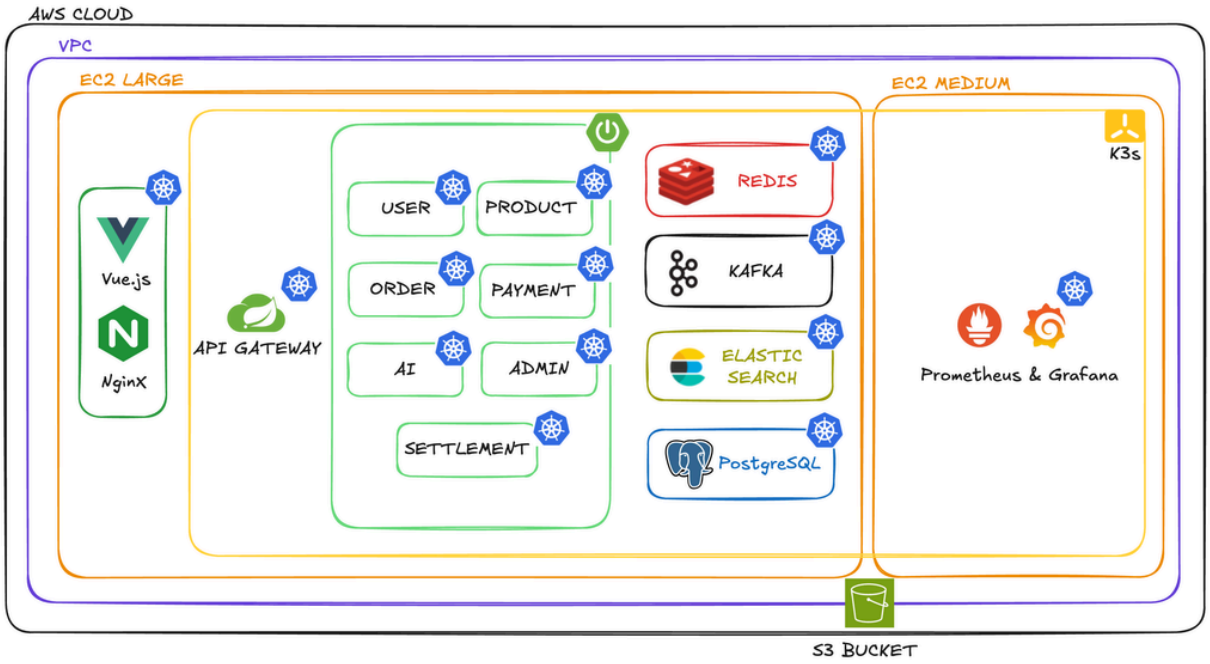
임요섭 포트폴리오 (신입)

Proficiency

- Spring Boot & Java 활용하여 애플리케이션 개발 및 구현이 가능합니다.
- 데이터베이스 레벨에서 병목지점을 분석하고 인덱스 및 쿼리 개선을 통해 성능 개선이 가능합니다.
- Kafka와 Outbox 패턴을 활용하여 MSA 환경에서 이벤트 유실 없이 서비스 간 데이터 정합성을 보장할 수 있습니다.
- Elasticsearch 기반 전문 검색 시스템 설계 및 nori 형태소 분석기, fuzziness 설정을 통한 한국어 검색 품질 최적화가 가능합니다.
- Resilience4j 서킷 브레이커와 Fallback 전략을 적용하여 외부 시스템 장애에도 서비스 가용성을 유지할 수 있습니다.

Project Portfolio #1 : 원데이 클래스 예약 플랫폼 : 잡아 클래스 (Jaba Class)

Project Architecture

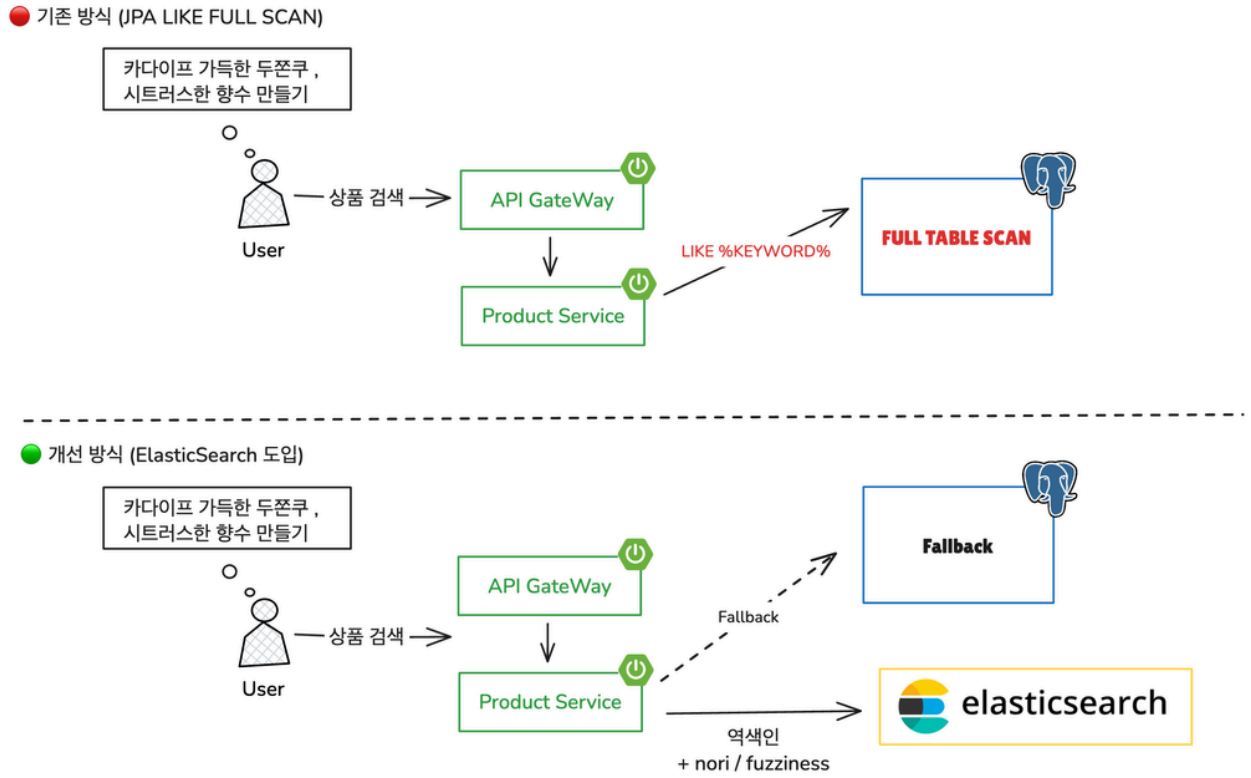


Technical Challenges

1. **유명 셀러 클래스 오픈 시 검색 트래픽 집중으로 인한 성능 저하를 해결하기 위해 Elasticsearch를 도입, p95 응답시간 31% 개선 및 실패율 0% 달성**

- 문제 원인
  - 유명 셀러의 클래스 오픈 시 단시간에 검색 트래픽이 집중되는 스파이크 패턴 발생
  - 기존 JPA LIKE %keyword% 방식은 앞 와일드카드로 인해 B-Tree 인덱스 미사용, Full Table Scan으로 동작
  - 데이터 규모가 커질수록 선형(O(N)) 비용 증가 — 1,000건 → 100,000건(100배) 시 DB p95 20ms → 50.6ms (+153%)
- 해결 과정
  - ES는 선택적 미들웨어이므로 DB 자체 최적화를 먼저 시도: PostgreSQL pg_trgm GIN 트라이그램 인덱스 직접 적용
  - GIN 적용 결과 100만 건 / 100 VU 기준 p95 1,879ms → 1,654ms (12% 개선)에 그쳤고, 극한 부하 시 실패 14건 발생
  - GIN의 한계를 수치로 검증한 뒤 ES 도입 결정
  - nori 형태소 분석기 적용으로 한국어 전문 검색 지원, title에 fuzziness: AUTO + prefixLength: 1로 오타 허용

- 아키텍처



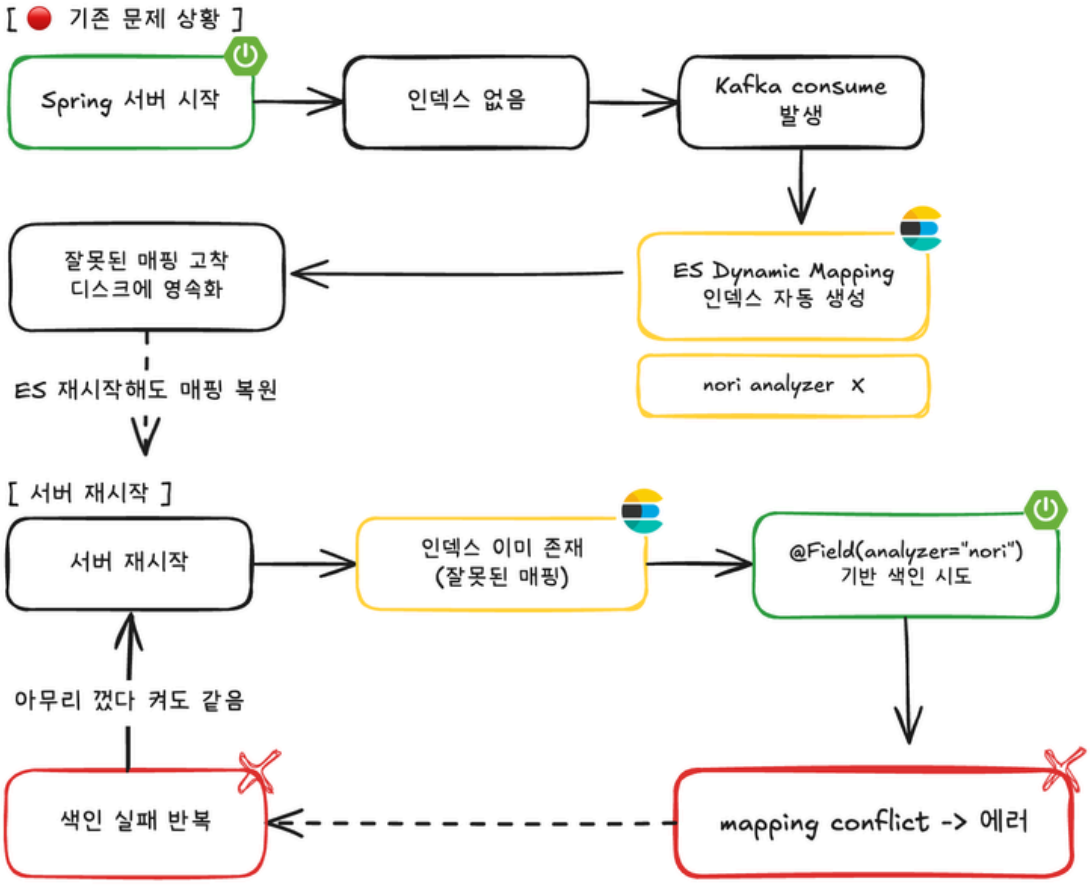
- 테스트
  - k6로 1,000건 / 10만 건 / 100만 건 3단계 부하 테스트 수행
  - ES vs DB 풀스캔 vs DB+GIN 3가지 옵션에 대한 테스트 진행
- 결과
  - DB 풀스캔 p95 1,879ms → DB+GIN p95 1,654ms → ES p95 1,294ms (DB 풀스캔 대비 31% 개선)
  - DB+GIN 대비 ES 22% 빠름, 실패율 0% 유지
  - 데이터 1,000건 → 100만 건 증가 시 DB p95 +153%, ES p95 +85% 증가로 ES의 확장 안정성 확보

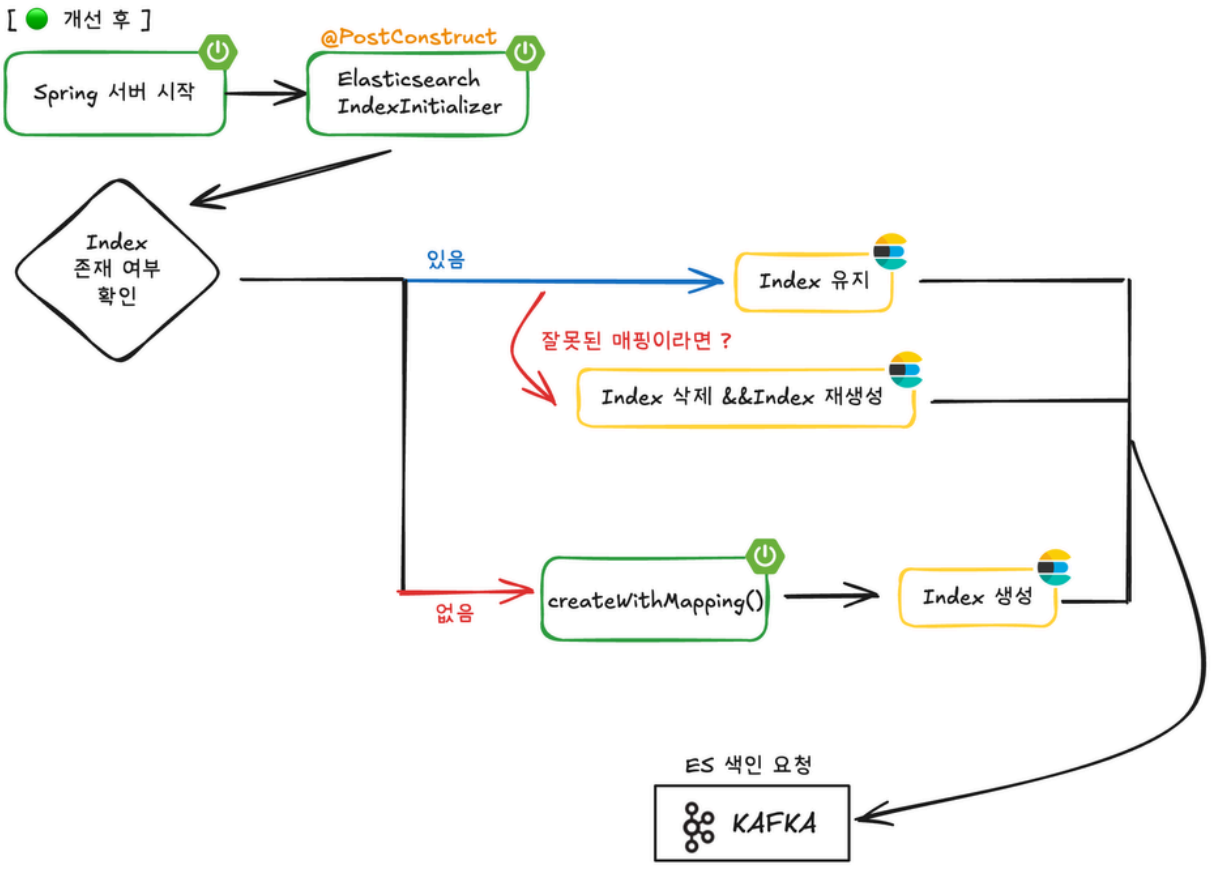
---

2. ES Dynamic Mapping으로 인한 nori 분석기 누락 문제 해결 및 재발 방지 설계

- 문제 원인
  - ES는 첫 문서 색인 시 필드 타입을 자동 추론(Dynamic Mapping)하여 매핑을 생성함
  - Kafka consumer가 인덱스 초기화보다 먼저 실행되어 nori 분석기가 빠진 상태로 매핑이 고착됨
  - 잘못된 매핑은 디스크에 영속화되어 파드 재시작 후에도 살아남음 → 한국어 형태소 분석 불가 상태 지속
- 해결 과정
  - 최초에 인덱스 삭제 후 재생성으로 서비스 복구 시도
    - ddl-auto: create / create-drop 감지 시 기존 인덱스를 삭제하고 재생성하는 로직 추가
  - 재발 방지를 위해 명시적 매핑 선생성으로 Dynamic Mapping 차단
    - @Document, @Setting, @Field로 nori 분석기 포함 매핑을 코드로 명시
    - createWithMapping()으로 인덱스 생성 시 매핑을 함께 적용, ES의 자동 추론을 차단
  - 초기화 순서 보장으로 근본 원인 개선
    - ddl-auto 감지 대응 이후에도 ApplicationRunner 방식은 Kafka consumer 시작 순서를 보장하지 못해 race condition 가능성 존재
    - @PostConstruct로 전환하여 Kafka consumer 등록 전 인덱스 초기화가 반드시 먼저 실행되도록 보장

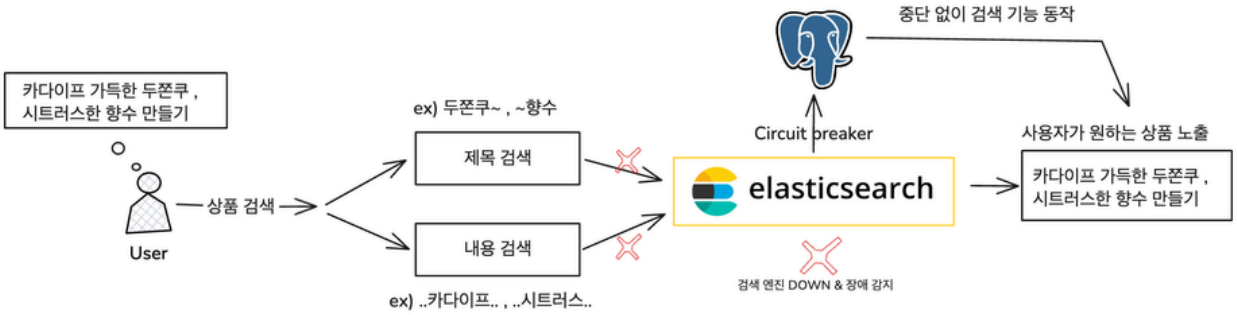
- 아키텍처


[기존 문제 상황]
Spring 서버 시작
인덱스 없음
Kafka consume 발생
ES Dynamic Mapping 인덱스 자동 생성
nori analyzer X
잘못된 매핑 고착 디스크에 영속화
ES 재시작해도 매핑 복원
[서버 재시작]
서버 재시작
인덱스 이미 존재 (잘못된 매핑)
@Field(analyzer="nori") 기반 색인 시도
아무리 껐다 켜도 같음
색인 실패 반복
mapping conflict -> 에러


[개선 후]
Spring 서버 시작
@PostConstruct
Elasticsearch IndexInitializer
Index 존재 여부 확인
있음
Index 유지
잘못된 매핑이라면 ?
Index 삭제 &&Index 재생성
없음
createWithMapping()
Index 생성
ES 색인 요청
KAFKA

3.ES 장애 시 검색 API 전체 오류를 방지하기 위해 Resilience4j 서킷 브레이커를 도입, 카오스 테스트(ES 컨테이너 강제 종료)로 장애 중 실패율 0% 유지 검증

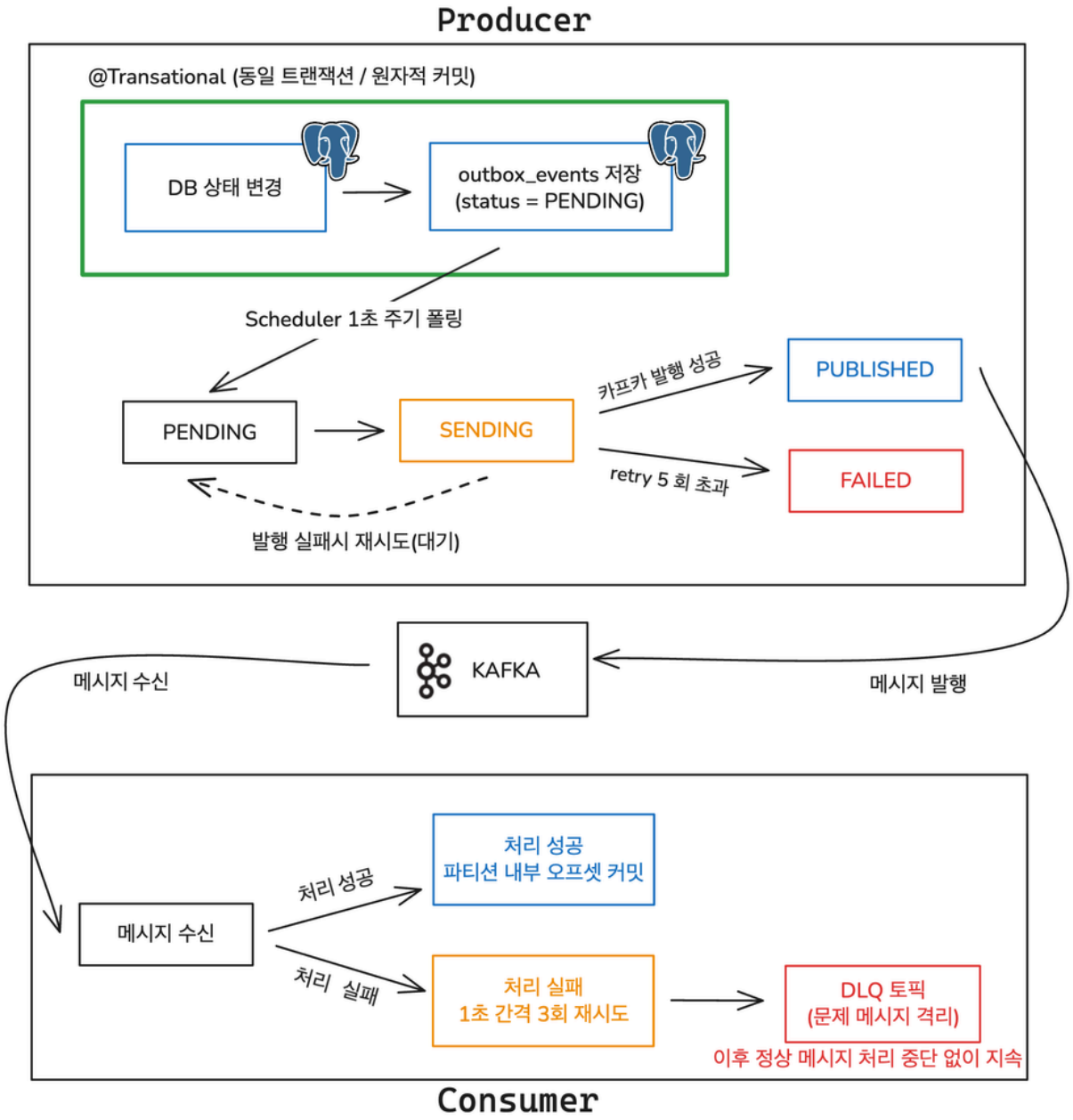
- 아키텍처



- 문제 원인
  - ES를 메인 검색 엔진으로 사용하는 구조에서 ES 단일 장애 발생 시 검색 API 전체 오류로 이어지는 단일 장애점(SPOF) 문제 존재
  - ES 응답 무한 대기로 인해 Circuit Breaker 감지 전까지 최대 3,259ms 지연 발생
  - socket-timeout 미설정 상태에서 ES 장애 시 스레드가 응답을 무한 대기하며 서비스 전체 성능 저하 유발
- 해결 과정
  - searchAll() / searchMy() 메서드에 @CircuitBreaker(name = "elasticsearchCB") 적용, ES 장애 시 즉시 PostgreSQL 직접 조회로 fallback 전환
  - slidingWindowSize: 10 / failureRateThreshold: 50% / waitDurationInOpenState: 30s 설정으로 CB 동작 기준 수립
  - ES socket-timeout 2s 설정으로 CB 감지 전 무한 대기 문제 선제 차단 → 최대 지연 3,259ms → 2,000ms 단축
  - CB CLOSED 전환(ES 복구) 감지 시 장애 중 누적된 FAILED 상태 outbox 이벤트를 자동으로 재처리하여 ES 색인 정합성 자동 복구
- 테스트
  - ES 컨테이너 강제 종료(카오스 테스트)로 실제 장애 상황 재현
  - CB 상태 전이(CLOSED → OPEN → HALF_OPEN → CLOSED) 흐름 및 fallback 동작 검증
- 결과
  - ES 장애 중 검색 API 실패율 0% 유지 (fallback DB 조회로 응답 지속)
  - ES 복구 후 FAILED outbox 이벤트 자동 재처리로 ES 색인 정합성 자동 복구
  - socket-timeout 2s 적용으로 장애 시 최대 응답 지연 3,259ms → 2,000ms 단축

4. **MSA 환경에서 글로벌 트랜잭션 불가로 인한 데이터 불일치 문제를 Outbox 패턴으로 해결, 이벤트 유실 0% 및 데이터 정합성 보장**

- 아키텍처



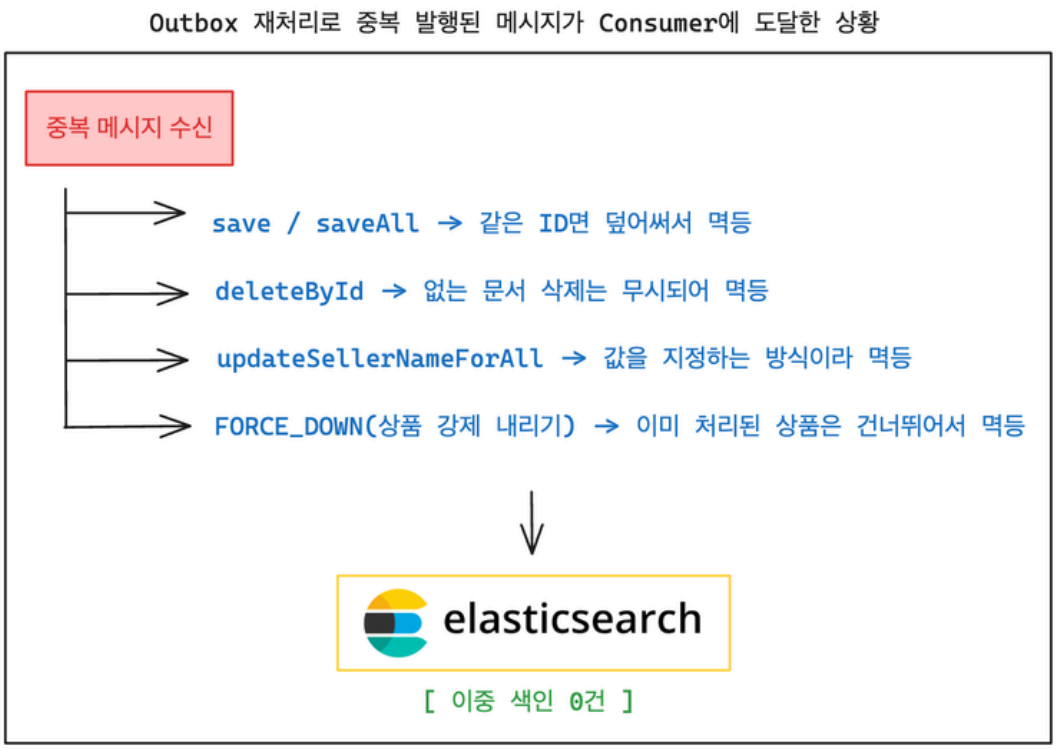
- 문제 원인
  - MSA 환경에서는 글로벌 트랜잭션 사용이 불가하여 DB 저장과 Kafka 발행을 하나의 트랜잭션으로 묶을 수 없음
  - 상태 변경 후 Kafka 발행 실패 시 DB는 변경됐으나 이벤트는 미발행되는 데이터 불일치 발생
  - Kafka 브로커 일시 장애 시 이벤트가 영구 유실되어 재고/예치금 복구가 되지 않는 상황 발생 가능

- **해결 과정**
  - Kafka에 직접 발행하지 않고 같은 트랜잭션 안에 outbox_events 테이블에 먼저 저장하여 DB 커밋과 이벤트 저장을 원자적으로 처리
  - 별도 스케줄러가 1초 주기로 미발행 이벤트를 폴링하여 Kafka 발행 후 상태 전이 (PENDING → PUBLISHED)
  - 다중 인스턴스 환경에서 중복 발행 방지를 위해 FOR UPDATE SKIP LOCKED 적용
  - 발행 실패 시 재시도 횟수 누적, 임계값 초과 시 FAILED 처리로 무한 재시도 방지
  - Consumer 처리 최종 실패 시 DLQ로 라우팅하여 문제 메시지를 격리, 이후 정상 메시지 처리는 중단 없이 지속
- **테스트**
  - Kafka 브로커 강제 종료 후 재기동 시 미발행 이벤트 자동 재발행 여부 확인
  - 다중 인스턴스 동시 실행 환경에서 중복 발행 방지 동작 검증
- **결과**
  - Kafka 장애 상황에서도 이벤트 유실 0%, 브로커 복구 후 자동 재발행
  - DB 상태와 이벤트 발행 간 데이터 정합성 보장

---

5. Kafka at-least-once 환경에서 ES 색인 컨슈머 멱등성을 검증·설계, 이중 색인 0건 및 별도 멱등성 인프라 없이 중복 수신 대응

- **아키텍처**

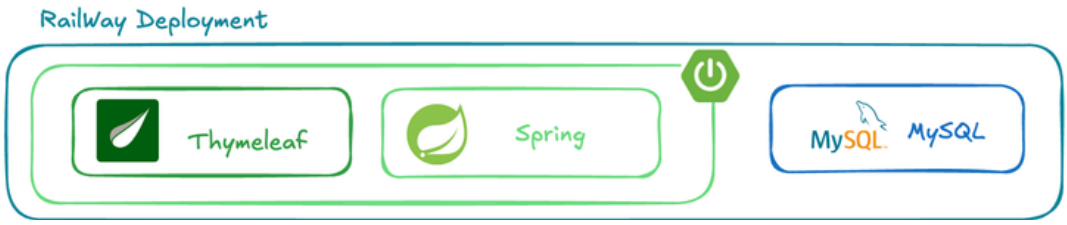


- **문제 원인**
  - Outbox 발행 성공 후 PUBLISHED 전환 전 인스턴스 장애 시, 타임아웃 재처리로 같은 메시지가 중복 발행될 수 있음
  - ES 색인 컨슈머가 같은 이벤트를 두 번 처리하면 이중 색인 또는 잘못된 상태가 남을 수 있음

- **해결 과정**
  - **save / saveAll** : ES는 동일 document ID 재색인 시 전체 덮어쓰기(upsert) → 같은 ID로 다시 색인하면 ES가 덮어써서 결과가 같음
  - **deleteById** : 존재하지 않는 문서 삭제 시 result=not_found 처리, 예외 없음 → 이미 삭제된 문서를 다시 삭제해도 에러 없이 무시됨
  - **updateSellerNameForAll** : 판매자 이름을 누적이 아닌 고정값으로 지정하는 방식으로 스크립트 작성 → 재실행해도 결과가 같음
  - **FORCE_DOWN** : status=DISABLE && deleteDt != null 조건으로 early return 가드 추가 → 이미 처리된 상품 이벤트 건너뜀
  - ES document ID가 멱등 키 역할을 하므로 processed_events 테이블 없이 동일 연산의 중복 수신에 대한 멱등성 확보
- **테스트**
  - Outbox SENDING 타임아웃 재처리로 Kafka 중복 발행 재현 → ES 최종 색인 상태 동일 여부 확인
  - FORCE_DOWN 이벤트 중복 수신 시 두 번째 이벤트가 건너뛰어지는지 확인
- **결과**
  - Outbox 재처리 시나리오에서 이중 색인 0건
  - processed_events 테이블 없이 자연 멱등성 활용, 인프라 추가 없이 동일 연산 중복 수신에 대한 안전성 확보

Project Portfolio #2 : 다니엘 청년부 행정 관리 웹 애플리케이션

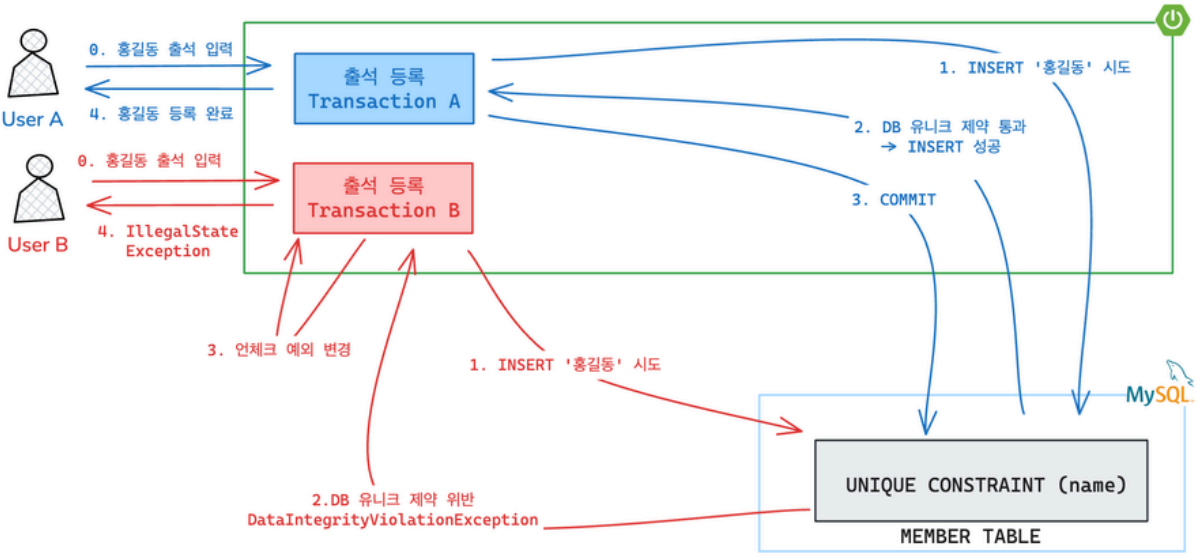
Project Architecture



Technical Challenges

1. **다수의 인원이 동일 인원 출석 등록 시 발생하는 동시성 문제(데이터 중복)를 해결하기 위해 DB Unique 제약을 적용하여, 데이터 중복률 0% 및 무결성 100% 달성**

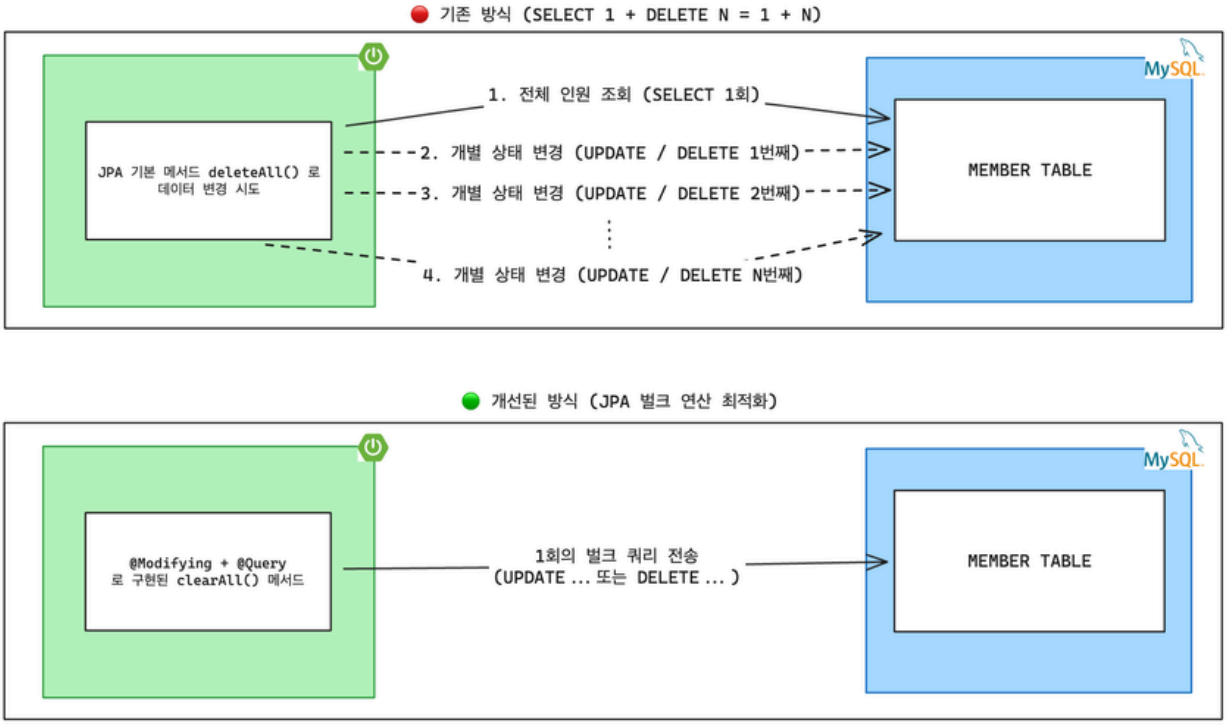
- 아키텍처



- 문제 원인
  - 동일 인원을 여러 관리자가 동시에 출석 등록 요청할 때 같은 인원이 중복 저장되는 동시성 이슈 발생.
  - 애플리케이션 단의 중복 검증 로직이 없는 상태에서 여러 트랜잭션이 동시에 DB에 INSERT를 시도하여 데이터가 중복 저장되는 현상 확인
  - 데이터 무결성이 깨져 관리자가 수동으로 중복 데이터를 식별하고 삭제해야 하는 불필요한 운영 리소스 낭비 발생
- 해결 과정
  - @Column(unique = true)로 name 컬럼에 유니크 제약을 설정하여, 동시 INSERT 시 InnoDB가 묵시적 락으로 트랜잭션을 직렬화 처리
  - 동시 요청으로 발생하는 DataIntegrityViolationException을 catch하여 IllegalStateException으로 변환, 사용자에게 명확한 에러 메시지 응답
- 테스트
  - Postman을 활용하여 동일한 사용자 이름으로 다중 POST 요청을 동시다발적으로 전송해 Race Condition 상황을 의도적으로 재현 → 서버 로그와 DB 조회로 데이터 정합성 교차 검증
- 결과
  - 동시 출석 등록 요청이 몰려도 DB 유니크 제약이 INSERT를 직렬화해 단 1건만 저장, 나머지는 IllegalStateException으로 차단 → 데이터 중복률 0%
  - 동명이인은 유니크 제약 위반을 감지해 "이름 뒤 A 붙여 입력(예: 홍길동A)" 안내로 구분 등록을 유도 → 차단이 아닌 분기로 처리해 정상 인원 누락 방지

2. 매주 전체 인원의 상태 일괄 갱신시 인원수(N)만큼 쿼리가 발생하는 성능 문제를 해결하기 위해 JPA 벌크 연산 최적화로, 쿼리 실행 횟수 N회 -> 1회 단축

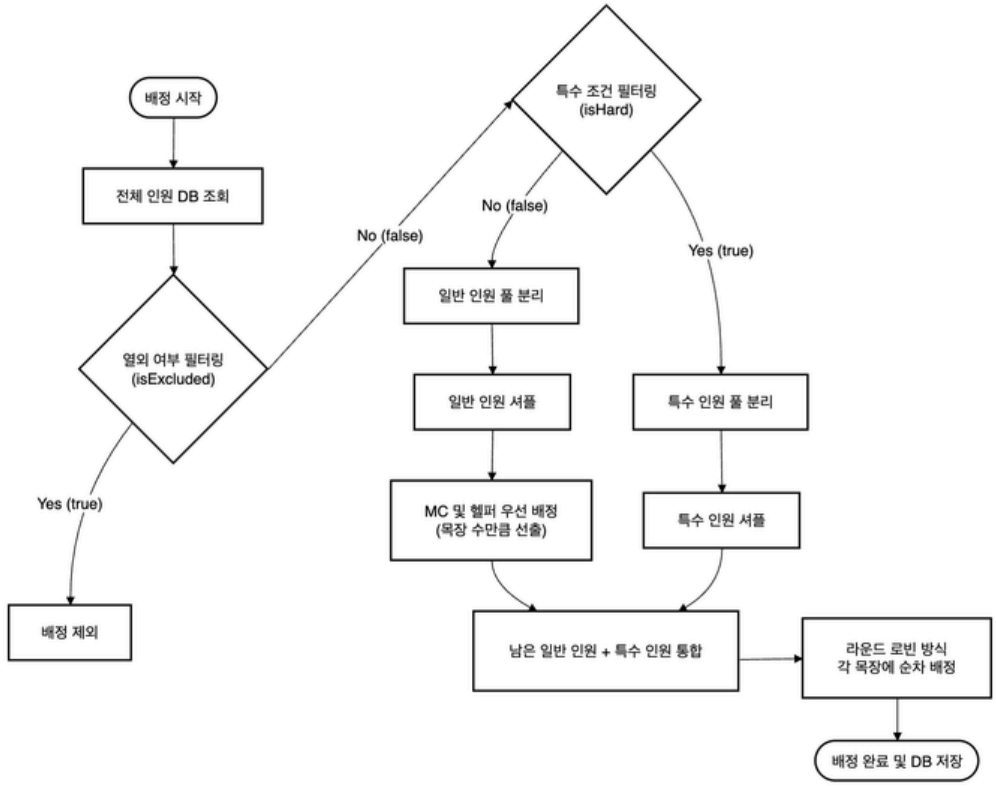
- 아키텍처



- 문제 원인
  - 매주 전체 인원의 상태(목장 배정 등)를 일괄적으로 초기화하는 기능에서 성능 저하 우려 발견
  - JPA가 기본 제공하는 엔티티 삭제/수정 메서드를 사용할 경우, 먼저 대상 엔티티를 모두 조회(SELECT 1회)한 후 각 엔티티마다 개별적으로 쿼리(UPDATE/DELETE N회)가 발생하는 1+N 문제 발생
  - 현재 규모(30 ~ 40명 내외)를 넘어 향후 대규모 트래픽 환경이 되었을 때, 1+N 쿼리로 인한 불필요한 네트워크 I/O 급증이 시스템 성능 저하로 이어질 가능성을 고려하여 사전 최적화 진행
- 해결 과정
  - 1+N 문제를 근본적으로 해결하기 위해, 영속성 컨텍스트(1차 캐시)를 거쳐 개별적으로 처리하는 방식 대신 데이터베이스에 직접 단일 쿼리를 전달하는 JPA 벌크 연산 도입
  - MemberRepository 인터페이스에 전체 데이터를 한 번에 수정/삭제하는 커스텀 JPQL 작성
  - 벌크 연산은 영속성 컨텍스트를 거치지 않고 DB에 직접 쿼리를 실행하므로, 컨텍스트에 남아있던 변경분이 벌크 연산에 누락되지 않도록 @Modifying(flushAutomatically = true)로 실행 전 flush를 보장
- 결과
  - 기존 1 + N회 발생하던 쿼리 실행 횟수를 단 1회로 단축
  - 대량의 데이터 갱신 시 발생할 수 있는 메모리 과부하 및 트랜잭션 타임아웃 위험을 사전에 차단하여 시스템 안정성 확보

3. 다중 필터링 로직 기반의 랜덤 알고리즘 : 구성원의 장애 여부, 당일 모임 참석 여부 등 세부 조건에 따른 유연한 그룹핑이 가능한 목장 생성 기능 구현

- **도입 배경 및 목적**
  - 기존 수동 편성 방식은 매주 관리자의 리소스가 크게 소모되며, 인적 편향이 개입될 여지가 존재
  - 단순한 랜덤 분배를 넘어, '목장 모임 열외 인원 처리' 및 '특수 조건(장애 여부 등) 인원의 균등 분배' 등 복잡한 비즈니스 요구사항을 유연하게 처리할 수 있는 자동화 시스템 필요
- **핵심구현 로직**
  - 엔티티의 isExcluded, isHard 속성을 활용하여 전체 인원 중 배정 제외자를 1차로 필터링하고, 균등 분배가 필요한 특수 인원(장애 여부 등) 풀과 일반 인원 풀을 2차로 분리 → 이후 Collections.shuffle() 등을 적용하여 매주 겹치지 않는 새로운 조합이 나오도록 난수 기반 셔플링 적용
  - 특수 인원을 먼저 각 목장에 1명씩 균등하게 배치한 후, 남은 일반 인원을 순차적으로 배분하여 특정 목장에 인원이나 조건이 편중되지 않도록 알고리즘 구현
- **알고리즘 플로우차트**



- **도입 효과**
  - 복잡한 조건이 반영된 목장 편성 작업을 클릭 한 번으로 자동화하여 관리자의 업무 시간을 획기적으로 단축
  - 알고리즘 기반의 다중 필터링 및 랜덤 배정을 통해 편성의 공평성과 투명성을 기술적으로 보장, 구성원들의 서비스 신뢰도 및 사용자 경험 증대
  - 추후 새로운 조건이 추가되더라도 필터링 파이프라인에 쉽게 추가할 수 있는 확장성 있는 코드 구조 구현