신지섭

2 **이벤트 기반 게임 스토어 백엔드**
- 3 저장소 장애에 따른 오환불 방지
- 5 환불 후 재구매 허용과 보유권 중복 방지
- 7 Outbox 릴레이의 키별 발행 순서 역전 방지

9 **수집형 RPG 게임 서버 백엔드**
- 10 단계·청크 체크포인트 커밋으로 정산 롤백 범위 축소
- 12 조건부 UPDATE로 정산 시작 경합 해결
- 14 플레이어 단위 액터 직렬화로 쓰기 경합 제어

16 **멘토링·채용 정보 서비스 백엔드**
- 17 영상 행 선생성으로 비동기 변환과 뉴스 작성 분리
- 18 STOMP 메시지 발신자를 연결 인증 사용자로 고정

프로젝트

이벤트 기반 게임 스토어 백엔드

2026-08 – 진행 중

창작자 등록부터 결제·다운로드·정산까지 Kafka 이벤트로 잇는 게임 스토어 백엔드

개요

게임 스토어 업무를 Gateway·Auth와 크리에이터, 커머스, 이용·정산 트랙의 앱 모듈 **11개**로 나눈 백엔드입니다.

서비스 간 이벤트는 Outbox로 Kafka에 발행하고, 수신 측은 Inbox로 중복 처리를 막습니다.

결제·환불은 PortOne TEST 채널로 검증했습니다.

범위

- 주문·결제·환불과 보관함·보유권 흐름(Order·Payment·License)
- Outbox 릴레이의 키 순서 보존, DLT 보류와 재투입
- 판매자 배분·월 마감 정산과 ArchUnit 모듈 경계 검사

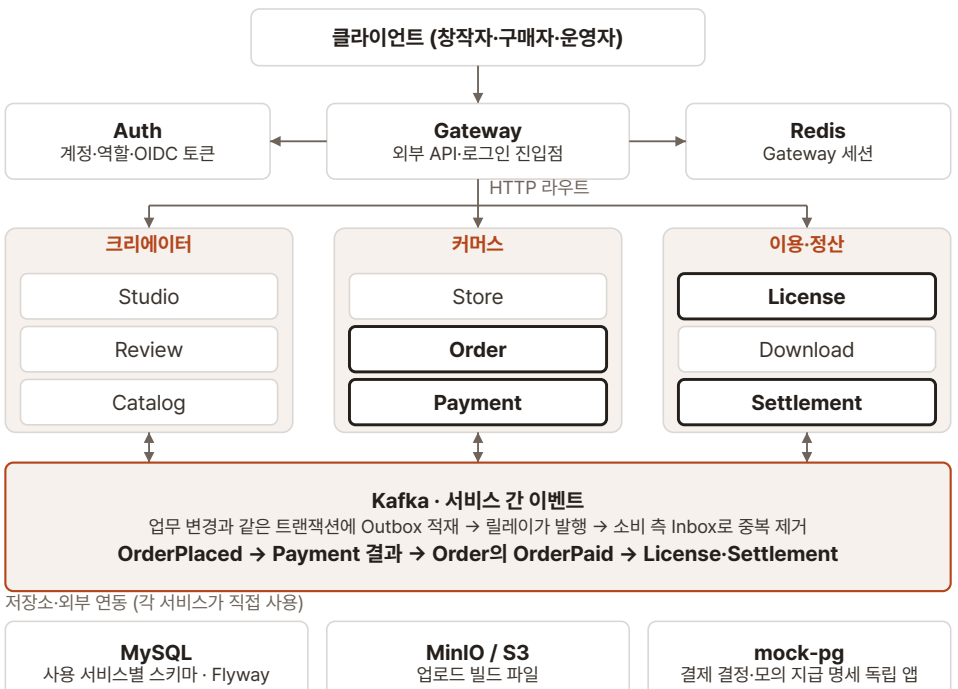
역할

단독 개발 프로젝트로, 설계 판단과 검증 기준을 직접 수립했습니다.

기술

Java 21 · Spring Boot · Spring Cloud · Kafka · MySQL · Redis · Testcontainers · ArchUnit · Prometheus · Grafana

`github.com/tlswltjq/esd-platform`



저장소 장애에 따른 오환불 방지

재시도 소진을 지급 실패로 단정하지 않고, 저장소 장애이거나 기존 라이선스가 확인되면 자동 환불 대신 DLT에 보류하도록 변경했습니다.

재작성 전 구현의 장애 주입 재시험에서 정상 결제 자동 환불이 8건에서 0건으로 줄었고, 보류된 주문은 재투입해 지급을 확인했습니다.

문제

저장소 장애 중 지급 결과를 확인할 수 없는 정상 결제 일부가 자동 환불되었습니다.

부하 테스트 중 License 원장 테이블의 접근 권한을 일시적으로 차단하고, 장애가 해소된 뒤 지급이 복구되는지 확인하는 장애 주입 시험에서 관찰했습니다.
License 스키마 전체를 차단한 다른 회차의 로그에서는 이미 지급된 주문도 이벤트 재배달과 재처리 실패를 거쳐 보상 경로에 진입한 것을 확인했습니다.
이 회차는 보상 이벤트 기록도 실패해 DLT로 이동했습니다. 원장 접근만 차단한 조건에서 이미 지급된 주문이 실제 환불되는 상황은 재현하지 못했습니다.

원인

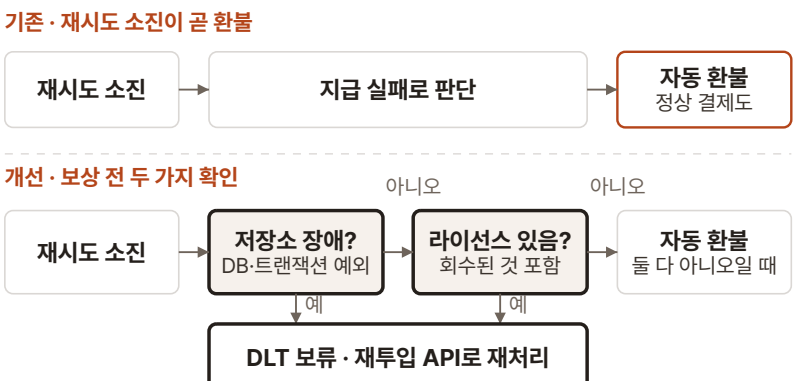
재시도 소진을 지급 실패로 간주해 보상을 시작한 것이 원인이었습니다.

기존 로직은 License의 이벤트 처리 재시도가 모두 소진되면 보상 이벤트를 Outbox에 기록하고, Payment가 이를 받아 자동 환불했습니다.
재시도 소진만으로는 지급 실패를 확정할 수 없지만, 저장소 장애로 결과를 알 수 없는 상황과 이미 라이선스가 지급된 상황을 구분하지 않았습니다.

설계 판단

지급 결과가 불확실하면 자동 환불보다 보류가 안전합니다.

잘못된 환불은 되돌리기 어렵지만, 보류한 주문은 장애가 해소된 뒤 DLT 재투입으로 다시 처리할 수 있습니다. 이에 따라 저장소 장애 판정 범위를 의도적으로 넓게 설정했습니다.
재시도 횟수만 늘리는 방법은 장애가 더 길어지면 같은 문제가 반복되고 파티션 처리도 지연되어 선택하지 않았습니다.



저장소 장애에 따른 오환불 방지

8건
수정 전 자동 환불된 정상 결제

0건
수정 후 자동 환불

7건
수정 후 DLT 보류

1초 안
수정 후 보류 건 재투입~전부 지급

측정 조건: 재작성 전 구현 · 원장 권한 **60초** 차단 · 회차당 시험 주문 **40건** · 수정 전·후 **조건당 1회** · **1초 안**은 재투입부터 전부 지급까지의 관찰값

구현 핵심

- **저장소 장애 판별**: 리스너 예외의 원인 사슬에 DB·트랜잭션 예외가 있으면 자동 환불하지 않고 DLT에 보류합니다.
- **기존 지급 확인**: 주문에 연결된 라이선스가 이미 있으면, 회수된 상태여도 자동 환불하지 않습니다.
- **재처리 경로**: 보상하지 않은 레코드는 DLT에 남아, 기존 HTTP 재투입 API로 다시 처리할 수 있습니다.

상태와 불변식

- 재시도가 소진되고, 저장소 장애가 아니며, 기존 라이선스가 없는 경우에만 보상 이벤트를 기록합니다.

검증

- **장애 주입**: 수정 전·후의 지급·환불·보류 건수를 DB에서 직접 집계했습니다.
  License 원장 테이블만 차단(outbox_event·inbox 정상)했고, 주입 직후 프로브로 장애 발생과 Order 무영향을 확인했습니다.
- **재투입**: DLT 보류 건을 재투입 API로 넣어 지급을 확인했습니다.
- **누적 대사**: 결제 완료와 라이선스가 **115건 대 115건**으로 일치했습니다.
  여러 회차를 합산한 기준이라, 회차당 장애 주입 결과와 집계 범위가 다릅니다.
- **단위 테스트**: 저장소 장애와 이미 지급된 주문에서 보상 없이 보류하는 것을 확인했습니다.
  DB 장애, 트랜잭션 생성 실패, 감싸인 SQL 예외를 다룹니다. 커넥션 풀 고갈 시 환불하던 기존 회귀 테스트는 저장소와 무관한 예외로 보상을 확인하도록 바꿨습니다.

보장 범위와 한계

- **미검증**: 장애 주입은 재작성 전 구현에서 **조건당 1회** 측정했고, 현재 구현에서는 다시 측정하지 않았습니다.
- **미검증**: 이미 지급된 주문의 실제 환불은 재현하지 않았습니다.
- **설계 한계**: DLT 보류 건은 사람이 제때 확인하고 재투입해야 합니다.

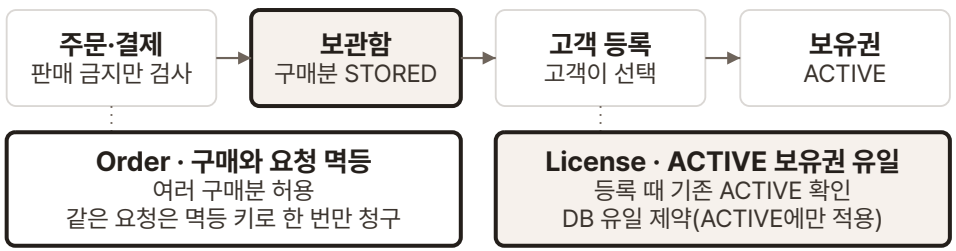
환불 후 재구매 허용과 보유권 중복 방지

기존 주문 때문에 환불한 상품을 다시 구매하지 못하던 문제를 해결했습니다. 재구매를 허용하면서도 같은 고객·상품의 활성 보유권은 최대 하나로 제한했습니다.

기존 · 과거 주문 상태로 신규 구매 제한



개선 · 여러 구매분 허용 / ACTIVE 보유권 최대 하나



문제

환불을 완료한 상품도 다시 구매할 수 없는 문제가 있었습니다.

재구매 요청은 '이미 구매한 상품입니다.'라는 응답으로 거절되었습니다. 주문 생성 과정에서 같은 고객·상품의 PAID 주문이 있으면 새 주문을 만들지 않았기 때문입니다.

원인

환불 후에도 PAID로 남은 기존 주문을 주문 생성 조건이 구매 완료로 판단했습니다.

또한 결제 시 License가 보유권을 즉시 생성했습니다. 따라서 구매 제한만 제거하면 같은 상품의 보유권이 중복 생성될 위험이 있었습니다.

설계 판단

구매분(STORED)과 활성 보유권(ACTIVE)의 책임을 분리했습니다.

결제된 구매분은 보관함(STORED)에 저장하고, 고객이 등록할 때 License가 기존 ACTIVE 보유권을 확인합니다. 같은 상품을 여러 번 구매해도 구매분을 따로 관리하면서, 활성 보유권은 하나만 유지할 수 있습니다.

대신 처음 구매한 상품도 등록 절차가 필요하고, 환불 실패 시 복구 경로가 복잡해지는 비용을 감수했습니다.

환불 후 재구매 허용과 보유권 중복 방지

구현 핵심

- **구매와 보유권 분리**: 결제된 구매분은 보관함(STORED)에 두고, 고객이 등록할 때 License가 보유권을 만듭니다.
  (buyer_subject, request_id) 멱등 키로 같은 요청의 중복 청구를 막고, Order는 판매 가능 여부만 확인합니다.
- **등록 검증**: 보관함 상태가 아니거나 같은 상품의 ACTIVE 보유권이 있으면 등록을 거절합니다.
- **DB 유일 제약**: 생성 열로 고객·상품당 ACTIVE 보유권을 최대 하나로 제한합니다.
- **잠금 후 재조회**: 같은 고객·상품의 구매 근거를 id 순서로 잠근 뒤 다시 읽어, 환불 중인 구매분을 오래된 상태로 덮어쓰지 않습니다.
- **환불 실패 시 복구**: 잠금 안에서 다른 ACTIVE 보유권을 확인하고, 있으면 기존 구매분을 보관함으로 되돌려 중복 활성화를 막습니다.

상태와 불변식

- 구매 근거: STORED · REGISTERED · 환불 진행 중 SUSPENDED · 환불 완료 REFUNDED
- 보유권: ACTIVE · 환불 진행 중 SUSPENDED · 환불 완료 ENDED
- **불변식: 같은 고객·상품의 ACTIVE 보유권은 환불과 복구가 엇갈려도 최대 하나만 존재합니다.**

검증

- **수정 전 문제 재현**: PAID 주문이 있으면 새 주문이 거절되는 기존 동작을 확인했습니다.
  PR 본문에 기록된 테스트 결과입니다.
- **License 통합 테스트**: 병합 후 main CI에서 License 통합 테스트 태스크가 성공했습니다.
  반복 구매, 환불 중 재등록 후 복구, 환불 완료 후 늦은 복구, 동시 등록 시나리오를 다룹니다.
- **실제 스택 시나리오**: 환불 완료 후 재구매·등록 단계를 검증 도구에 추가했습니다.
  통과 여부는 PR 본문의 기록입니다. 실제 PG 대신 PG 환불 결과 이벤트만 주입했습니다.

보장 범위와 한계

- **미검증**: 통합 테스트는 운영 코드 뒤에 추가해 처음부터 통과했으며, 기존 코드에서 실패하는지는 확인하지 않았습니다.
- **미검증**: 실제 PG 결제·환불과 PG 취소 구간은 실행하지 않았습니다.
- **설계 한계**: 선물은 정책이 정해지지 않아 구현하지 않았고, 고객 알림도 만들지 않았습니다.

Outbox 릴레이의 키별 발행 순서 역전 방지

> **선행 이벤트 발행에 실패해도 같은 키의 후행 이벤트가 추월하지 않도록, 배치 내 발행 순서와 다음 폴링의 조회 조건을 단계적으로 개선했습니다.**

문제

선행 이벤트 발행에 실패해도 같은 키의 후행 이벤트가 발행되었습니다.

수정 전 테스트에서 선행 이벤트는 PENDING으로 남았지만, 같은 키(partitionKey)의 후행 이벤트는 SENT로 처리되는 결함을 재현했습니다. 지급 완료보다 취소 이벤트가 먼저 전달되면 환불한 주문에 라이선스가 남을 위험이 있습니다.

원인

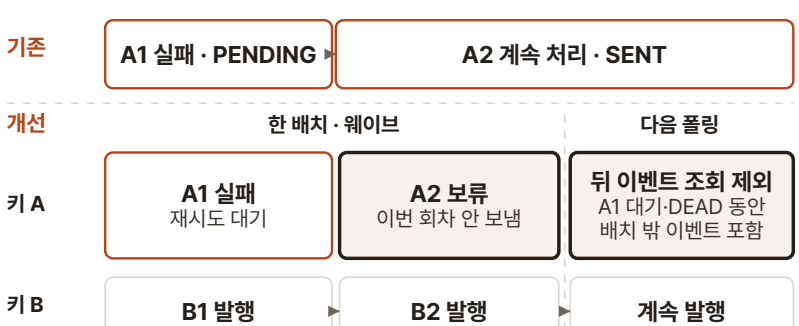
Kafka의 순서 보장만으로는 릴레이의 발행 순서 역전을 막을 수 없습니다.

Kafka는 파티션에 기록된 순서만 보장하므로, 릴레이가 후행 이벤트를 먼저 발행하면 이를 바로잡지 못합니다.
키별 웨이브 발행으로 같은 배치 안의 역전은 막았지만, 선행 이벤트가 재시도 대기로 조회에서 빠지면 다른 배치의 후행 이벤트가 먼저 선택될 수 있었습니다.

설계 판단

같은 키의 순서만 제어하고, 다른 키의 발행은 계속 허용했습니다.

전체 배치를 멈추면 무관한 주문까지 지연되기 때문입니다.
CDC는 커넥터 운영 부담이 있고, 수신 측 재정렬은 소비자마다 진행 상태와 보류 버퍼가 필요해 선택하지 않았습니다.



Outbox 릴레이의 키별 발행 순서 역전 방지

구현 핵심

- **배치 안 순서**: 같은 키는 선행 이벤트의 성공을 확인한 뒤 다음 이벤트를 발행합니다.
  배치를 키별로 묶어 다른 키는 병렬로 발행하고, 선행 이벤트가 실패한 키는 그 회차에 후행 이벤트를 보내지 않습니다.
- **초기 보완(next_attempt_at 전파)**: 선행 이벤트가 실패하면 같은 배치의 후행 이벤트도 재시도 시각까지 조회에서 제외했습니다.
  다만 배치에 포함되지 않은 후행 이벤트는 여전히 먼저 발행될 수 있었습니다.
- **후속 보완(NOT EXISTS 조회 조건)**: 같은 키의 선행 이벤트가 재시도 대기 중이거나 DEAD이면 후행 이벤트를 조회하지 않습니다.
  배치에 포함되지 않았던 이벤트에도 적용됩니다.
- **보류와 DEAD**: 순서를 위한 보류는 발행 실패가 아니므로 재시도 횟수를 올리지 않습니다.
  초기 구현은 선행 이벤트가 DEAD가 되면 키를 풀었지만, 현재는 운영자가 회수(requeue)할 때까지 해당 키를 멈춥니다.

검증

- **배치 안 역전**: 수정 전 테스트 실패와 수정 후 통과를 확인했습니다.
  KafkaTemplate 대역으로 선행 이벤트의 전송 실패를 주입한 단위 테스트입니다. 실제 Kafka 기록 순서는 확인하지 않았습니다.
- **다음 폴링 역전**: 초기 보완의 회귀 테스트가 통과했습니다.
  같은 배치에서 시작한 후행 이벤트만 다룹니다. 수정 전 실패는 커밋에 기록된 내용입니다.
- **조회 조건 보완**: 해당 PR의 CI에서 실제 MySQL 통합 테스트 태스크가 성공했습니다.
  재시도 대기·DEAD인 선행 이벤트가 나중에 적재된 후행 이벤트를 막는지 확인합니다. 수정 전 실패는 결함 대장에 기록된 내용입니다.
- **회귀 테스트**: 다른 키 비차단과 보류 이벤트의 재시도 횟수 유지를 단위 테스트로 확인했습니다.

보장 범위와 한계

- **미검증**: 실제 Kafka 브로커의 순서 역전과, 환불 후 라이선스 잔존 같은 서비스 간 영향은 재현하지 않았습니다. 검증은 릴레이와 조회 계층의 테스트입니다.
- **설계 한계**: 릴레이 한 대를 전제로 조회·발행 제어 조건을 설계했습니다. 여러 대에서는 같은 키가 나뉘어 처리될 수 있으며, 이 경우의 순서는 확인하지 않았습니다.
- **설계 한계**: 선행 이벤트가 DEAD로 남으면 운영자가 회수할 때까지 같은 키의 후행 이벤트가 지연됩니다.

프로젝트

수집형 RPG 게임 서버 백엔드

2026-05 – 2026-07

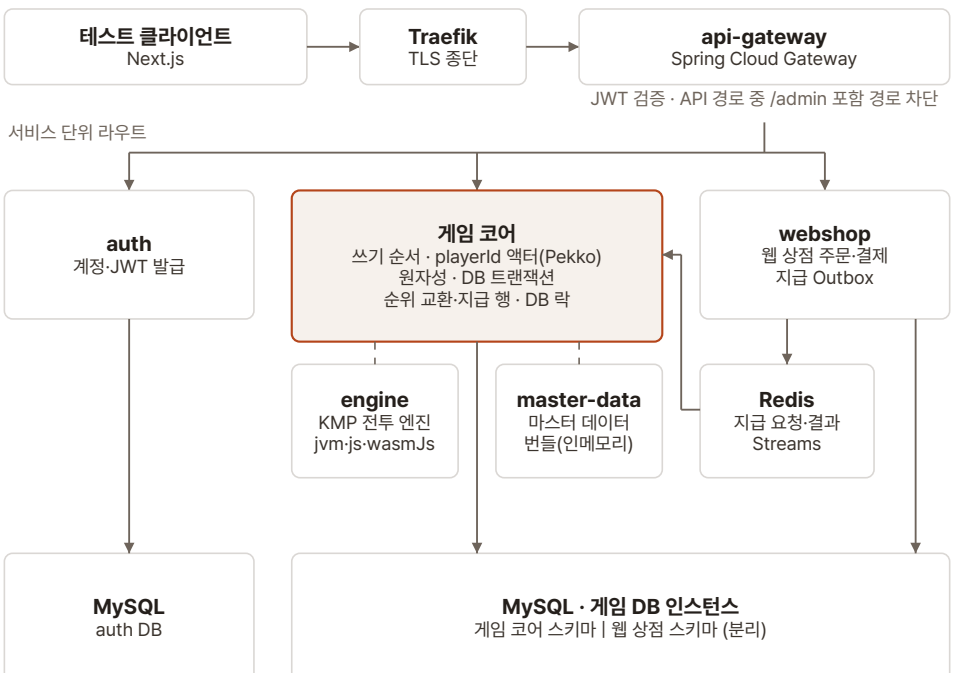
수집형 RPG의 전투·뽑기·시즌 정산과 웹 상점을 여러 서버로 나눠 재구현한 게임 서버

개요

멀티모듈 구성: Spring Cloud Gateway 뒤에 auth, 게임 코어, 웹 상점(webshop) 서버 모듈을 두고, 전투 엔진(engine)과 마스터 데이터(master-data)는 라이브러리 모듈로 나눈 Gradle 멀티모듈 백엔드입니다.

플레이어 쓰기 처리: 게임 코어에서는 같은 플레이어의 쓰기 커맨드를 PlayerActors가 playerId별 Apache Pekko Cluster Sharding 엔티티로 보내고, 엔티티가 메일박스에서 차례로 실행합니다.

웹 상점 지급: 웹 상점은 결제 원장을 별도 스키마에 두고 Outbox와 Redis Streams로 게임 코어에 지급을 요청합니다.



범위

- 게이트웨이 라우팅·인증 경계와 서버 모듈 분리
- 플레이어 액터 쓰기 직렬화와 시즌 정산 트랜잭션 경계
- jvm·js·wasmJs에서 같은 골든 해시를 검사하는 결정론 전투 엔진

역할

백엔드 단독 개발. 기능 기획과 게임 규칙 정의, 설계 방향 결정, 테스트 서버 배포를 담당했습니다.

기술

Kotlin · Spring Boot · Spring Cloud Gateway · Apache Pekko · Kotlin Multiplatform · Spring Data JPA · MySQL · Redis · Testcontainers · k6

단계·청크 체크포인트 커밋으로 정산 롤백 범위 축소

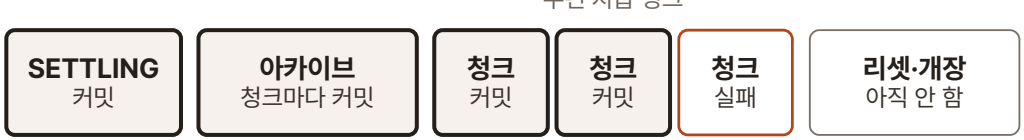
> **트랜잭션 하나로 묶여 있던 시즌 정산을 단계·청크마다 따로 커밋하도록 바꿔, 지급 도중 실패해도 앞서 커밋한 진행분이 남게 했습니다.**

기존 · 트랜잭션 하나



우편 지급 중 실패 → 경계 전체가 롤백, 이어 붙일 상태 없음

개선 · 단계·청크마다 커밋



같은 지점 실패 → 실패한 청크만 롤백

굵은 테두리의 커밋은 실패 뒤에도 남아 재개의 출발점이 된다

개념도: 청크 수는 설명용

문제

지급 도중 실패하면 앞서 처리한 정산 단계까지 모두 롤백됐습니다.

테스트로 재현: SETTLING 전이와 아카이브까지 함께 롤백되어 이어서 처리할 중간 상태가 남지 않았습니다.
코드·커밋 분석: 재개용 멱등 장치(지급 상태 스킵, 존재 스킵)는 쓰일 기회가 없었고, 목 기반 테스트는 롤백을 반영하지 못해 도달할 수 없는 재개 로직이 통과했습니다.

원인

시즌 정산 전체가 하나의 트랜잭션에 묶여 있었습니다.

진행분은 실패해도 남아야 하고, 쓰기 동결(SETTLING)은 정산 중 다른 요청에 보여야 합니다. 코드 분석 결과, SETTLING은 커밋 전까지 다른 커넥션에 보이지 않고(READ_COMMITTED) 리셋의 전체 삭제까지 정산 내내 잠금이 유지됩니다.

설계 판단

정산을 단계·청크별 트랜잭션으로 분리해 진행분을 커밋하도록 변경했습니다.

재개 로직을 보강해도 경계가 하나면 재개에 필요한 중간 상태가 함께 롤백되므로, 경계 자체를 나눴습니다.
경계는 @Transactional(REQUIRES_NEW) 메서드 대신 TransactionTemplate으로 열었습니다. 같은 빈 안의 자기 호출은 프록시를 우회해 선언한 경계가 조용히 사라질 수 있기 때문입니다.
부분 커밋이 생기므로 중복 반영 방지와 명시적 재개를 함께 두었습니다.

단계·청크 체크포인트 커밋으로 정산 롤백 범위 축소

구현 핵심

- **체크포인트 커밋**: 단계·청크마다 별도 트랜잭션으로 커밋합니다.
  SettleSeason의 @Transactional을 제거하고, 트랜잭션 경계에는 TransactionTemplate(REQUIRES_NEW)을 사용합니다.
- **아카이브 중복 방지**: 이미 옮긴 성적은 건너뜁니다.
  (seasonId, playerId) 유니크 제약으로 중복을 막습니다.
- **지급 중복 방지**: 우편 발행과 지급 완료(SENT) 표시를 한 트랜잭션에 둡니다.
  지급 대기 청크는 잠금 읽기로 선점합니다.
- **명시적 재개**: 관리자가 resume=true로 명시할 때만 이어 붙입니다.
  아직 실행 중인 정산과 겹치면 이중 지급될 수 있기 때문입니다.

상태와 불변식

- 체크포인트 하나가 커밋 하나입니다. 실패는 진행 중인 단계·청크만 롤백합니다.
- 아카이브 행은 (seasonId, playerId)당 하나입니다.
- 우편을 발행한 지급 건은 같은 트랜잭션에서 SENT가 되고, 재개 때 건너뜁니다.
- SETTLING인 시즌은 resume 없이 다시 정산에 들어가지 않습니다.

검증

- **재현·대조 테스트**: 이 테스트를 더한 커밋의 CI test 잡이 성공했습니다.
  단일 트랜잭션 버전과 같은 지급 지점에서 실패시켜, 기존 방식은 아카이브까지 롤백되고 새 방식은 진행분이 남는지 단언합니다.
  테스트별 결과는 로그 만료로 확인하지 못했습니다.
- **기능 테스트**: 지급 중간 실패 뒤 앞 청크가 남는지 단언합니다.
  기존 목 테스트는 H2 실트랜잭션 테스트로 바꿨습니다.

보장 범위와 한계

- **미검증**: 모두 내장 H2 기능 테스트입니다. MySQL의 잠금·격리 동작과 처리 시간은 검증하지 않았습니다.
- **미검증**: SETTLING 노출과 라이브 테이블 잠금 축소는 재현 테스트가 없는 코드 분석 결과입니다.
- **설계 한계**: 러너가 살아 있는지 자동으로 판별하지 않아, 재개 여부는 사람이 정해야 합니다.

조건부 UPDATE로 정산 시작 경합 해결

> **정산 시작 때의 상태 확인과 전이를 'ACTIVE일 때만 SETTLING으로 바꾸는' 조건부 UPDATE 한 문장으로 합쳐, 재개가 아닌 새 정산에서는 행을 갱신한 요청 하나만 정산에 들어가게 했습니다.**

문제

동시 정산 요청에서 서버 오류와 데드락이 발생했습니다.

사용자 진술: 정산 요청을 동시에 보내자 서버 오류가 났고, 로그에서 season_standings 유니크 인덱스 데드락을 확인했습니다.

기존 동작

정산 시작 권한을 여러 단계의 조회와 상태 판정으로 결정했습니다(코드 확인).

수정 전 판정과 전이는 한 트랜잭션(REQUIRES_NEW) 안에서 '잠금 없는 읽기 → PESSIMISTIC_WRITE(FOR UPDATE) 재조회 → 엔티티 상태로 판정 → 저장' 순서로 나뉘어 있었습니다.

원인

잠금 전에 읽은 ACTIVE 상태가 판정에 쓰였을 가능성이 있습니다(추론, 미검증).

커밋 기록의 분석: 여러 요청이 함께 ACTIVE 상태를 확인하고 병렬로 정산에 진입하면서, 아카이브 과정에서 데드락이 발생한 것으로 분석했습니다. 기존 목 테스트는 SETTLING을 직접 넣어 이 경로를 거치지 않았습니다.

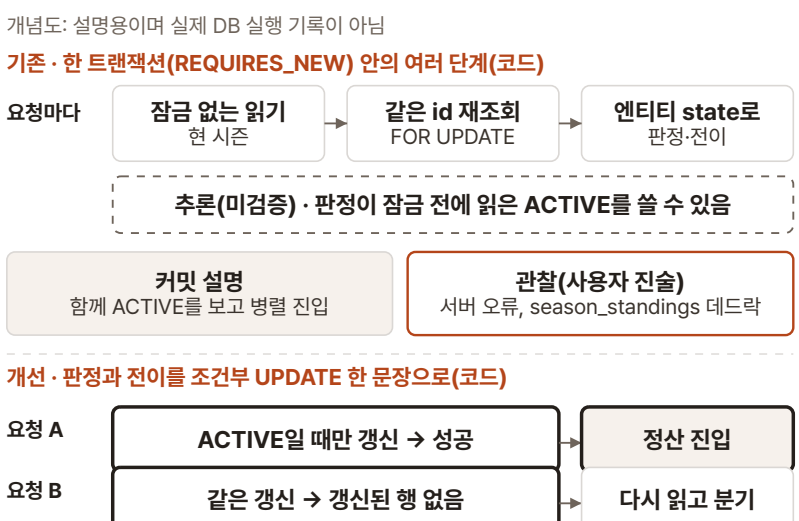
추론: Hibernate가 영속성 컨텍스트의 엔티티를 잠금 조회 결과로 새로 고치지 않으면, FOR UPDATE로 SQL은 차례로 실행돼도 판정은 잠금 전에 읽은 ACTIVE를 쓸 수 있습니다.

미확인: 두 요청이 모두 정산 본체에 들어갔는지, 병렬 아카이브가 데드락의 직접 원인인지는 기록이 없습니다.

설계 판단

상태 확인과 전이를 조건부 UPDATE 한 문장으로 합쳤습니다.

WHERE state = ACTIVE 조건과 InnoDB 행 락으로 경합 창 없이, 갱신된 행 수만으로 정산 시작 권한을 얻은 요청을 가릅니다. 대가로 경합에서 밀린 요청은 DB 상태를 다시 읽어 분기해야 합니다.



조건부 UPDATE로 정산 시작 경합 해결

구현 핵심

- **조건부 UPDATE**: 시즌이 아직 ACTIVE일 때만 SETTLING과 컷오프 시각을 씁니다.

  WHERE state = ACTIVE 조건으로 확인과 전이를 한 문장에서 처리합니다.
- **갱신 실패 분기**: 영속성 컨텍스트를 비우고 DB 상태를 다시 읽어 분기합니다.

  SETTLING이면 거절하고, SETTLED면 다음 시즌 개장만 하며, 다시 ACTIVE면 경합에서 밀린 것으로 보고 거절합니다.

상태와 불변식

- 새 정산에서 ACTIVE를 SETTLING으로 바꾸는 요청은 조건부 UPDATE로 행을 갱신한 하나뿐입니다.
- 경합에서 밀린 요청의 분기는 다시 읽은 DB 상태로 정합니다.
- 재개는 resume=true를 명시할 때만 SETTLING 시즌으로 들어갑니다.

검증

- **코드 확인**: 수정 후 판정과 전이는 조건부 UPDATE 한 문장이고, 수정 전은 두 단계 조회였습니다.

  커밋 전후 코드를 대조했습니다.
- **수동 재시험(사용자 진술)**: 수정 후에는 서버 오류가 다시 발생하지 않았습니다.

  수정 전 오류가 발생하던 방식으로 동시 요청을 수행했습니다.

보장 범위와 한계

- **미검증**: 자동 동시성 테스트가 없으며, 당시 부하 실행 기록과 로그도 저장소에 남아 있지 않습니다.
- **설계 한계**: 수정은 정산 진입 경합만 다룹니다. 정산 본체의 다른 동시성·데드락 가능성까지 없앤다고 보장하지 않습니다.
- **설계 한계**: SETTLING이 실행 중인지 멈춘 것인지 코드는 구분하지 못해, 재개 여부는 사람이 정해야 합니다.

플레이어 단위 액터 직렬화로 쓰기 경합 제어

> **같은 플레이어의 쓰기는 playerId 액터 하나로 직렬화하고, 원자성은 DB 트랜잭션에, 플레이어 간 순위 교환은 DB 락에 남겼습니다.**



문제

플레이어 쓰기 경합을 도메인별 DB 락으로 제어하고 있었습니다(코드 확인).

뽑기 카운터·일일 사용량·성물·생산 슬롯·스태미나 충전 행에 PESSIMISTIC_WRITE 잠금을 걸고, 행이 없을 때를 위한 REQUIRES_NEW 삽입 가드도 따로 두어, 새 쓰기 경로마다 같은 장치를 챙겨야 했습니다.
커밋에 적힌 위험(재현 기록 없음): 생산 슬롯에서 완성품이 이중 지급되거나 재료가 이중 소모될 수 있었습니다.

원인

같은 플레이어의 쓰기 진입점이 여러 컨트롤러에 분산되어 있었습니다.

진입점마다 각자 DB 트랜잭션을 열었고, 같은 플레이어의 쓰기를 차례로 처리하는 장치가 없어 행 단위 DB 락이 직렬화를 대신했습니다.
삽입 가드는 바깥 트랜잭션의 커넥션을 쥔 채 커넥션을 하나 더 사용합니다.

설계 판단

직렬화는 플레이어 액터에, 원자성은 DB 트랜잭션에 맡겼습니다.

액터가 더하는 것은 같은 플레이어 안의 직렬화뿐이므로 원자성은 계속 @Transactional이 맡고, 플레이어 간 순위 교환과 정산의 지급 대기 행 선점은 DB 락에 남겼습니다.
액터는 도메인별이 아니라 플레이어당 하나로 두었습니다. 도메인별로 나누면 뽑기가 재화·카운터·지급을 가로지르는 액터 간 사가가 되기 때문입니다.
액터 메모리를 정본으로 삼는 단계는 원자성 경계를 옮겨야 해 진행하지 않았습니다.

플레이어 단위 액터 직렬화로 쓰기 경합 제어

구현 핵심

- **플레이어 액터**: 쓰기 진입점을 PlayerActors로 모았습니다.
  playerId별 엔티티 하나(Pekko Cluster Sharding)가 메일박스에서 하나씩 실행합니다.
- **타입 있는 명령**: 쓰기는 PlayerCommand 값으로 보냅니다.
  핸들러가 명령 종류를 빠짐없이 분기(exhaustive when)해, 새 명령의 처리 누락이 컴파일 시점에 드러납니다.
- **락 제거**: 모든 쓰기 진입점이 액터를 거치는 것을 확인한 도메인부터 플레이어 범위 락을 지웠습니다.
  행 잠금(PESSIMISTIC_WRITE)과 REQUIRES_NEW 삽입 가드를 지운 뒤, 지갑·아이템·일일 입장의 락도 지웠습니다.
- **우회 경로 정리**: 액터를 우회하던 쓰기를 액터 경유로 바꿨습니다.
  대상은 컨트롤러의 직접 쓰기와 core의 예약 작업(@Scheduled)입니다.

상태와 불변식

- 같은 playerId의 쓰기는 모두 PlayerActors를 거칩니다. 우회 진입점이 하나라도 있으면 직렬화가 깨집니다.
- 플레이어 범위 행 락은 그 도메인의 모든 진입점이 액터를 거친다고 확인한 뒤에만 지웁니다.
- 같은 플레이어 액터는 한 프로세스에만 떠 있어야 합니다(단일 노드 전제).

검증

- **정적 규칙 검사(ArchUnit)**: 이 규칙이 포함된 main CI test 잡이 성공했습니다.
  규칙은 api 계층이 재화를 옮기는 core 메서드를 직접 호출하거나 core에 @Scheduled를 두면 실패합니다.
- **코드 경로 확인**: 플레이어 쓰기가 PlayerActors를 거쳐 액터 하나로 들어갑니다.
  main HEAD 코드를 확인했습니다.

보장 범위와 한계

- **미검증**: ArchUnit 규칙은 구조 검사이지 동시성 검증이 아닙니다. 동시 부하·통합 테스트 기록이 없고, 서버는 테스트용으로만 띄워 실제 동시 사용자 환경에서 검증하지 않았습니다.
- **설계 한계**: 쓰기 권위는 여전히 DB입니다.
- **설계 한계**: 단일 노드 전제입니다. 같은 플레이어 액터가 두 프로세스에 동시에 뜨면 직렬화가 깨지므로, 블루그린 대신 중단 배포로 바꿨습니다(스플릿브레인은 추정 위험).

프로젝트

멘토링·채용 정보 서비스 백엔드

2025-09 – 2025-10

멘토링 예약, 화상·채팅 멘토링, 영상 뉴스를 제공하는 멘토링·채용 정보 서비스

개요

Spring Boot 백엔드에 비동기 영상 처리와 실시간 통신을 붙인 구조입니다.

비동기 영상 처리: 클라이언트가 영상을 S3에 직접 올리고, Python 워커가 화질별 DASH로 변환한 뒤 진행 상태를 Kafka로 알립니다.

실시간 통신: 멘토링 세션의 채팅과 WebRTC 시그널링은 STOMP로 중계합니다.

범위

- S3 직접 업로드, SNS 브리지, Kafka·FFmpeg DASH 변환 파이프라인
- STOMP 연결 인증과 채팅·WebRTC 시그널링 중계
- 뉴스·댓글 도메인

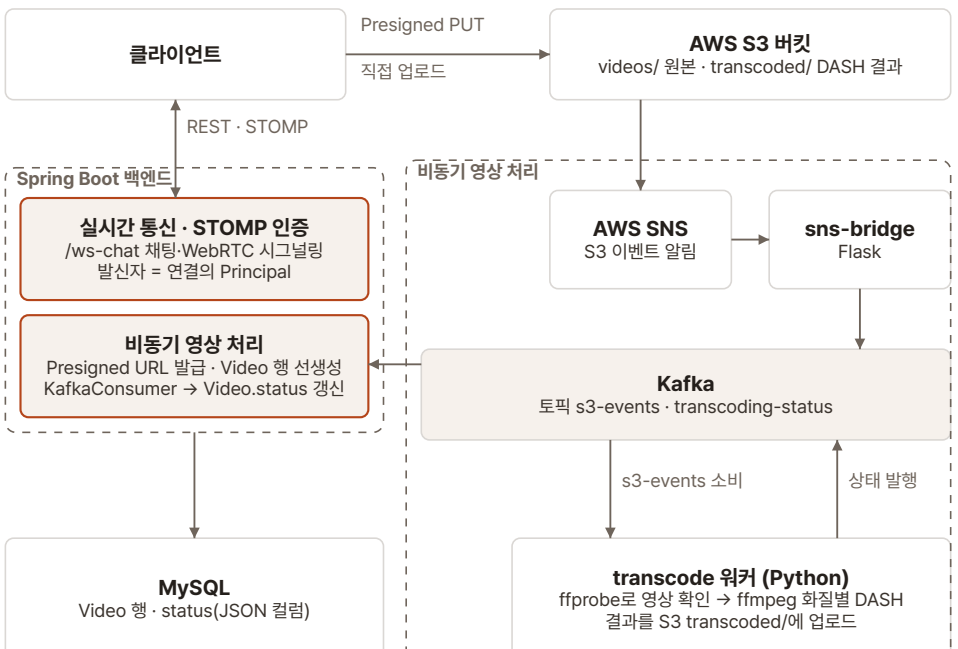
역할

미디어 처리 파이프라인과 실시간 시그널링을 담당하고, 뉴스·댓글 도메인을 구현했습니다.

기술

Java 21 · Spring Boot · Spring WebSocket(STOMP) · Kafka · AWS S3 · Python · Flask · FFmpeg

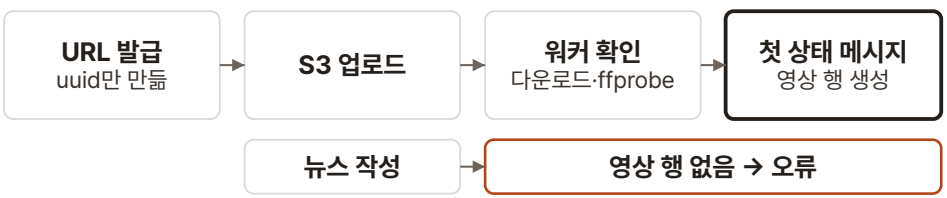
`github.com/prgrms-web-devcourse-final-project/WEB6_8_FiveLogic_BE`



영상 행 선생성으로 비동기 변환과 뉴스 작성 분리

> **영상 행 생성을 URL 발급 시점으로 앞당겨, 뉴스 생성 시 영상 행 조회가 워커의 첫 상태 메시지에 의존하지 않도록 구조를 변경했습니다.**

기존



개선



문제

워커의 첫 상태 메시지 전에는 영상 행이 없어 뉴스 작성이 실패했습니다.

관찰: Video not found 오류. 행 부재 로그 확인은 사용자 진술입니다.

원인

영상 행을 URL 발급이 아닌 워커의 첫 상태 메시지에서 생성했습니다.

구현 핵심

- **영상 행 선생성**: URL 발급 시 uuid에 대응하는 Video 행을 만들고, 워커의 첫 상태 메시지는 그 행의 상태를 갱신하도록 변경했습니다.
- **식별자·다운로드 응답**: 응답에 uuid를 넣고, 다운로드는 uuid·화질로 Presigned URL을 줍니다.

검증

- **컨트롤러 테스트**: main test 잡 통과가 조건인 릴리스가 생성됐습니다.
  개별 실행 기록은 조회할 수 없습니다.

설계 판단

영상 행 생성 시점을 URL 발급 단계로 앞당겼습니다. 식별자(uuid)는 URL 발급 때 정해지므로 행도 그때 만들고, 비동기 변환은 그 행의 상태만 갱신합니다. 대가로 업로드하지 않은 URL의 행도 남습니다.

보장 범위와 한계

- **미검증(수정 후 동작)**: 영상 행 생성 시점을 변경했지만, 수정 후 뉴스 작성이 실제로 성공한 기록은 없습니다.
- **미검증**: MockMvc 컨트롤러 테스트는 URL 응답과 권한만 확인합니다. 파일 처리 계층을 목으로 바꿔 행 선생성·화질 검증·없는 영상 분기는 검증하지 않습니다.
- **구현에 남은 한계**: 상태 갱신 중 어떤 예외든 새 행을 만드는 catch 분기가 남아 있습니다(코드 관찰).
- **구현에 남은 한계**: 미업로드 URL의 행을 정리하는 코드가 없고, 다운로드 전 확인은 변환 완료 여부를 보지 않습니다(코드 관찰).

STOMP 메시지 발신자를 연결 인증 사용자로 고정

> **STOMP 메시지의 발신자를 본문의 from이 아니라, CONNECT 때 토큰으로 확인한 연결의 Principal로 정하도록 바꿨습니다.**

문제

HTTP 인증 정보가 STOMP 처리에서 유지되지 않아 발신자를 식별할 수 없었습니다.

관찰: HTTP 요청에서 생성한 인증 객체가 핸드셰이크 이후 유지되지 않았습니다.

추론: 이 상태에서 클라이언트가 보낸 from 값을 신뢰하면 다른 사용자로 위장할 위험이 있습니다.

원인

HTTP 요청과 STOMP 메시지 처리가 서로 다른 스레드에서 이루어졌습니다.

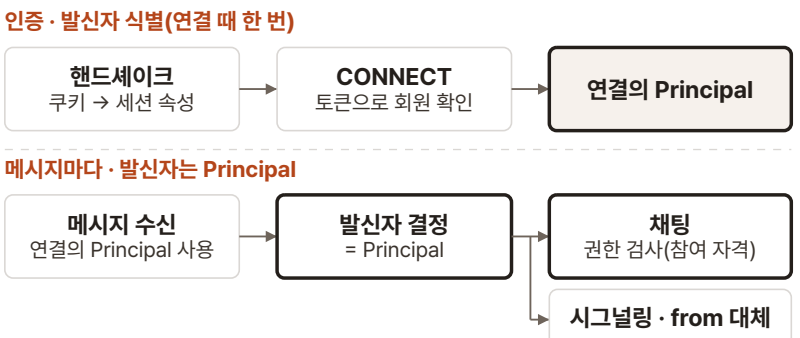
SecurityContextHolder는 요청 스레드에 묶이지만, STOMP 메시지는 다른 스레드에서 처리됩니다. 처리 채널은 clientInboundChannel입니다(종료 뒤 재구현으로 확인).

설계 판단

메시지 발신자는 클라이언트 본문이 아닌 서버가 확인한 Principal로 결정합니다.

구현 핵심

- **연결 인증**: 핸드셰이크 때 쿠키의 accessToken을 세션 속성에 두고, CONNECT에서 확인해 Principal을 설정합니다. 토큰·회원이 없으면 연결을 거부합니다.
- **발신자 결정**: 시그널링은 클라이언트의 from을 버리고 Principal로 메시지를 다시 만들고, 채팅은 Principal로 회원을 조회합니다.
- **참여 자격 검사**: ChatMessage.create에서 발신자가 예약의 멘토·멘티인지 검사합니다.



검증

- **단위 테스트**: main test 잡 통과가 조건인 릴리스가 생성됐습니다.
  ChatMessageTest는 참여자가 아닌 발신자의 예외를 검사합니다. 실행 기록은 조회할 수 없습니다.
- **수동 시험**: 결과는 따로 기록되지 않았습니다.
  시연 스크린샷과 로컬 시험 조건을 PR에 공유했습니다.

보장 범위와 한계

- **미검증**: 인터셉터와 시그널링 컨트롤러에는 자동 테스트가 없습니다.
- **설계 한계**: 시그널링에는 참여자 검사도, SUBSCRIBE 목적지 검사도 없습니다(코드 관찰).
- **설계 한계**: 인증은 CONNECT 때만 하고, 연결 중 토큰 만료는 다시 확인하지 않습니다.