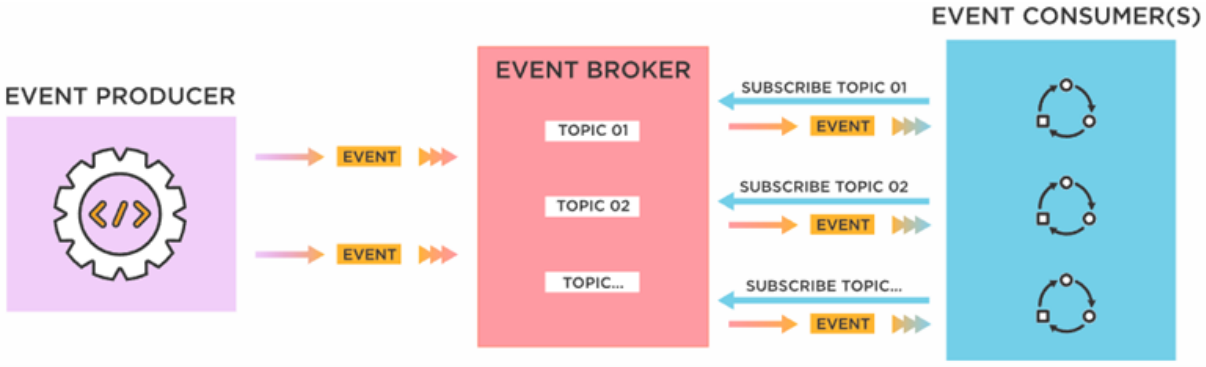
그림 8.2 이벤트 루프 모델에서 스레드는 집중적인 작업을 기다리는 동안 차단되지 않고 다른 요청을 처리할 수 있다.

- 명령형 방식
  - 처리 스레드가 차단되어 I/O 작업에서 응답을 기다린다
- 리액티브 방식

- 스레드가 기다리지 않고, 응답이 오면 사용 가능한 스레드에 의해 비동기적으로 처리된다

다만, 두가지 방식 모두 두 애플리케이션 간의 상호작용은 동기적이다 (무조건 응답이 올 것으로 예상)

이벤트 중심 아키텍처

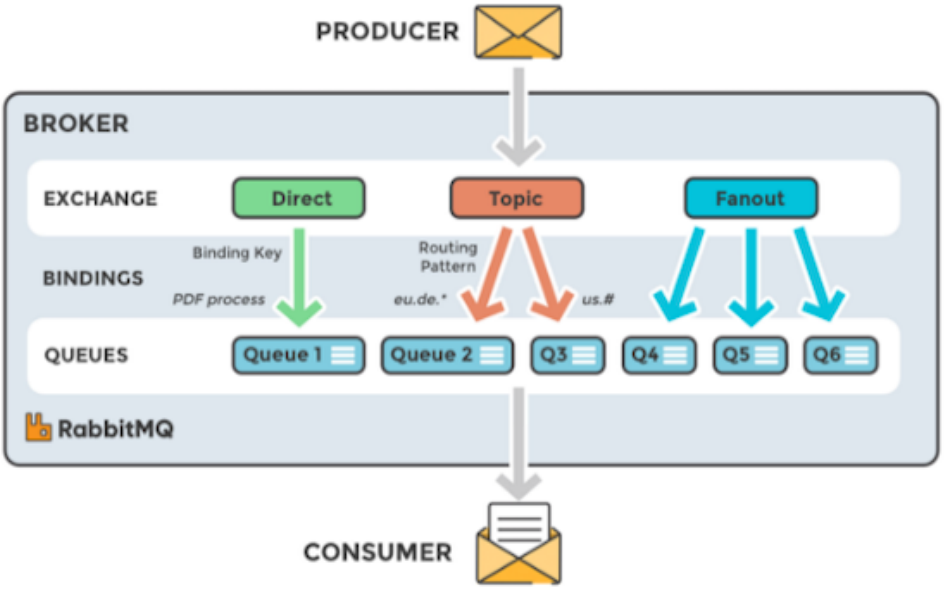


- 이벤트 생산자
  - 이벤트 브로커가 운영하는 채널에 메시지 발행
- 이벤트 소비자
  - 브로커는 이벤트 소비자에게 알리고, 소비자는 조치를 취한다
- 브로커, 채널
  - 메시지를 수집하고 이벤트 소비자에게 라우팅하는 역할 수행

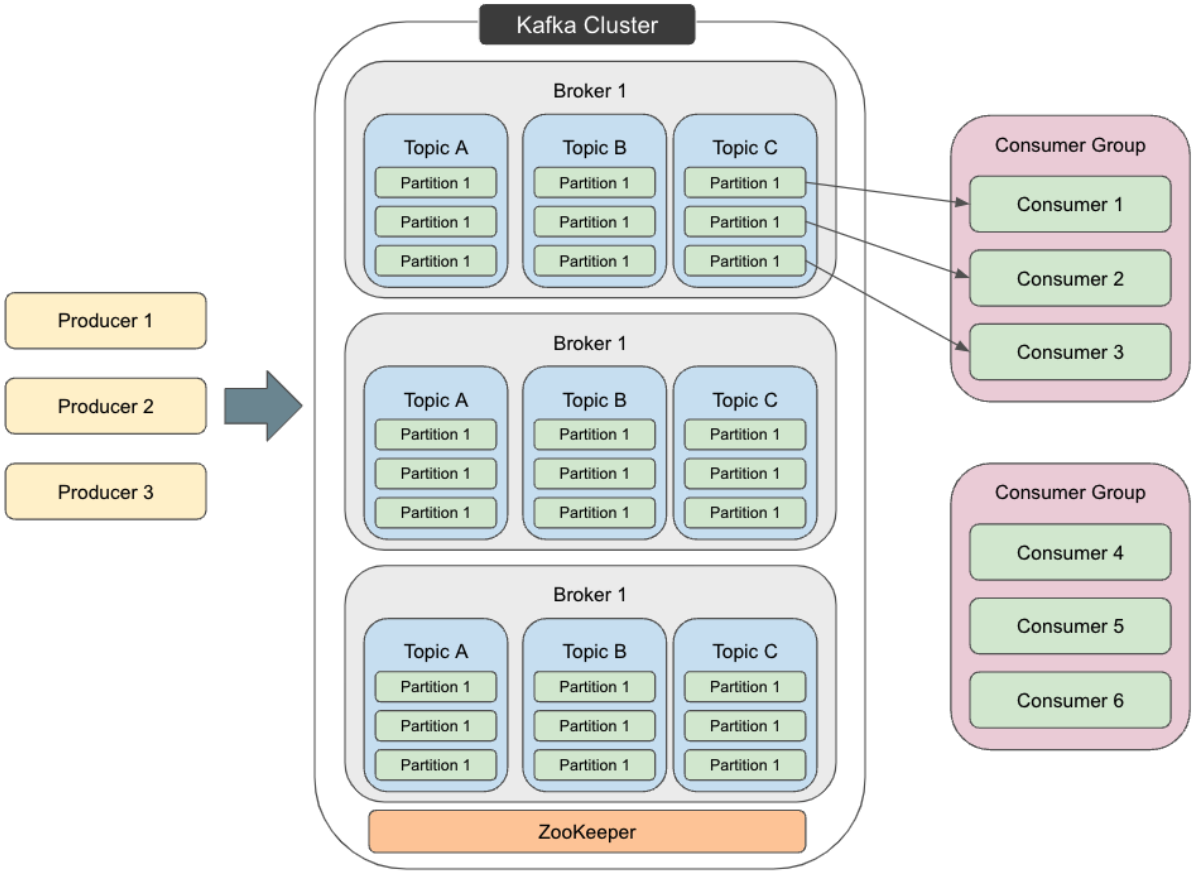
서로를 알지 못한 채 독립적으로 실행한다 → 생산자와 소비자 간의 느슨한 결합

시간적으로 분리 → 소비자는 원하는 때에 언제든지 메시지를 가져와 처리

이벤트 중심 모델



- 발행자/구독자 모델
  - 구독 기반
  - 새로 가입한 소비자는 과거 이벤트를 받지 못한다



- 이벤트 스트리밍 모델
  - 로그에 기록
  - 생산자는 발생 순서대로 이벤트 저장
  - 소비자는 언제든지 구독하고 과거 이벤트를 받을 수 있다

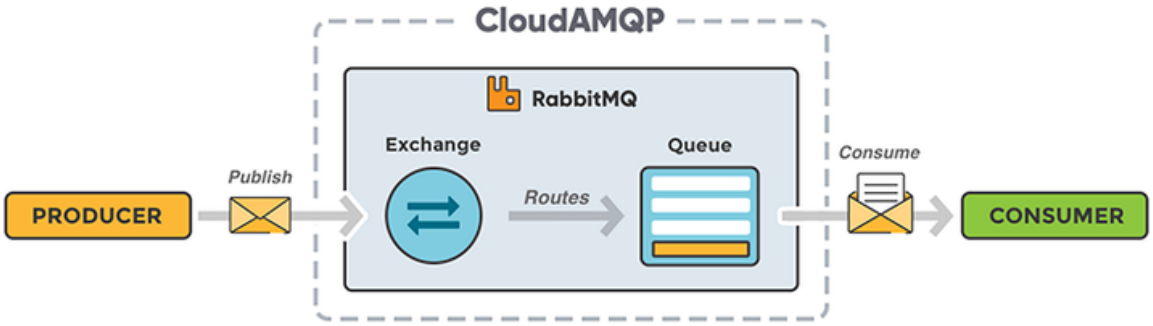
메시지 브로커와 RabbitMQ

발행자/구독자 모델

메시징 시스템에는 두 가지 요소가 필요하다

- 메시지 브로커
- 프로토콜
    - AMQP는 플랫폼 간의 호환성과 안정적인 메시지 전달을 보장

AMQP



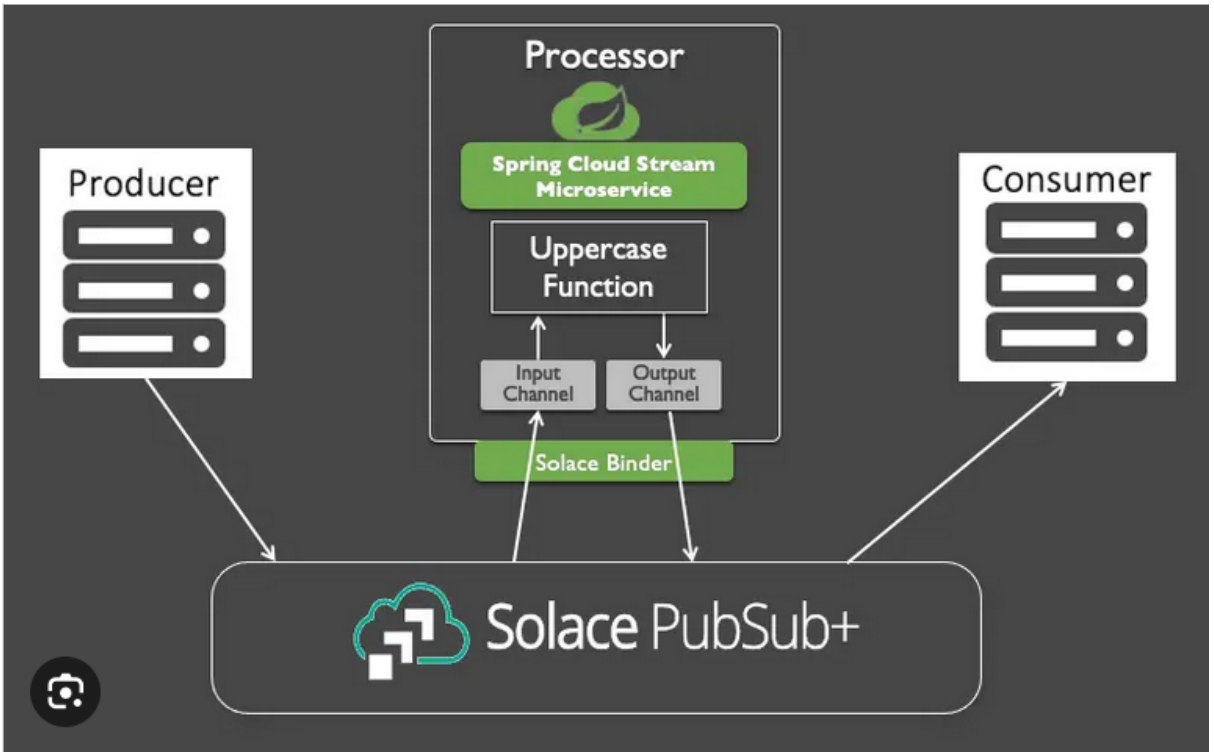
- 생산자
    - 메시지를 보내는 개체
- 소비자
    - 메시지를 받는 개체
- 메시지 브로커
    - 생산자로부터 메시지를 받아 소비자에게 라우팅하는 미들웨어

익스체인지와 큐

- 익스체인지 (카프카 토픽과 유사한 개념)
    - 메시지를 수신하여 이를 적절한 큐로 전달하기 위한 라우팅 로직 관리
    - 유형

        - Direct Exchange
            - 라우팅 키를 기반으로 정확히 일치하는 큐에 메시지 전달
            - 라우팅 키가 “error”인 메시지는 “error” 라우팅 키가 설정된 큐에만 전달
            - 사용 예시 : 에러 로그만 특정 큐로 전달하고자 할 때
        - Fanout Exchange
            - 라우팅 키를 무시하고, 익스체인지에 연결된 모든 큐에 메시지 전달
            - 메시지를 브로드캐스트할 때 유용
            - 사용 예시 : 모든 소비자에게 동일한 정보를 전달 할 때
        - Topic Exchange
            - 라우팅 키에 와일드카드를 사용하여 패턴에 맞는 큐에 메시지 전달
            - `*` 는 단일 단어, `#` 은 여러 단어에 매칭
            - 사용 예시 : 특정 주제에 맞는 소비자에게만 전달 할 때 (`stock.price.*`)
        - Headers Exchange
            - 메시지 헤더의 특정 속성에 따라 큐 결정
            - 라우팅 키 대신 헤더 값을 사용하여 유연한 라우팅 가능
            - 사용 예 : 다양한 속성에 따라 세분화된 큐에 메시지를 전달 할 때
- 큐
    - 메시지를 저장하는 일종의 버퍼 역할
    - 주요 기능
        - 메시지 저장
        - 지속성 설정
            - 시스템 장애나 재시작 시에도 메시지 유지 (`Durable Queue`)
        - TTL
        - 우선순위

스프링 클라우드 함수를 통한 함수



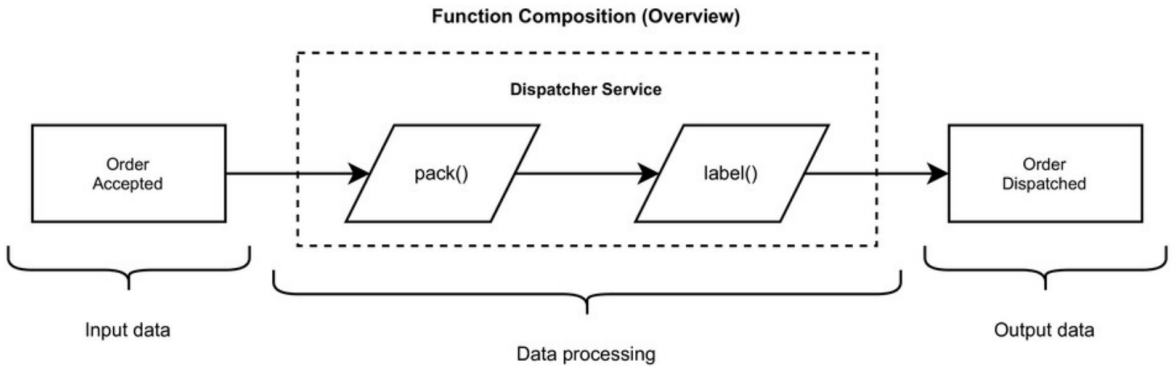
- 공급자
  - 출력만 있고 입력이 없는 함수
- 함수
  - 입력과 출력을 모두 가진다
- 소비자
  - 입력만 있고 출력은 없는 함수

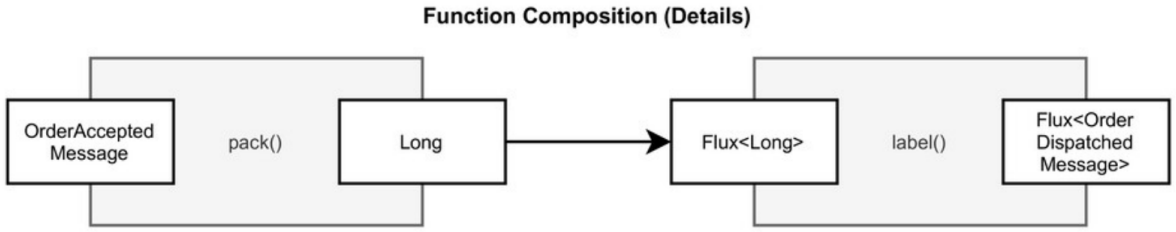
스프링 클라우드 함수의 장점? 사용하는 이유?

1. 함수 중심 개발로 인한 모듈화 및 재사용성 향상
2. 클라우드 간 이식성
   - 클라우드 제공자에 종속적이지 않도록 설계되어 있다

3. 서버리스 환경에 최적화된 함수 실행 방식
    - 초기화 지연
        - 함수가 호출될 때만 인스턴스가 초기화되므로, 불필요한 자원 사용 방지
    - 단일 책임
        - 각 함수는 특정 작업을 수행하도록 단순화되어 효율적으로 실행
    - 비용 절감
4. 다양한 이벤트 소스와의 손쉬운 통합
    - Spring Cloud Stream
    - Srping Cloud Function Adapter
    - 이벤트 기반 처리
5. 비지니스 로직과 인프라 종속성 분리
    - 설정 기반의 바인딩
        - 함수 로직과 인프라 구성 분리
        - 코드 수정 없이 인프라 설정만 변경해도 새로운 이벤트 소스와 통합 가능
    - 클라우드 서비스 종속성 최소화
    - 함수 바인딩 방식
        - 함수 정의는 코드에서 분리되어, 구성 파일로 필요한 함수 연결 정의하여 사용자가 함수 로직을 자유롭게 변경 가능
6. 자동 구성
    - Spring Boot

스프링 클라우드 함수 합성





- 관심사의 분리 원칙에 따라 작은 단위의 함수를 별도로 개발
- 이후 필요에 따라 결합하여 복잡한 작업을 하나의 파이프라인처럼 동작하도록 사용

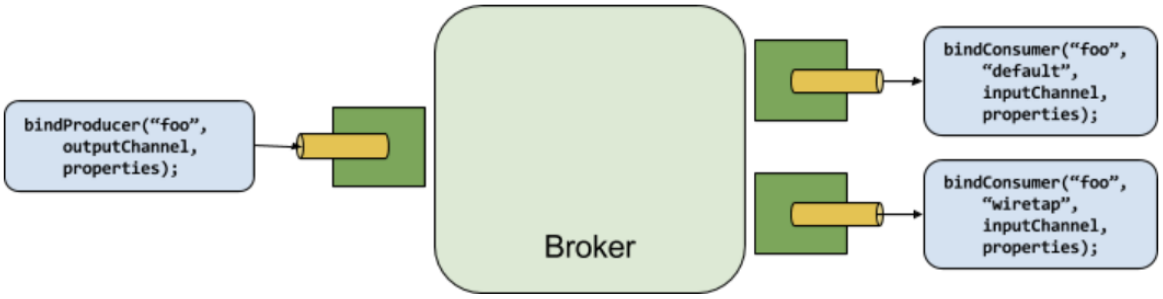
순차적 합성 예시

```
@Bean
public Function<String, String> trimFunction() {
    return input -> input.trim();
}

@Bean
public Function<String, String> uppercaseFunction() {
    return input -> input.toUpperCase();
}

// 함수 합성 설정
spring.cloud.function.definition=trimFunction|uppercaseFuncti
```

스프링 클라우드 스트림과의 연계

```
# 스프링 클라우드 스트림과 연계하여 메시지 처리에 함수 합성 사용
spring.cloud.stream.bindings.input.destination=my-topic
spring.cloud.stream.bindings.output.destination=processed-top
spring.cloud.function.definition=processStep1|processStep2|pr
```

스프링 클라우드 스트림을 통한 메시지 처리



구성 요소

1. 대상 바인더
    - 스프링 클라우드 스트림 애플리케이션을 외부 메시지 시스템과 연결하기 위한 컴포넌트
    - 메시지 브로커에 맞춰 설계된 바인더 라이브러리
    - 각 메시지 브로커에 대한 연결과 통신 로직 처리
    - 추상화 계층 역할을 하여 애플리케이션이 브로커에 대한 구현을 알 필요 없다
2. 대상 바인딩

- 애플리케이션의 생산자나 소비자를 외부 메시지 시스템의 개체(큐, 토픽 등)에 연결하는 구성 요소
- 바인딩을 사용하여 메시지 흐름을 제어, 외부 시스템과의 메시지 전달 설정

3. 메시지
    - 애플리케이션의 생산자와 소비자 간에 교환되는 데이터 구조
    - 메시지 구조
        - Payload
            - 실제 데이터가 포함된 부분
            - JSON, XML, 텍스트 등 다양한 포맷 지원
        - Header
            - 메시지에 대한 메타데이터
            - 라우팅 키, 타임스탬프, 메시지 ID 등 메시지의 처리를 위한 정보 포함

대상 바인딩 네이밍 규칙

- 입력 바인딩 : `<functionName>-in-<index>`
- 출력 바인딩 : `<functionName>-in-<index>`

> 인덱스는 주로 파티션을 사용할 때 유용하며, 파티션이 없을 경우 기본값 0이다.

RabbitMQ와 스프링 클라우드 스트림 통합

```
# RabbitMQ에서 사용할 바인딩 설정 예시
spring.cloud.stream.bindings.input.destination=myQueue-in-0
spring.cloud.stream.bindings.output.destination=myQueue-out-0

# RabbitMQ 바인더 사용 설정
spring.cloud.stream.binders.rabbit.type=rabbit
```

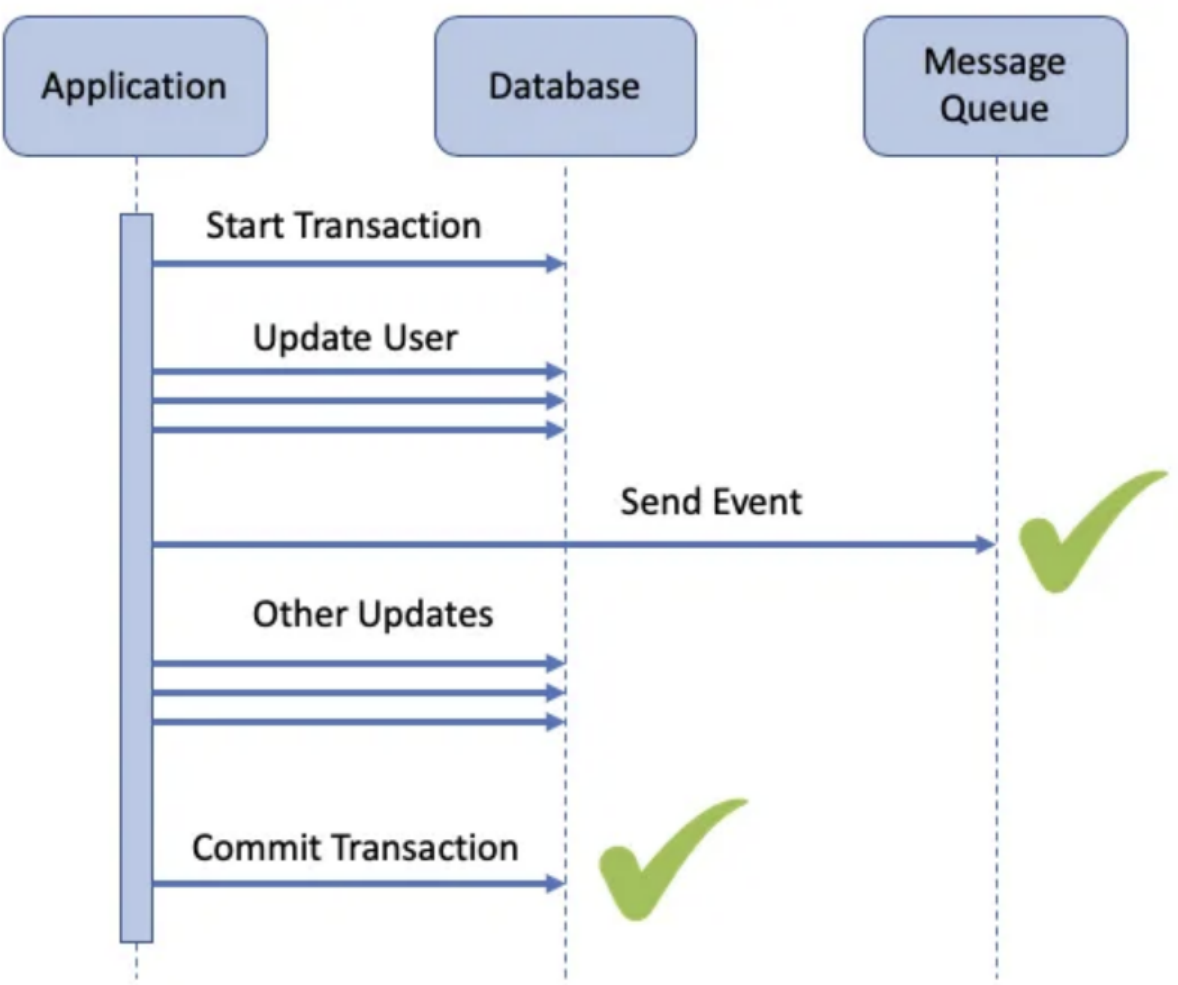
```
spring.cloud.stream.binders.rabbit.environment.spring.rabbitm
spring.cloud.stream.binders.rabbit.environment.spring.rabbitm
```

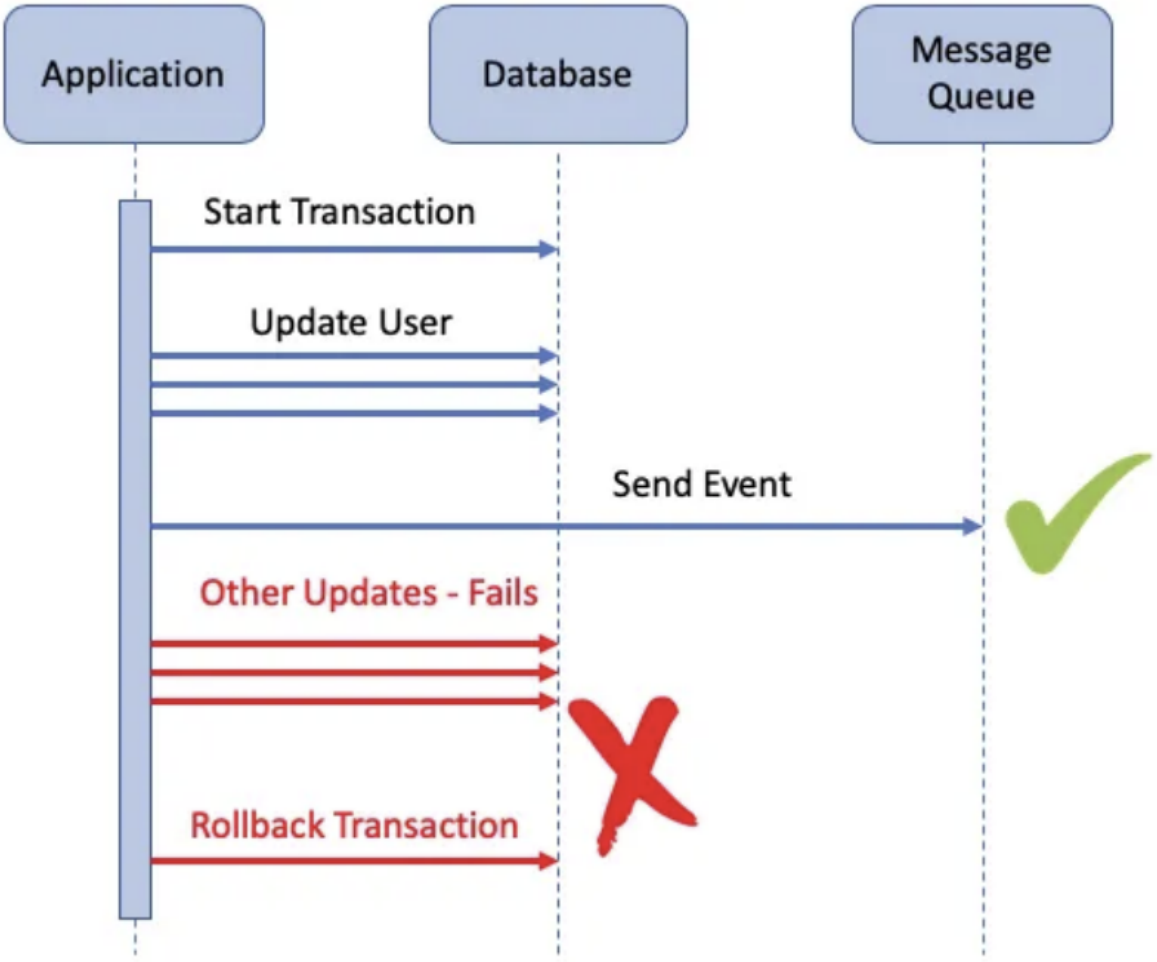
스프링 클라우드 스트림을 통한 메시지 생성 및 소비

> 참고 아티클

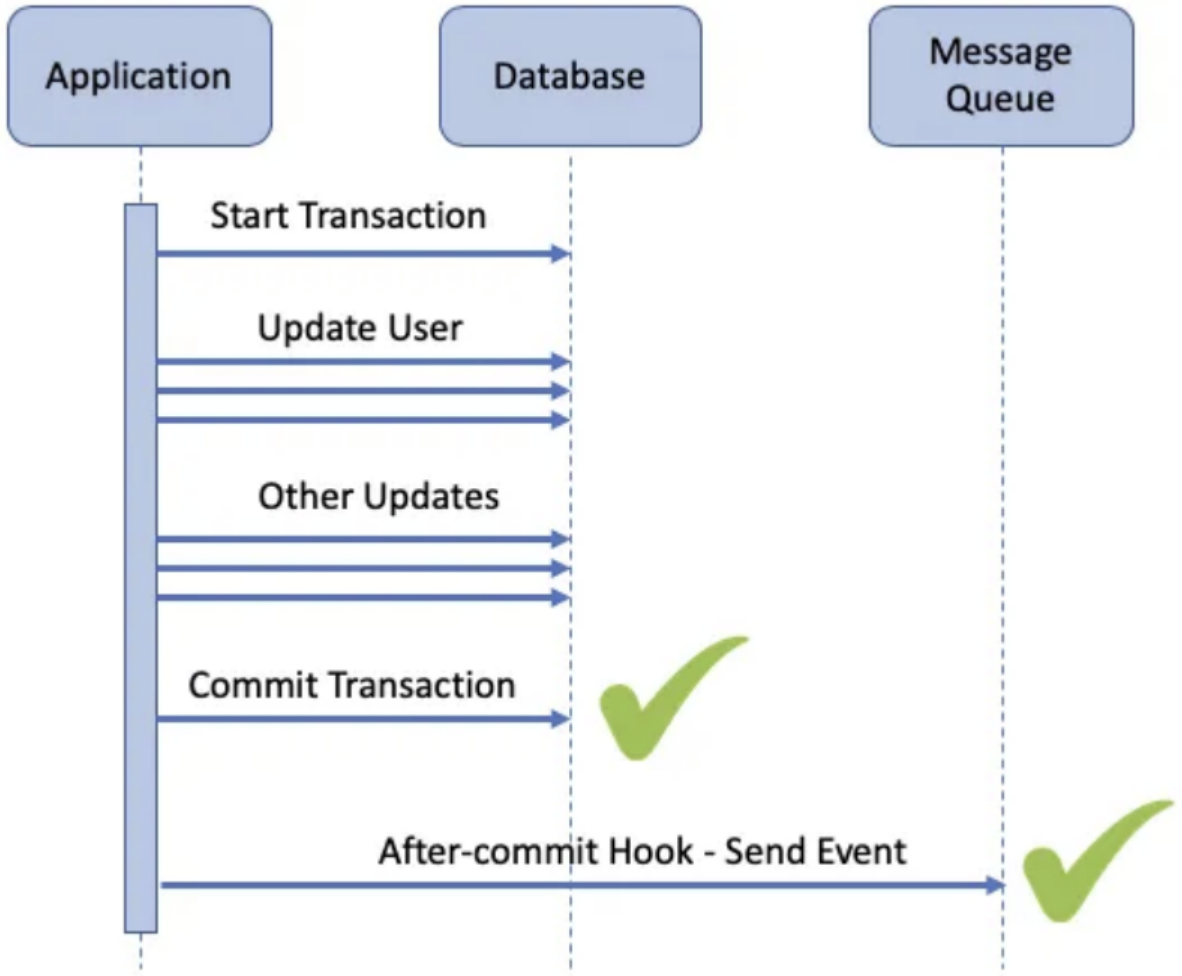
마이크로서비스 아키텍처에서 데이터베이스 저장과 메시지 전송이 원자적으로 수행되어야 한다

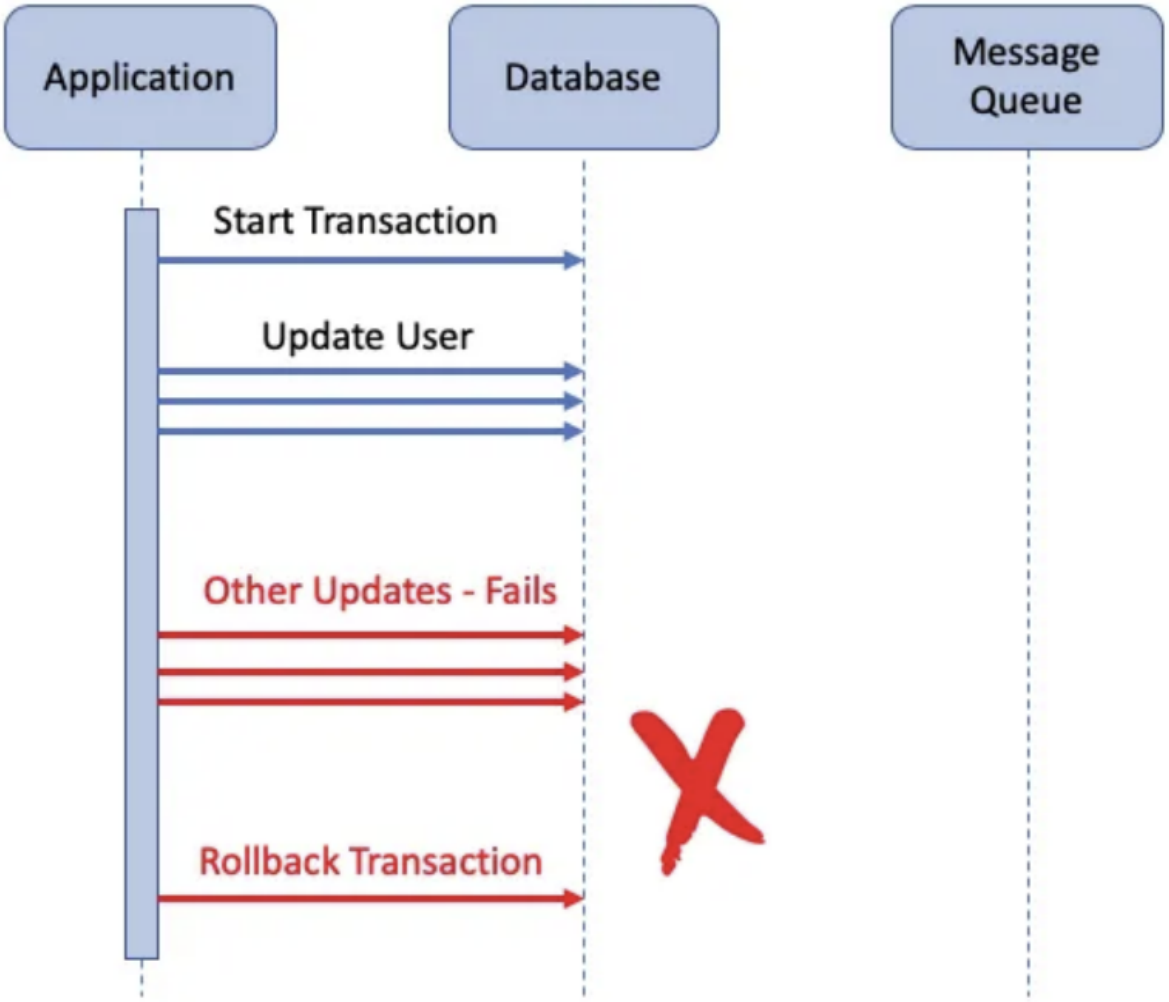
시도 1. 데이터베이스 트랜잭션 안에서 이벤트 발생

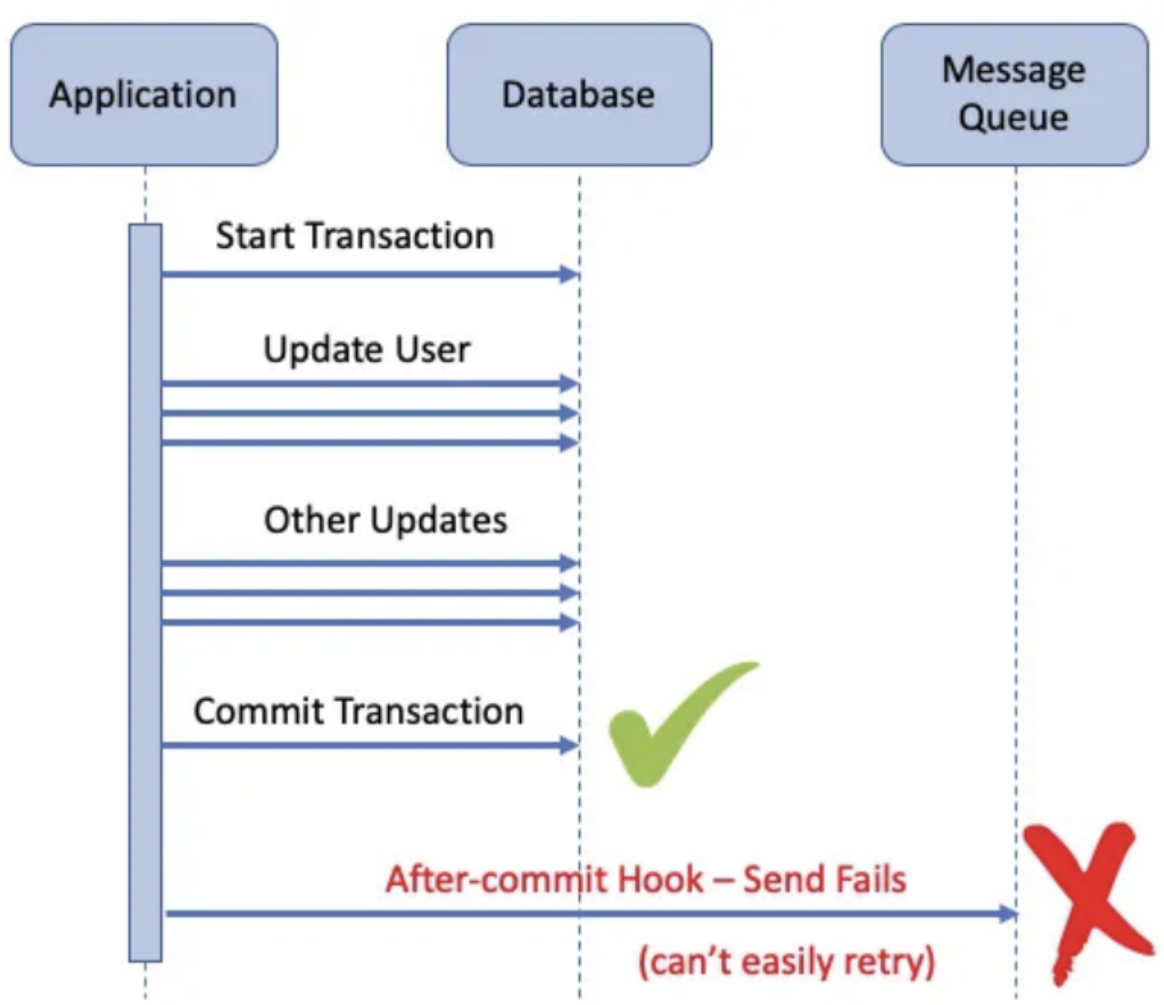

Application
Database
Message Queue
Start Transaction
Update User
Send Event
Other Updates
Commit Transaction



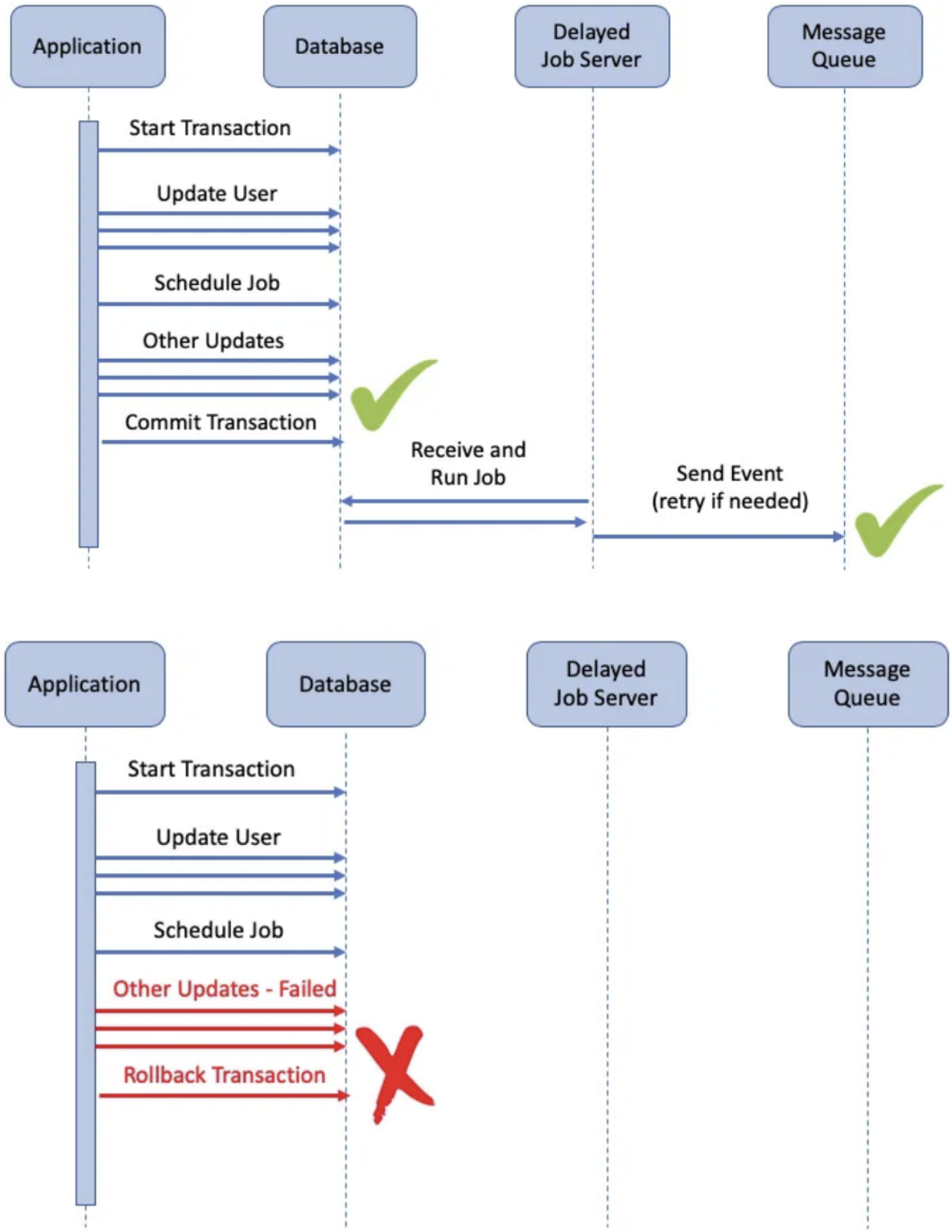
시도 2. 트랜잭션 커밋 이후 이벤트 발생


Application
Database
Message Queue
Start Transaction
Update User
Other Updates
Commit Transaction
After-commit Hook - Send Event

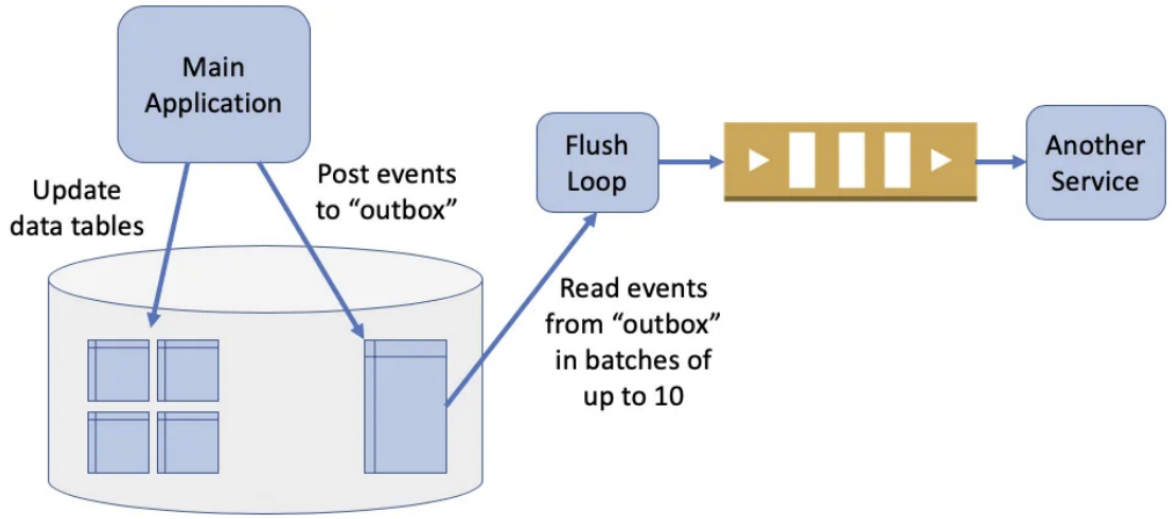

Application
Database
Message Queue
Start Transaction
Update User
Other Updates - Fails
Rollback Transaction



시도 3. 딜레이 이후 이벤트 발생



시도 4. 이벤트 정보를 데이터베이스에 삽입하고, 추후 전송



Applications persists events to "outbox" table, while flush loop batches and sends to queue.