Kafka 실시간 이벤트 데이터 동기화 전략

Skill Stack

분야	기술 스택
MSA & Gateway	🟦 Eureka, Spring Cloud Gateway, Spring Cloud Netflix
Backend	🟩 Spring Boot 3.x, Spring Web, Spring Security, Spring Data JPA, Spring Retry, Spring Data Redis, Spring Data Elasticsearch, Spring Kafka, Spring Cloud OpenFeign
Database	🟥 MySQL 8.0 (도메인별 DB 분리), PostgreSQL 15
Messaging	🟧 Apache Kafka 2.8.1, Zookeeper 3.8
Cache & Token	🟨 Redis 7, JWT (jjwt 0.12.5)
Logging & Monitoring	🟪 ELK Stack (Elasticsearch 7.17 운영 로그, Elasticsearch 8.13 검색 전용, Logstash, Kibana, Filebeat), Logstash Logback Encoder
Infra & CI/CD	🐋 Docker (docker-compose 기반 MSA 로컬 인프라 구성), GitHub Actions
Libraries & Tools	🛠️ Lombok, Jakarta Validation API, MySQL JDBC Driver

💥왜 Event-Driven Architecture(이벤트 중심 아키텍처)인가?

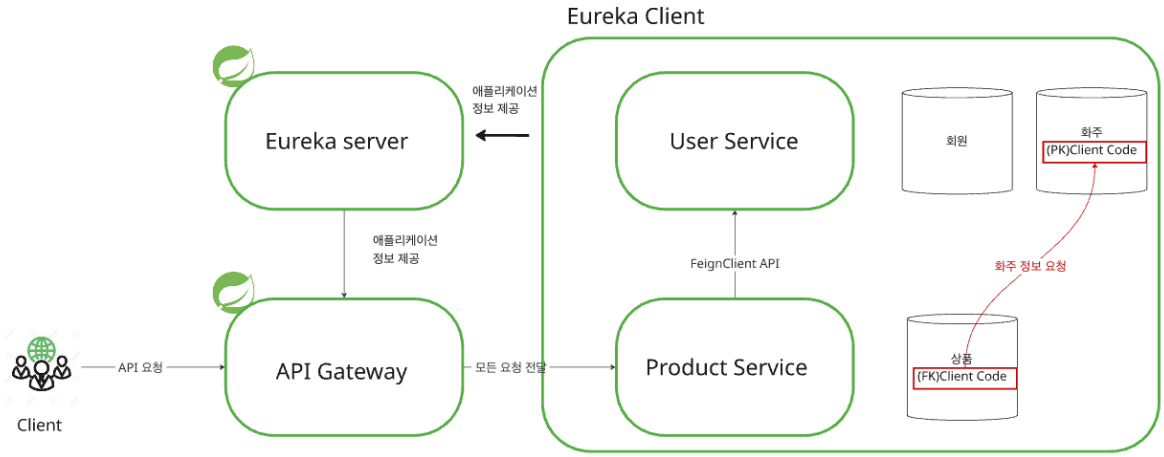
현대 대형 서비스 환경에서 MSA(Microservices Architecture)는 모든 문제를 단번에 해결하는 만능 해법이 될 수 없다고 판단합니다.

특히 서비스 별로 분리된 데이터베이스와 트랜잭션의 경계가 생기는 순간,

서비스 데이터의 타 서비스 실시간·정확 반영 가능성에 대한 고민이 시작됩니다.

이것은 실제로 이런 문제가 수시로 발생하는 것이 문제점이기도 합니다.

- 상품 시스템의 처리 상태는 ‘완료’인데, 회원 시스템의 처리 상태는 ‘대기’로 남아있는 경우
- 주기적 배치로 시스템 연동 중, 몇 분씩 데이터가 뒤늦게 동기화 되는 장애 발생(데이터 동기화 지연)



많은 기업들이 Feign, REST API, DB Link, 주기적 배치 등 다양한 방법을 시도합니다.

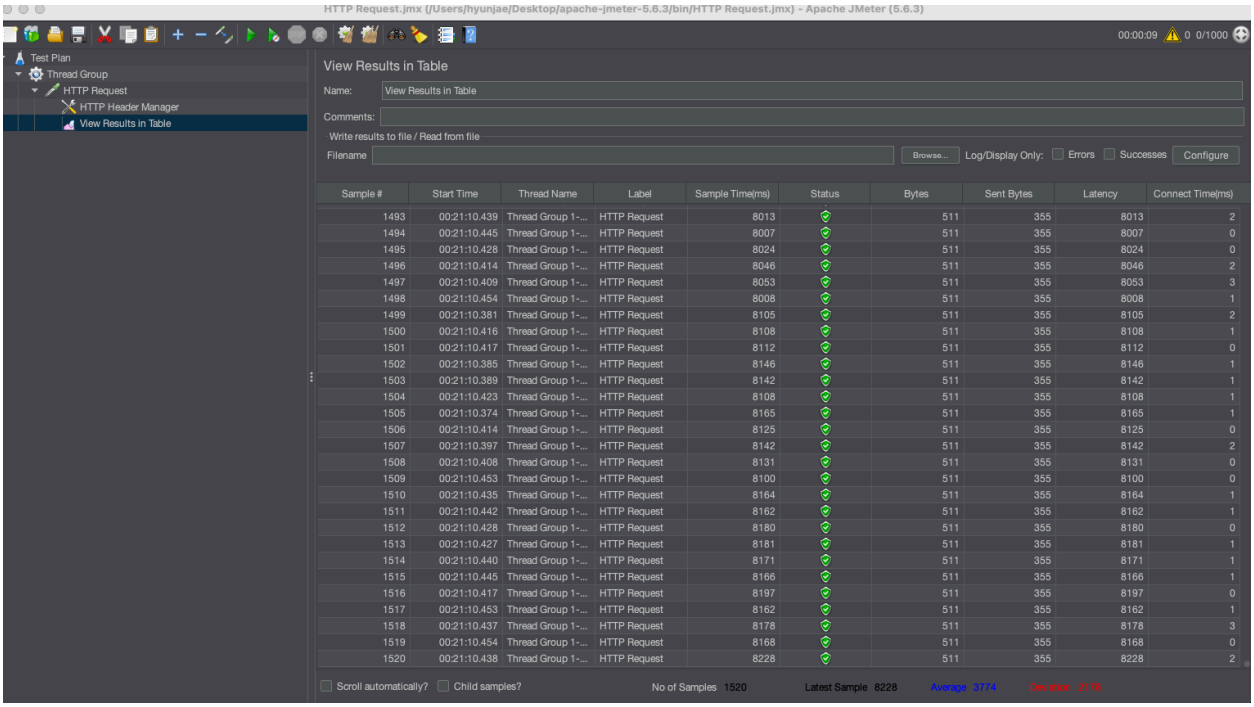
하지만 실시간성·확장성·운영성·장애 복구력 등의 문제를 해결하는 방법은 결국 "이벤트 동기화(Eventual Consistency via Events)"에 수렴하게 됩니다.

N+1 트래픽 병목과 데이터 정합성 이슈

대표적인 문제는 다음과 같습니다.

> "/products/with-client" API는 상품+화주명을 한 번에 보여주기 위해
>
> 상품마다 Feign → Gateway → UserService로 반복 요청합니다.
>
> 동시 요청자가 1000명만 넘어가도,
>
> 응답 지연이 발생하고, 이어서 DB 과부하와 네트워크 병목이 누적되며, 결국 서비스 장애로 확산됩니다.

JMETER 부하테스트 측정(Threads 1000 User)



- JMeter로 1000명/1500건 동시 요청:

  → 평균 응답 3.7초, 최대 8초까지 증가
- 초기 20명/30건에서는 120ms로 충분히 빠른 편
- 동시성이 높아질수록 서비스 자체의 확장성이 한계에 다다름

> 즉, 실시간 대량 트래픽에서 N+1 패턴과 동기 API 호출은 더 이상 통하지 않는다는 현실에 부딪힙니다.

구조적 한계와 데이터 일관성의 두 양면성

- MSA의 서비스/DB 분리로 인해 *조인 쿼리* 불가
- Feign, RestTemplate 등으로 타 서비스에 반복 요청 → N+1 패턴
- 데이터 일관성(Consistency) 보장 위해 매번 "즉시 조회" 필요 → 네트워크 및 DB 부하
- 장애, 네트워크 단절, 서비스 불가 시 전체 시스템 정합성 깨짐

구조 개선 – Kafka 기반 테이블 이벤트 동기화 아키텍처

(1) 구조 설계 변경의 필요성과 그 근거는 무엇인가?

사실 MSA 도입만으로 모든 게 해결되지는 않습니다.
상품 + 화주 통합 조회 API는 각 서비스의 DB가 분리되어 있어, 각 서비스로부터 데이터를 가져와 **실시간 조인**을 수행해야 했습니다.
초기에는 Feign 또는 Gateway 호출로 간단히 구현했지만, JMETER 부하 테스트 결과 **N+1 문제**와 **네트워크 병목 현상**이 발생했습니다.

이러한 문제를 해결하기 위해 구조를 다음과 같이 개선했습니다.

- **UserService(회원·화주)**
  - 화주 생성·수정 시 주요 변경 이벤트를 **Kafka**로 발행
  - 이벤트는 브로드캐스트 방식으로 전송
- **ProductService(상품)**
  - Kafka 이벤트를 실시간 수신하여 **자체 DB에 데이터 복제**
  - 상품+화주 통합 조회 시, 외부 호출 없이 자체 DB에서 바로 처리
- **효과**
  - Feign 호출 제거 → **N+1 문제 해소**
  - 서비스 간 강한 결합 제거 → **결합도↓, 확장성↑**
  - **단일 트랜잭션으로** 전체 서비스에 데이터 실시간 동기화

(2) 이벤트 동기화 설계 기준

- **실시간성** : 화주 정보 변경 후 **1~2초 이내** 모든 서비스에 반영
- **확장성**: 신규 서비스 추가, 트래픽 급증 상황에서도 **구조적으로 병목 없는 처리** 보장
- **신뢰성**: 장애·네트워크 문제·재처리(Replay) 상황에서도 **데이터 손실·순서 오류 없이** 복구 가능
-

데이터 일관성과 장애 복구 보장 전략

Kafka를 떠올리면 흔히 **'대용량 메시지 브로커'**, 비동기 이벤트 처리'를 먼저 생각하지만,
저는 **장애 복구와 데이터 정합성**을 가장 먼저 고려했습니다.

예를 들어, 다음과 같은 상황이 발생할 수 있습니다.

- **컨슈머 장애** : 프로세스가 죽었다가 재기동되는 경우
- **네트워크 단절** : 연결이 끊겼다가 복구되는 경우
- **중복 이벤트** : 동일 이벤트가 여러 번 전달되는 경우

이런 상황에서 데이터가 꼬이면, 운영자가 DB를 직접 수정해야 하고, 결국 MSA의 장점은 사라집니다.

이를 방지하기 위해 다음 설계를 적용했습니다.

- **멱등성(Idempotency)** : 동일 이벤트 반복 처리 시 결과가 변하지 않도록 구현
- **Key 기반 파티셔닝** : 동일 키의 이벤트를 동일 파티션에서 순서 보장
- **병렬 소비 설계** : 처리량 확장을 위한 안전한 병렬 컨슈머 구조

멱등성(Idempotency) — 장애·재처리 상황에서도 “한 번만” 반영

Kafka 환경에서는 **동일 이벤트가 여러 번 소비되는 경우**가 흔합니다.
네트워크 장애, 컨슈머 재기동, 재처리(Replay) 등의 이유로
같은 메시지가 반복 전달되더라도 **데이터는 단 한 번만 반영**되도록 보장해야 합니다.

그렇기때문에 Producer에서는

- `enable-idempotence: true`
- `acks: all`
- `retries: 5`

이러한 옵션을 통해 **가능한 한 중복 발행 자체를 차단**합니다.
또한 **Consumer** 단계에서는 **이미 처리한 이벤트**를 DB에서 직접 확인하여,
중복 데이터가 저장되지 않도록 **마지막 방어선**을 구축합니다.

```
if (clientMasterRepository.existsById(clientCode)) {
    ack.acknowledge();
    return;
}
clientMasterRepository.save(...);
ack.acknowledge();
```

이렇게 하면 컨슈머가 장애로 중단됐다가 재기동되거나,**Replay** 상황이 발생하더라도 **데이터는 단 한 번만 반영**됩니다.

Key 기반 파티셔닝

목표: “순서 보장”과 “병렬 처리”를 동시에 달성

- **순서 보장**

  동일 화주 데이터 이벤트가 섞여 **순서가 바뀌면** 안 됩니다.

- **병렬 처리**

  전체 트래픽을 단일 컨슈머가 처리하면 **대량 데이터에서는 감당 불가**합니다.

Kafka의 **Key 기반 파티셔닝**을 사용하면:

- 메시지 전송 시 Key(예: `clientCode`)를 지정
- 해당 Key의 모든 이벤트는 **항상 동일 파티션**에서, **순서대로** 소비
- 파티션 개수만큼 컨슈머 스레드를 확장하여 **자동 수평 확장** 가능

```
kafkaTemplate.send(TOPIC, String.valueOf(event.getClientCode()), message);
```

컨슈머 병렬 처리와 장애 복구를 위한 세부 설정

실제 환경에서는 컨슈머 속도가 느려지거나, 장애로 일부 인스턴스만 살아 있을 때 자동으로 **파티션 리밸런싱**이 발생합니다.
이때 **컨슈머 인스턴스/스레드 수를 파티션 개수와 맞추는 것**이 가장 안정적입니다.

또한, **수동 Ack 모드**를 사용하면 **DB 트랜잭션이 성공한 경우에만 Offset 커밋**을 통하여

데이터 유실과 순서 꼬임 위험을 크게 줄일 수 있습니다.

```
ConcurrentKafkaListenerContainerFactory factory = new ConcurrentKafkaListenerContainerFactory();
factory.setConcurrency(3);
factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
```

이렇게 세팅하면,

- **컨슈머가 장애로 중단**되더라도, 나머지 컨슈머가 **해당 파티션을 자동 인계**받아 처리 지속
- 장애 발생시, Kafka에서 **특정 offset부터 Replay**를 수행하면 **전체 데이터 복구 가능**

코드 & 설정 – 적용 사례와 세세한 배경

Producer (User Service)

- 화주 등록·수정 등 **핵심 이벤트**는 **반드시 Key 값을 지정**하여 전송하도록 명확히 규정했습니다.

```
@Component
@RequiredArgsConstructor
public class ClientEventProducer implements ClientEventPublisherPort {
    private static final String TOPIC = "client-created-topic";
    private final KafkaTemplate<String, Object> kafkaTemplate;
    private final ObjectMapper objectMapper;

    @Override
    public void publishClientCreatedEvent(ClientMasterEvent event) {
        ObjectNode message = objectMapper.createObjectNode();
        message.put("clientCode", event.getClientCode());
        message.put("clientName", event.getClientName());
        // ... 기타 필드 세팅
        kafkaTemplate.send(TOPIC, String.valueOf(event.getClientCode()), message);
```

```
    }
}
```

- ObjectMapper를 이용해서 DTO 객체 대신 JSON 형태로 변환

Producer 설정 (application.yml)

```
spring:
  kafka:
    producer:
      acks: all
      enable-idempotence: true
      retries: 5
```

- Key 없이 메시지를 전송하는 코드는 금지합니다.

Consumer (Product Service)

컨슈머는 다음 **세 가지 원칙**을 항상 준수하도록 구현했습니다.

1. **멱등성 체크** : 동일 이벤트 반복 처리 시 결과 불변 보장
2. **수동 Ack** : DB 트랜잭션 성공 시에만 Offset 커밋
3. **트랜잭션 일치** : 메시지 처리와 DB 작업의 원자성 유지

```
@KafkaListener(
    topics = "client-created-topic",
    groupId = "product-service-group",
    containerFactory = "kafkaListenerContainerFactory"
)
@Transactional
public void consume(JsonNode message, Acknowledgment ack) {
    int clientCode = message.get("clientCode").asInt();
    if (clientMasterRepository.existsById(clientCode)) {
        ack.acknowledge();
        return;
    }
    clientMasterRepository.save(...);
    ack.acknowledge();
}
```

- `Acknowledgment ack` :수동으로 오프셋(읽은 위치)을 커밋하는 객체 → 직접 처리 성공 시점에만 커밋이 가능합니다.

```
if (clientMasterRepository.existsById(clientCode)) {
    ack.acknowledge();
```

```
    return;
}
```

멱등성 체크

이미 DB에 동일한 `clientCode` (화주)가 존재한다면,
즉 **중복된 이벤트/메시지**라면 아무 작업도 수행하지 않습니다.
대신 `ack.acknowledge();` 를 호출하여 **"이 메시지까지 처리 완료"** 가 되었다는 신호만 Kafka에 전송합니다.

```
clientMasterRepository.save(...);
ack.acknowledge();
```

- DB에 없는 화주라면

  → 이벤트 정보를 바탕으로 신규 저장(복제)

- 모든 처리가 정상적으로 완료된 후

  → `ack.acknowledge();` 를 호출하여 **Offset 커밋**

  (Kafka에 "이 메시지가 문제없이 처리되었다"는 신호 전송)

KafkaListenerContainerFactory 세부 설정

- **concurrency**:
  - 파티션 수와 동일하게 설정하여 **자동 병렬 소비** 지원
- **AckMode**
  - `MANUAL_IMMEDIATE` 로 설정
  - 트랜잭션 완료 후에만 Offset 커밋

```
ConcurrentKafkaListenerContainerFactory factory = new ConcurrentKafkaListenerContainerFactory();
factory.setConcurrency(3);
factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
```

마치며...

이번 프로젝트/스터디를 통해
"MSA 환경에서 데이터 정합성과 확장성을 실시간으로 어떻게 지킬 것인가?"라는
오래된 숙제에 한 걸음 더 다가설 수 있었습니다.

Kafka 기반 이벤트 동기화 구조를 직접 설계·구현하며,
그동안 이론으로만 접하던 개념들이 실제 환경에서 어떻게 적용되는지,
그리고 어떤 부분을 반드시 고려해야 하는지를 몸소 체감한 시간이었습니다.