BACKEND PORTFOLIO

염영인

Node.js · TypeScript 백엔드 엔지니어

경력 5년 5개월 · 최근 주식회사 로워드

주어진 문제를 깊이 이해하고, 포기하지 않고 해결까지 완성하는 개발자 염영인입니다. 다양한 B2G·B2C 백엔드 플랫폼 구축 및 고도화 경험을 보유한 Node.js 개발자입니다.

01

PG 연동·결제 정합성

결제 멱등성 및 상태 정합성 확보

02

메시지 큐·장애 격리

메시지 큐 기반 비동기 처리 시스템 구축

03

캐시 설계·무효화

서비스 · 데이터 특성에 맞는 Redis 캐시 설계와 무효화 전략

04

AI 활용 개발·FDE

Harness Engineering 기반 AI 개발 환경 구축

duaduddls1919@gmail.com · github.com/Yeom-YoungIn · threezerosin.tistory.com/category

01

PG 연동·결제 정합성

결제 멱등성 및 상태 정합성 확보

GuideME, 퍼플아카데미, 로워드에서 PG 결제부터 결제 검증, 후속 비즈니스 로직까지 이어지는 결제 프로세스 구현. 중복 요청과 동시성 문제에 대응해 결제 멱등성을 확보하고, PG 결제 상태와 서비스 내 주문·예약·입학 상태를 연동해 데이터 정합성 유지.

사례	PG 결제 시스템
기간	2024.12 — 2026.10
역할	로워드 · 퍼플아카데미 · GuideME · 백엔드
기술	Toss Payments, PostgreSQL, TypeORM, Idempotency

다루는 프로젝트

프레스런
중복 요청에도 결제 결과가 한 번만 반영되도록 처리

퍼플아카데미
여러 PG 상점과 결제 방식에 대응하는 구조

GuideME
예약 충돌을 방지하고 정산까지 이어지는 결제 처리

프레스런 — 중복 요청에도 결제 결과가 한 번만 반영되도록 처리

문제 상황

프레스런에서는 Toss Payments로 카드 결제, 가상계좌, 분할결제, 결제 수단 변경 기능을 구현했습니다. 결제 방식이 다양해지면서 하나의 결제에 여러 승인 시도와 상태 변경이 생길 수 있었습니다.

예를 들어 사용자가 결제 완료 후 페이지를 새로고침하거나, 가상계좌 입금 통보와 결제 기한 만료 배치가 비슷한 시점에 실행되는 상황을 고려해야 했습니다. 이때 같은 결제 건이 중복 처리되거나, 실제 결제는 완료됐는데 서비스에서는 실패로 기록되지 않도록 하는 것이 중요했습니다.

이력 분리

먼저 결제 정보와 개별 결제 시도를 구분해 관리했습니다.

payments: 하나의 결제에 대한 정보

payment_transactions: 분할결제 회차, 승인 시도, 결제 수단 변경 등에 따른 개별 결제 건

이렇게 나누면 하나의 결제에서 여러 차례 시도가 생겨도 각각의 이력을 남길 수 있습니다. 또 PG 주문번호와 (결제 ID, 회차, 시도) 조합에 Unique 인덱스를 걸어, 애플리케이션의 중복 확인과 별개로 DB에서도 같은 결제 건이 중복 생성되지 않도록 했습니다.

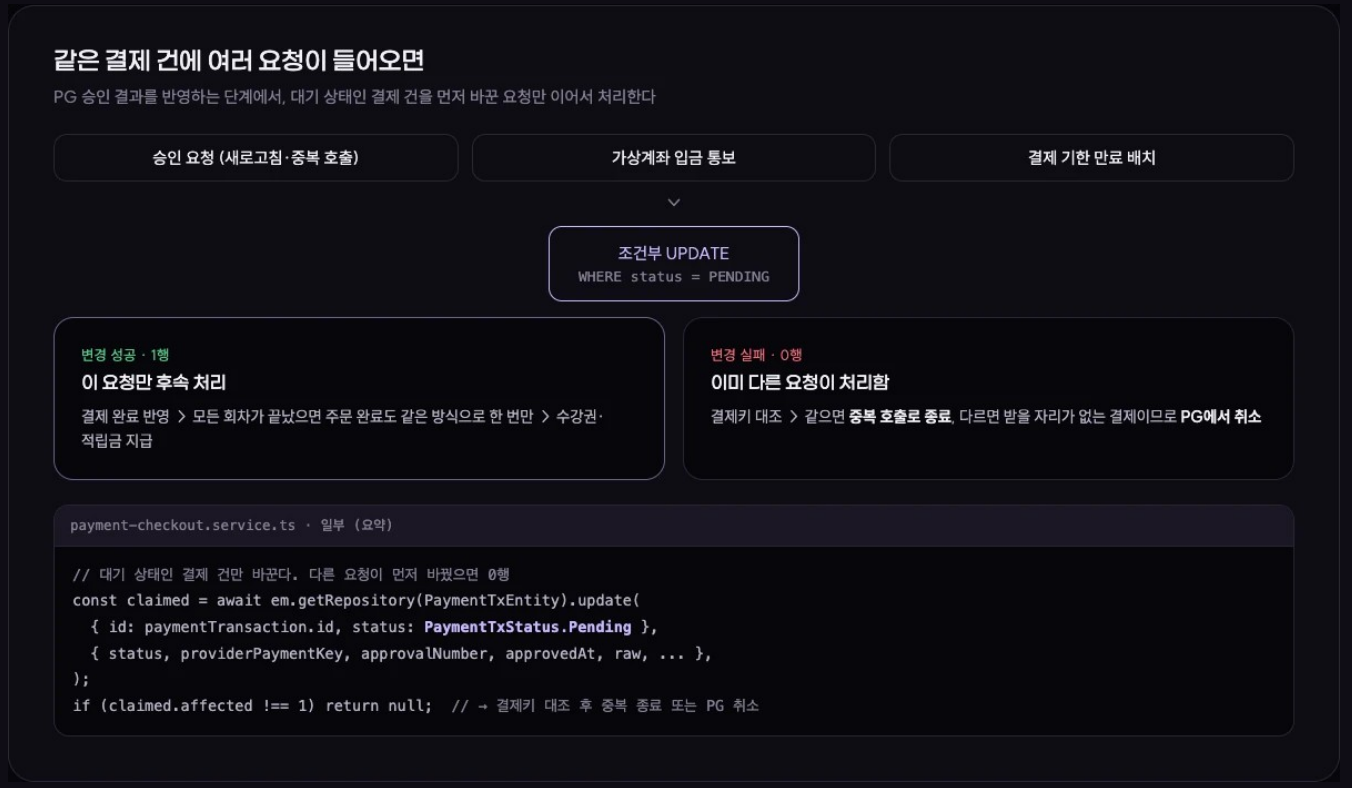
프레스런 — 중복 요청에도 결제 결과가 한 번만 반영되도록 처리 (계속)

동시 요청

결제 승인 요청, 가상계좌 입금 통보, 결제 기한 만료 배치는 모두 같은 결제 건의 상태를 바꿀 수 있습니다. 상태를 먼저 조회한 뒤 바꾸는 방식만 쓰면, 두 요청이 같은 대기 상태를 보고 각각 처리를 진행할 수 있습니다.

그래서 PG 승인 결과를 반영하거나 만료 처리를 할 때 대기 상태인 결제 건만 바꿀 수 있도록 조건부 UPDATE를 적용했습니다. 상태 변경에 성공한 요청만 후속 처리를 진행하고, 이미 상태가 바뀐 결제 건은 다시 반영하지 않습니다.

분할결제에서는 여러 회차의 승인 처리가 겹칠 수 있어 주문 완료 상태를 바꾸는 요청도 같은 방식으로 한 번만 성공하게 했습니다. 이를 통해 수강권과 적립금이 중복 지급되지 않도록 했습니다.



같은 결제 건에 여러 요청이 들어올 때의 처리 흐름과 실제 코드 일부

프레스런 — 중복 요청에도 결제 결과가 한 번만 반영되도록 처리 (계속)

상태 확인

결제 API를 호출했다고 해서 돌아온 성공·실패 응답만으로 최종 상태를 판단하지 않았습니다. 사용자가 승인 요청을 중복으로 보내면 Toss Payments는 이미 처리된 결제라는 오류를 돌려줄 수 있는데, 이 오류를 그대로 결제 실패로 저장하면 실제로는 결제가 완료됐지만 서비스에서는 실패로 기록되는 상황이 생깁니다.

따라서 승인 API에서 오류가 나면 PG에 결제 상태를 다시 조회해 실제 승인 여부를 확인했습니다. PG에서는 승인됐지만 내부 결제 건이 이미 종료된 경우에는 결제키를 비교했습니다.

- 결제키가 같으면 이미 처리된 요청의 중복 호출로 판단
- 결제키가 다르면 받을 자리가 없는 결제이므로 해당 PG 결제를 취소

가상계좌 입금 통보도 마찬가지로, 통보 요청의 secret을 검증한 뒤 전달받은 본문만으로 입금 완료를 확정하지 않고 결제사에 실제 상태를 다시 조회했습니다. 승인 금액과 결제 수단이 요청한 내용과 일치하는지도 확인했습니다.

결과

결제 시도와 상태 변경이 겹치는 상황에서도 결제 결과가 한 번만 반영되는 구조를 만들었습니다.

- 결제 시도 이력 분리와 DB Unique 제약으로 중복 데이터 생성 방지
- 승인·입금 통보·만료 처리의 동시 실행을 고려한 상태 변경 로직 구현
- PG 상태 재조회로 중복 승인 응답과 내부 결제 상태 불일치에 대응
- 주문 완료 상태를 기준으로 수강권·적립금 중복 지급 방지

퍼플아카데미 — 여러 PG 상점과 결제 방식에 대응하는 구조

문제 상황

퍼플아카데미에서는 과세·면세 여부, 상품 종류, 일반결제·BrandPay 여부에 따라 서로 다른 Toss 상점 인증 정보를 써야 했습니다. 2개의 Toss 상점에서 6개의 키를 관리하는 구조였습니다.

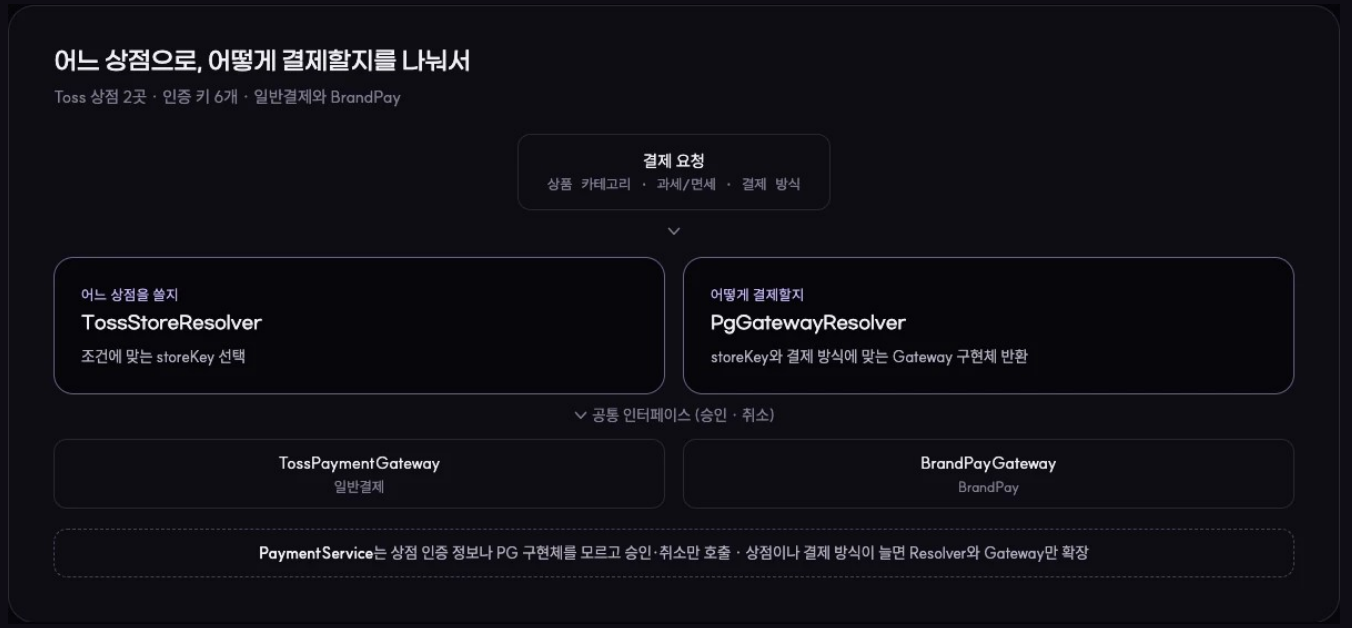
결제 서비스 안에서 상품 조건마다 인증 정보를 고르고 PG API를 직접 호출하면, 결제 방식이 추가되거나 상점 정보가 바뀔 때 고쳐야 할 코드가 많아집니다. 그래서 어떤 상점을 쓸지 결정하는 로직과 실제 결제를 실행하는 로직을 분리했습니다.

구조 분리

두 가지 Resolver와 Gateway 구조를 적용했습니다.

TossStoreResolver: 과세·면세 여부, 상품 카테고리, 결제 방식에 따라 사용할 storeKey를 선택

PgGatewayResolver: 선택된 상점과 결제 방식에 맞춰 TossPaymentGateway 또는 BrandPayGateway 구현체를 반환



TossStoreResolver와 PgGatewayResolver 구조

퍼플아카데미 — 여러 PG 상점과 결제 방식에 대응하는 구조 (계속)

구조 분리 (계속)

각 Gateway는 공통 인터페이스로 승인과 취소 기능을 제공합니다. 그 결과 PaymentService는 Toss 상점의 인증 정보나 결제 방식별 API 호출 구현을 직접 다루지 않아도 됐고, 새 상점이나 결제 방식이 추가되더라도 고칠 곳을 Resolver와 Gateway로 한정할 수 있었습니다.

결과

상점 선택과 PG 호출의 책임을 분리하고, 결제·입학 상태를 연동했습니다.

- 2개 Toss 상점과 6개 인증 키를 결제 조건에 따라 선택하는 구조 구현
- Resolver와 Gateway의 책임 분리를 통해 상점 선택 정책과 PG 호출 로직 간 결합도 감소
- 관심사 분리와 컴포넌트별 책임 정의를 통해 결제 시스템의 확장성 및 유지보수성 강화

GuideME — 예약 충돌을 방지하고 정산까지 이어지는 결제 처리

문제 상황

GuideME는 여행자가 현지 가이드를 예약하고 결제하는 서비스입니다. 일반적인 상품 구매와 달리 결제가 끝난 뒤에도 예약 확정, 서비스 진행, 서비스 완료, 정산까지 거래가 이어집니다. 그래서 결제 완료 여부뿐 아니라 예약 가능 상태와 정산 조건까지 함께 관리해야 했습니다.

특히 테스트 과정에서 같은 가이드의 동일한 시간대에 두 건의 예약이 동시에 결제 완료되는 문제가 있었습니다. 예약 가능 여부를 애플리케이션에서 확인하더라도, 두 요청이 거의 동시에 처리되면 각각 예약 가능한 상태라고 판단할 수 있었습니다.

중복 방지

주문과 결제 내역에는 각각 고유 식별자를 사용했습니다. 이미 처리한 주문이나 결제인지 애플리케이션에서 먼저 확인하고, DB에도 Unique 제약조건을 걸어 재시도나 중복 요청으로 같은 주문·결제 데이터가 반복 생성되지 않도록 했습니다.

GuideME — 예약 충돌을 방지하고 정산까지 이어지는 결제 처리 (계속)

예약 충돌

동일 시간대의 중복 예약을 방지하기 위해 애플리케이션의 예약 가능 여부 검증과 함께 PostgreSQL의 tstzrange 및 GiST 인덱스 기반 Exclusion Constraint를 적용했습니다.

예약 시작·종료 시간을 범위로 관리하고, 동일 가이드에게 겹치는 시간대의 예약이 생성되지 않도록 DB 수준에서 제약조건을 설정했습니다. 이를 통해 동시 예약 요청이 발생하더라도 데이터베이스에서 중복 예약을 차단해 예약 정합성을 확보했습니다.



같은 시간대 예약 충돌과 Exclusion Constraint (설명용 가상 시간대)

GuideME — 예약 충돌을 방지하고 정산까지 이어지는 결제 처리 (계속)

정산 관리

GuideME의 거래는 결제 완료만으로 끝나지 않기 때문에 예약과 서비스 진행 상태를 상태 머신으로 구분해 관리했습니다. 정산은 결제 직후 처리하지 않고, 매일 새벽 실행되는 배치 작업에서 정산 가능한 거래를 확인합니다. 정산 대상은 다음 조건을 모두 만족하는 거래로 제한했습니다.

- 결제가 정상적으로 완료된 거래
- 가이드 서비스가 정상적으로 완료된 거래
- 환불이 발생하지 않은 거래

조건을 만족하는 거래만 트랜잭션 단위로 정산하도록 구현했습니다.

결과

결제 이후 예약 충돌과 정산 조건까지 이어지는 거래 상태 관리 구조를 만들었습니다.

- 주문 ID와 결제 ID 기반 중복 처리 방지
- PostgreSQL Exclusion Constraint로 동일 가이드·시간대 예약 충돌 방지
- 결제·예약·서비스 완료·환불 상태를 고려한 정산 처리

02

메시지 큐·장애 격리

메시지 큐 기반 비동기 처리 시스템 구축

RabbitMQ 기반으로 예약 후속 작업과 알림 처리를 비동기화하고, AWS SQS를 활용해 알림 발송, 결제 취소·환불, 결제수단 변경 취소, 대용량 엑셀 추출을 별도 워커로 분리. 작업 특성에 따라 비동기 처리 구조를 통한 서버의 부담 저하 및 운영 안정성 확보

사례	메시지 큐 처리
기간	2024.12 — 2026.10
역할	로워드 · GuideME · 백엔드
기술	RabbitMQ, AWS SQS, DLQ, NestJS, Circuit Breaker

다루는 프로젝트

프레스런
AWS SQS 기반 비동기 처리 시스템 구축

GuideME
예약 처리와 외부 API 연동 분리

프레스런 — AWS SQS 기반 비동기 처리 시스템 구축

프레스런에서는 알림 발송, 결제 후속 처리, 대용량 엑셀 추출처럼 요청 안에서 끝낼 필요가 없거나 오래 걸리는 작업을 AWS SQS로 API 서버와 분리했습니다. 큐와 워커를 업무 단위로 나누고, 여러 워커가 함께 쓰는 공통 컨슈머 모듈을 SQS의 전달 방식에 맞춰 설계했습니다.

적용 사례

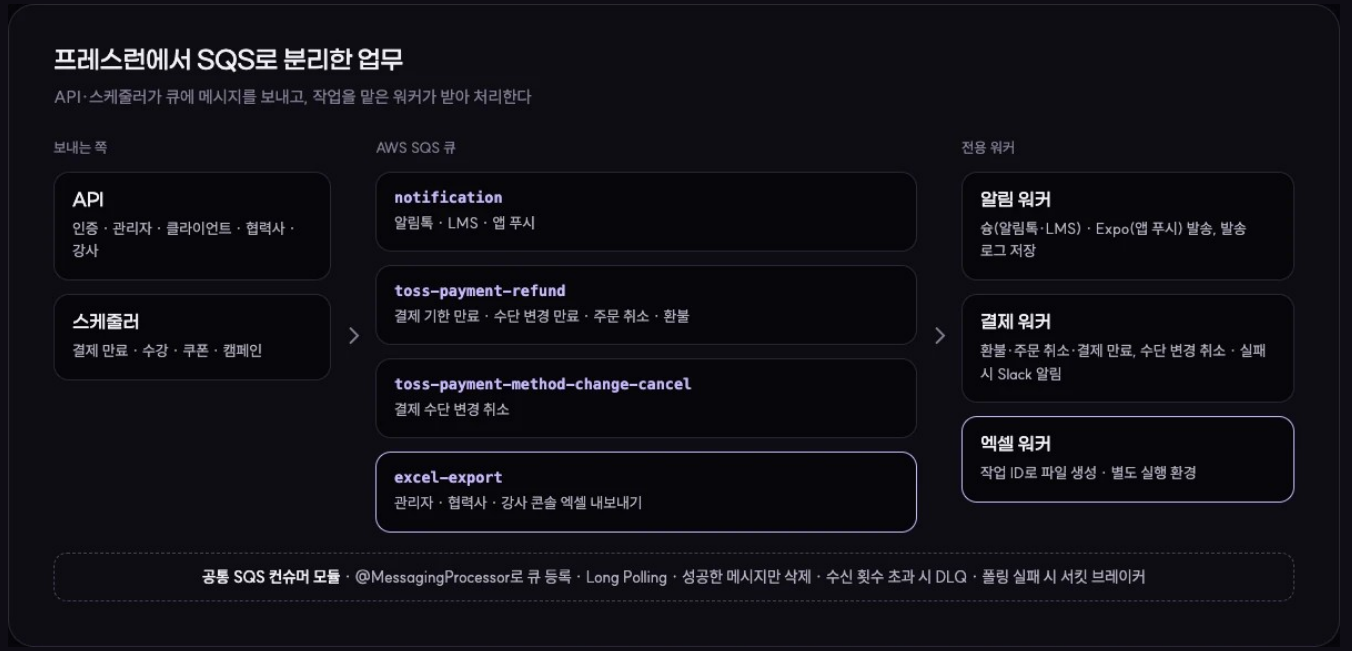
다음 업무에 SQS를 적용했습니다.

알림 (notification 큐): 알림톡·LMS·앱 푸시 발송과 발송 로그 저장

- 결제 (toss-payment-refund 큐): 결제 기한 만료, 결제 수단 변경 만료, 주문 취소, 환불 요청
- 결제 수단 변경 (toss-payment-method-change-cancel 큐): 결제 수단 변경 취소

엑셀 추출 (excel-export 큐): 관리자·협력사·강사 콘솔의 엑셀 내보내기

각 업무는 API나 스케줄러가 SQS로 메시지를 보내고, 그 작업을 맡은 워커가 메시지를 받아 처리합니다.



프레스런의 SQS 적용 구조: 큐 4개와 전용 워커 3개

프레스런 — AWS SQS 기반 비동기 처리 시스템 구축 (계속)

적용 사례 (계속)

알림 워커: 인증·관리자·클라이언트 API와 스케줄러에서 발생한 알림을 슝(알림톡·LMS)과 Expo(앱 푸시)로 발송. 외부 발송 API의 지연·실패가 원래 요청의 응답을 붙잡지 않도록 분리

결제 워커: 환불 요청, 주문 취소, 결제 만료 등 결제 후속 작업 처리. 결제사 취소 API가 실패해도 요청이 사라지지 않고 큐에서 재시도되도록 분리

엑셀 워커: 관리자·협력사·강사 API가 보낸 작업 ID를 기준으로 엑셀 파일 생성. 요청 안에서 파일을 만들다 관리자 API 서버가 OOM으로 종료되던 문제를 별도 워커로 분리해 해결

공통 모듈

워커마다 메시지 수신과 오류 처리를 반복 구현하지 않도록 공통 SQS 컨슈머 모듈을 만들었습니다. @MessagingProcessor 데코레이터로 처리할 큐와 메시지 처리기를 등록해, 워커별 비즈니스 로직과 공통 메시지 처리 로직을 분리했습니다.

Long Polling: 빈 응답을 줄이도록 대기하며 메시지 조회

처리 완료 후 삭제: 작업이 성공한 경우에만 메시지를 삭제하고, 실패하면 남겨 두어 다시 수신

DLQ: 최대 수신 횟수를 넘은 메시지는 Dead Letter Queue로 옮겨 정상 흐름과 격리

Circuit Breaker: SQS 폴링이 반복해서 실패하면 일정 시간 폴링 중단

중복 전달 대비: 같은 메시지가 다시 와도 한 번만 처리되도록 워커에서 작업 상태를 확인한 뒤 실행

공통 모듈은 메시지 수신과 실패 처리 흐름을 맡고, 실제 작업 실행과 재시도 여부 판단은 각 워커가 하도록 나눴습니다.

결과

알림·결제·엑셀 세 가지 업무를 큐 4개와 전용 워커 3개로 분리했습니다.

알림: 외부 발송 API의 지연·실패와 상관없이 API 요청에 바로 응답

결제: 결제사 취소가 실패해도 요청이 유실되지 않고 큐에서 재시도, 끝내 실패하면 DLQ로 확인

엑셀: 관리자 엑셀 23종과 협력사·강사 콘솔 다운로드를 워커로 전환해 관리자 API 서버의 OOM 문제 해결

공통: 새 비동기 작업은 큐와 처리기를 등록하는 것만으로 같은 수신·실패 처리 흐름을 사용

GuideME — 예약 처리와 외부 API 연동 분리

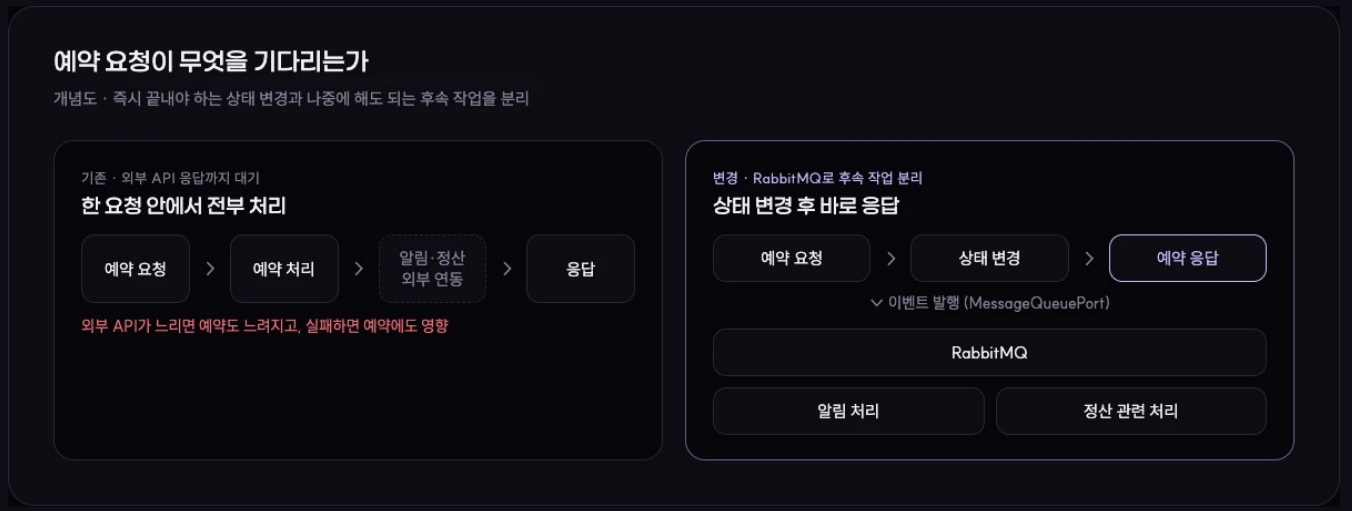
문제 상황

GuideME는 사용자가 현지 가이드를 예약하고 결제하는 서비스입니다. 기존에는 예약 요청 시 상태 변경과 알림 발송 등 여러 후속 비즈니스 로직을 함께 처리해, 작업량이 증가하면 예약 API의 응답 지연과 병목이 발생할 수 있는 구조였습니다.

작업 분리

이를 개선하기 위해 예약에 필요한 핵심 로직과 후속 작업을 분리하고, RabbitMQ 기반 비동기 처리 구조를 도입했습니다. 서비스 운영 초기의 인프라 비용을 고려해 별도 워커 서버를 구축하지 않고, 기존 운영 환경에서 후속 작업을 처리하도록 구성했습니다.

또한 메시지 큐 구현에도 변경 가능성을 고려했습니다. 도메인 로직에서 RabbitMQ를 직접 쓰지 않고 MessageQueuePort 인터페이스로 메시지를 발행하게 하고, RabbitMQ 관련 코드는 별도 구현체로 분리해 예약 로직이 특정 메시지 브로커에 직접 의존하지 않도록 했습니다.



RabbitMQ 도입 전후 예약 처리 흐름 (개념도)

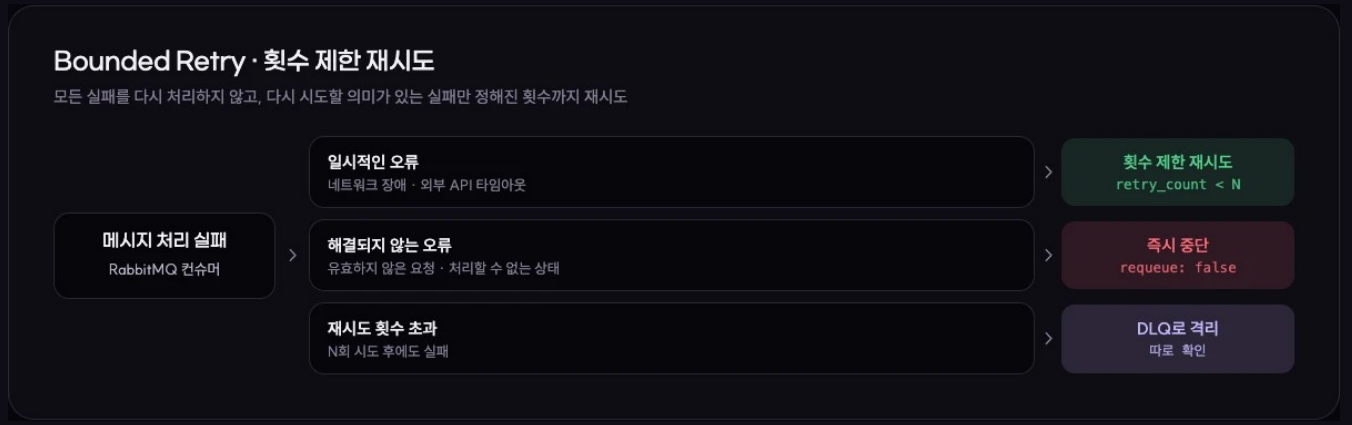
GuideME — 예약 처리와 외부 API 연동 분리 (계속)

재시도 정책

실패한 메시지를 일률적으로 재처리하면서 유효하지 않은 요청까지 반복 실행되고, 컨슈머 객체가 쌓여 힙 메모리가 급증했습니다.
그래서 횟수 제한 재시도(Bounded Retry) 패턴을 적용하고, 오류 원인에 따라 처리를 나눴습니다.

- 일시적인 오류(네트워크 장애·외부 API 타임아웃): 메시지 헤더의 retry_count로 정해진 횟수까지만 재시도
- 해결되지 않는 오류(유효하지 않은 요청·처리할 수 없는 상태): requeue: false로 즉시 중단

재시도 횟수 초과: DLQ로 격리해 정상 메시지 처리와 분리하고 따로 확인



Bounded Retry: 실패 원인에 따른 재시도·중단·DLQ 격리

GuideME — 예약 처리와 외부 API 연동 분리 (계속)

결과

예약 요청과 알림·정산 처리를 분리했습니다.

예약: 외부 API의 지연·장애와 상관없이 상태 변경 후 바로 응답

재시도: Bounded Retry로 무의미한 반복 처리를 막아 컨슈머 힙 메모리 급증 문제 개선

실패 메시지: 재시도 횟수를 넘기면 DLQ로 격리해 정상 처리 흐름과 분리

회고

GuideME와 프레스런에서 사용한 메시지 브로커는 달랐지만, 해결하려던 문제에는 공통점이 있었습니다. GuideME에서는 외부 API의 응답을 기다리지 않도록 예약 처리와 후속 작업을 분리했고, 프레스런에서는 리소스 사용량이 큰 작업을 별도 워커로 옮겼습니다.

두 프로젝트를 거치며 비동기 처리는 요청을 빨리 끝내는 방법에 그치지 않고, 실패한 작업을 어떻게 다시 처리하고 정상적인 요청에 영향을 주지 않게 할지까지 고려해야 하는 구조라는 점을 배웠습니다.

03

캐시 설계·무효화

서비스 · 데이터 특성에 맞는 Redis 캐시 설계와 무효화 전략

사용자별 위치 검색과 관리자 데이터 변경이라는 서로 다른 상황에 맞춰 캐시 무효화 전략을 설계. GuideME에서는 Version Key로 사용자별 캐시 관리 문제를 해결하고, 로워드에서는 공통 캐시 정의와 데이터 변경 시점의 캐시 삭제 구조를 적용.

사례	캐시 전략
기간	2024.12 — 2026.10
역할	GuideME · 로워드 · 백엔드
기술	Redis AWS ElastiCache(Valkey) Cache-Aside Version Key TTL

다루는 프로젝트

프레스런
서비스·데이터 특성에 맞는 캐시 설계와 무효화 전략

GuideME
사용자별 위치 검색 캐시를 효율적으로 갱신하기

프레스런 — 서비스·데이터 특성에 맞는 캐시 설계와 무효화 전략

문제 상황

프레스런은 사용자 API와 관리자 API가 분리되어 있고, 관리자가 바꾼 상품 정보가 사용자 화면에도 반영되어야 합니다. 캐시를 적용하면서 다음 상황을 고려해야 했습니다.

데이터 변경: 관리자가 상품 정보나 커리큘럼을 수정해도 기존 캐시가 남아 있으면 사용자에게 이전 정보가 표시될 수 있음

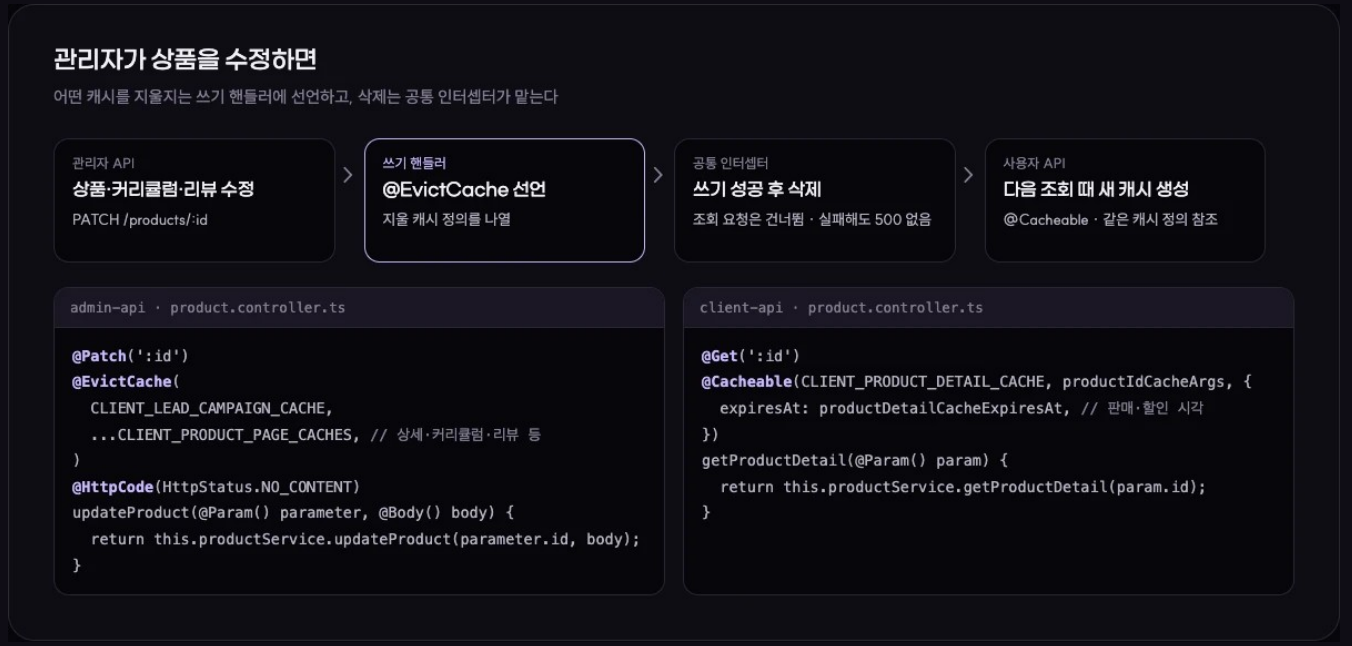
사용자별 응답: 같은 조회라도 회원이나 권한에 따라 응답이 다른데, 같은 캐시를 공유하면 다른 사람의 정보가 노출될 수 있음

시간에 따른 상태 변경: 판매·할인 시작이나 종료 시각이 지나도 캐시가 유지되면 가격과 구매 가능 상태가 실제와 다르게 표시될 수 있음

그래서 데이터 변경, 사용자 구분, 판매·할인 기간을 함께 고려한 캐시 관리 구조를 구성했습니다.

무효화

사용자 API와 관리자 API가 캐시 키와 TTL을 각자 관리하면, 관리자가 데이터를 수정해도 사용자 API의 캐시가 갱신되지 않을 수 있습니다. 그래서 캐시 키와 TTL을 공통 정의로 통합하고, 두 API가 같은 정의를 참조하도록 했습니다.



관리자 데이터 변경부터 사용자 캐시 갱신까지의 흐름과 실제 코드 일부

프레스런 — 서비스·데이터 특성에 맞는 캐시 설계와 무효화 전략 (계속)

무효화 (계속)

@Cacheable: 조회 API에 캐싱 적용

@EvictCache: 관리자 데이터 변경 핸들러에 지울 캐시 선언

공통 인터셉터: 데이터 변경이 성공한 뒤에만 선언된 캐시 삭제

예를 들어 관리자가 상품을 수정하면 상품 상세·커리큘럼·리뷰 캐시를 지우고, 이후 사용자 요청에서 바뀐 데이터를 다시 조회해 캐싱합니다.

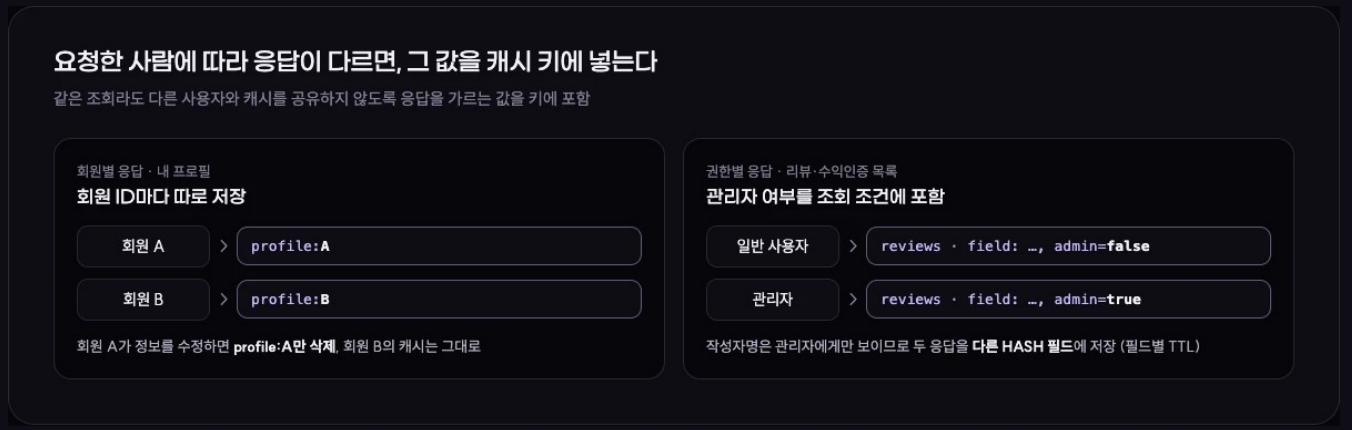
사용자 구분

요청한 사람에 따라 응답이 달라지는 캐시는 응답을 가르는 값을 빠짐없이 캐시 키에 넣어, 다른 사용자와 공유되지 않도록 했습니다.

회원별 응답: 내 프로필은 회원 ID별로 키를 나누고, 본인 정보를 수정하면 그 회원의 키만 삭제

권한별 응답: 리뷰·수익인증 목록은 관리자에게만 작성자명을 보여주므로 관리자 여부를 조회 조건에 포함

조회 조건별 저장: 조건마다 다른 결과는 Valkey HASH의 필드로 저장하고 HSETEX로 필드마다 TTL 설정



회원·권한에 따라 달라지는 응답을 캐시 키로 구분

프레스런 — 서비스·데이터 특성에 맞는 캐시 설계와 무효화 전략 (계속)

TTL 구분

모든 캐시에 고정 TTL을 쓰면 실제 데이터의 상태가 바뀌는 시점과 캐시가 끝나는 시점이 어긋납니다.

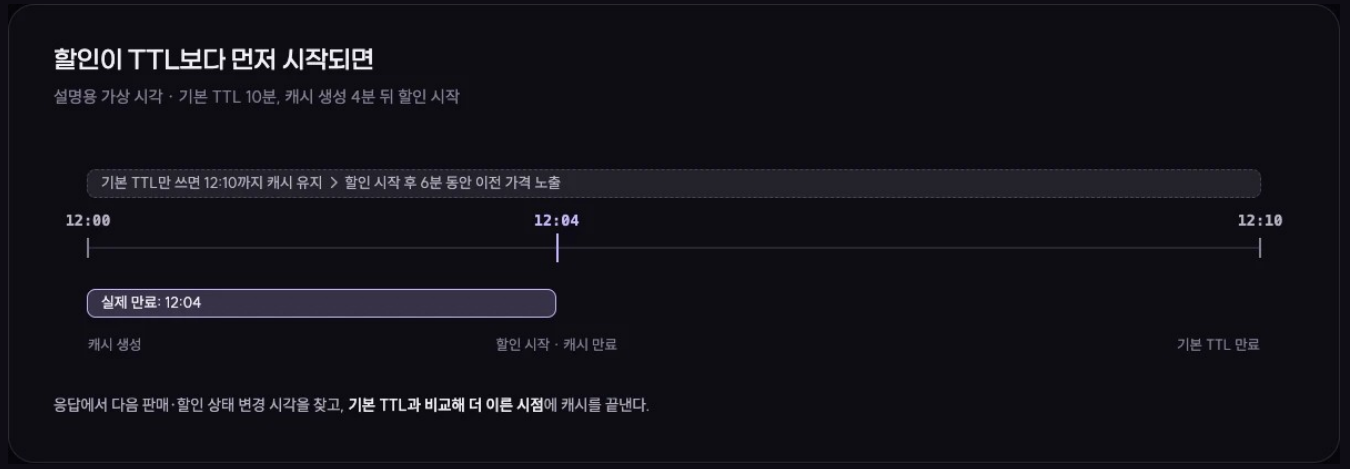
예를 들어 할인이 12시 4분에 시작되는데 캐시가 12시 10분까지 남으면, 할인 시작 뒤에도 이전 가격이 표시됩니다. 그래서 데이터 특성에 따라 TTL을 다르게 적용했습니다.

상품 목록: 여러 관리자 기능의 데이터를 함께 써서 최대 60초의 반영 지연을 허용

상품 상세·커리큘럼·리뷰: 기본 10분 TTL에 데이터 변경 시 무효화

판매·할인 기간: 응답에서 다음 상태 변경 시각을 찾아 기본 TTL과 비교해 더 이른 시점에 만료

이렇게 해서 관리자가 수정하지 않아도 판매·할인 시작과 종료 시각에 맞춰 캐시가 갱신됩니다.



판매·할인 시각에 맞춘 캐시 만료 예시 (설명용 가상 시각)

프레스런 — 서비스·데이터 특성에 맞는 캐시 설계와 무효화 전략 (계속)

장애 대응

캐시 서버 장애가 API 장애로 이어지지 않도록, 캐시 조회·저장에 실패하면 원본 DB에서 조회해 응답합니다. 캐시 삭제에 실패하면 이미 처리된 변경 요청을 500으로 돌려주지 않고 로그를 남기며, TTL로 오래된 데이터가 무기한 남지 않게 했습니다

결과

데이터 변경, 사용자 구분, 시간에 따른 상태 변경을 각각 다른 방식으로 처리하는 캐시 구조를 만들었습니다.

- 사용자·관리자 API의 캐시 정의를 통합해 관리 방식 일원화
- 관리자 데이터 변경 시 관련 캐시를 지워 이전 정보 노출 방지
- 회원·권한별 응답을 캐시 키로 구분해 다른 사용자의 정보 노출 방지
- 판매·할인 일정에 맞춘 TTL로 시간에 따른 상태 변경 반영
- 캐시 서버 장애 시에도 원본 데이터 조회로 API 요청 처리 유지

GuideME — 사용자별 위치 검색 캐시를 효율적으로 갱신하기

문제 상황

GuideME에는 사용자 주변에서 활동할 수 있는 가이드를 검색하는 기능이 있습니다. 단순히 가까운 가이드를 찾는 것이 아니라, PostGIS를 이용한 거리 계산과 함께 가이드 승인 상태, 서비스 가능 여부, 사용자 간 차단 관계까지 확인해야 했습니다.

여러 조건을 함께 조회하는 만큼 DB 조회 비용이 높아 Redis 캐시를 적용했습니다. 하지만 위치 검색은 사용자와 검색 조건에 따라 결과가 달라지기 때문에, 캐시를 저장하는 것보다 데이터가 바뀌었을 때 어떻게 갱신할지가 더 큰 문제였습니다.

캐시 키

사용자마다 검색 위치와 조건이 다르므로 캐시 키에도 사용자 정보와 검색 조건을 포함했습니다. 예를 들면 nearby-guide:v1:{userId}:{검색조건} 형태입니다.

문제는 가이드의 위치나 서비스 가능 상태가 바뀌는 경우였습니다. 가이드 한 명의 정보가 바뀌어도 그 가이드가 포함된 여러 사용자의 검색 결과가 영향을 받습니다. 기존 캐시를 지우려면 어떤 사용자의 검색 결과에 그 가이드가 들어 있는지 알아야 했고, 캐시가 늘어날수록 지워야 할 키를 찾아 관리하는 일도 복잡해지는 구조였습니다.

GuideME — 사용자별 위치 검색 캐시를 효율적으로 갱신하기 (계속)

해결 방식

캐시를 갱신하는 방법으로 세 가지를 검토했습니다.

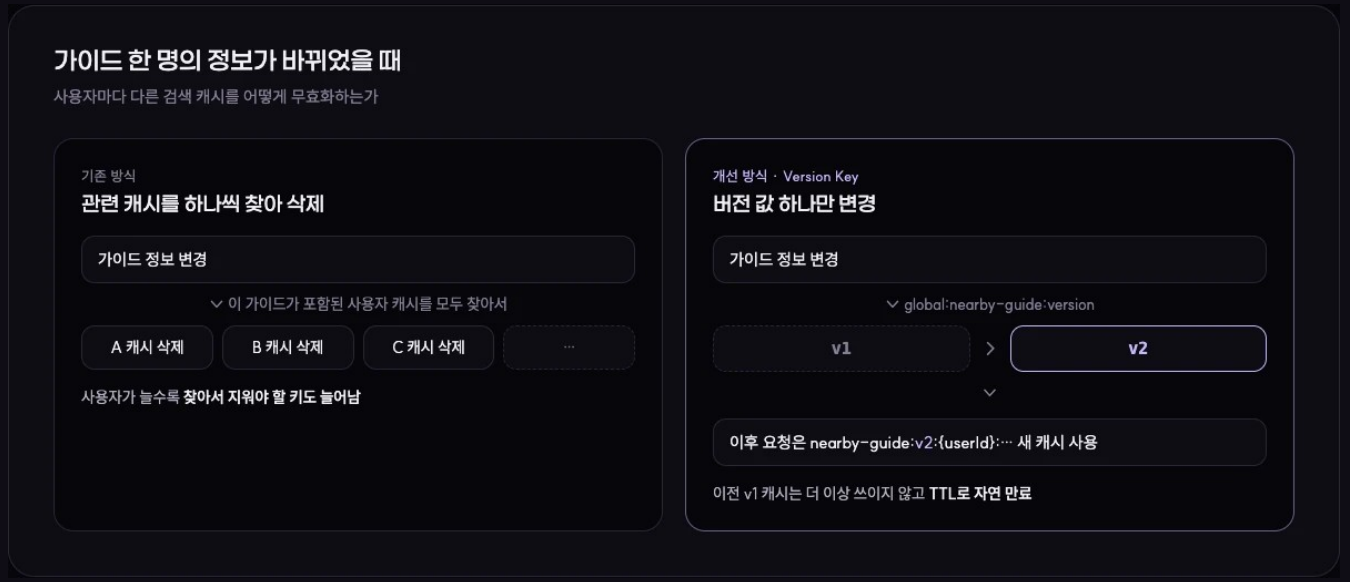
개별 캐시 삭제: 변경된 가이드와 관련된 사용자별 캐시를 찾아 지워야 하므로 관리 대상이 많아짐

TTL 만료: 구현은 단순하지만 캐시가 만료되기 전까지 이전 검색 결과가 노출될 수 있음

Version Key: 검색 기능 전체가 공유하는 버전을 바꿔 기존 캐시를 더 이상 쓰지 않도록 처리

이 중 Version Key 방식을 선택했습니다. Redis에 위치 검색용 버전 값(global:nearby-guide:version)을 두고 검색 캐시 키에 그 버전을 넣었습니다. 가이드 정보가 바뀌어 기존 검색 결과를 갱신해야 할 때는 버전 값을 올립니다.

예를 들어 버전이 v1에서 v2로 바뀌면 이후 검색 요청은 v2가 들어간 새 캐시 키를 사용하고, 기존 v1 캐시는 더 이상 쓰이지 않다가 TTL에 따라 자연스럽게 삭제됩니다. 이렇게 사용자별 캐시 키를 하나씩 찾아 지우지 않고도 위치 검색 캐시 전체를 무효화할 수 있었습니다.



Version Key 적용 전후의 캐시 무효화 흐름

GuideME — 사용자별 위치 검색 캐시를 효율적으로 갱신하기 (계속)

결과

Redis 캐싱을 적용해 위치 검색 API 응답 속도를 약 40% 개선했습니다.

- 사용자별 캐시 키를 추적하거나 하나씩 지우지 않고 버전 값 변경으로 무효화
- 기존 캐시는 TTL로 만료되도록 해 별도의 전체 삭제 작업 없이 관리

회고

두 프로젝트는 서비스에서 중요하게 다루는 데이터의 특성과 갱신주기에 따라 서로 다른 캐싱 전략을 적용했습니다.

GuideME는 가이드 목록 조회가 중요하고 검색 조건에 따라 다양한 캐시가 생성되므로, Version Key를 활용해 목록 캐시를 일괄 무효화했습니다.

반면 프레스런은 상품 상세 조회가 중요하고 상품의 추가·삭제가 드물어, 상품별·사용자별로 캐시를 구분하고 데이터 변경 시 필요한 캐시만 무효화하도록 구성했습니다.

이를 통해 캐싱 전략은 단순히 조회 성능을 높이는 것뿐 아니라, 서비스의 핵심 조회 패턴과 데이터 변경 특성을 고려해 선택해야 한다는 점을 배웠습니다.

04

AI 활용 개발·FDE

Harness Engineering 기반 AI 개발 환경 구축

Harness Engineering을 통해 Claude Code·Codex의 작업 규칙과 검증 절차를 표준화. AI 에이전트의 구현 결과를 테스트와 실제 동작으로 검증하며, 프레스런 TF에서 착수 한 달 만에 신규 플랫폼 오픈.

사례	Harness Engineering
기간	2025.12 — 2026.11
역할	프레스런 TF 백엔드 · 개인 프로젝트 FE·BE·인프라
기술	AI, Harness Engineering, Context Engineering, Agent Skills, Guardrails

다루는 프로젝트

프레스런
Harness Engineering 기반 AI 개발 프로세스 구축

개인 프로젝트
Harness 기반 Agentic Development Workflow

프레스런 — Harness Engineering 기반 AI 개발 프로세스 구축

문제 상황

프레스런은 착수 한 달 안에 신규 교육 플랫폼을 오픈해야 하는 프로젝트였습니다. 화면 기획과 세부 정책이 확정되지 않은 부분이 있어 요구사항을 구체화하면서 개발해야 했습니다.

AI가 기획을 임의로 해석하거나 기존 코드 구조를 고려하지 않고 수정하면 실제 서비스 정책과 다른 결과가 나올 수 있었습니다. 그래서 AI에 구현을 맡기되, 정책 판단과 작업 범위, 검증 기준은 사람이 통제하는 개발 환경을 구성했습니다.

규약 정의

Context Engineering. Claude Code와 Codex가 같은 기준으로 작업하도록 CLAUDE.md와 AGENTS.md에 프로젝트 구조와 코딩 규약을 정의했습니다. AI 도구가 달라져도 같은 개발 기준을 유지합니다.

- 미확정 기획의 임의 해석 및 구현 금지
- 요청 범위 밖의 코드 수정과 리팩터링 제한
- DB 스키마 변경 시 마이그레이션 적용
- 명시적인 승인 없는 Git 커밋 금지

presslearn-backend / .claude/rules/00-invariants.md — 항상 적용 · 다른 규칙과 충돌하면 이 파일이 우선

```
# 00 · 불변 규칙

어떤 작업에도 항상 적용된다. 다른 규칙 파일과 충돌하면 이 파일이 이긴다.

## 사람과의 경계

- 커밋은 사용자가 명시적으로 지시할 때만 한다. 구현·테스트·검증이 전부 끝나도 마찬가지다.
- 기획 의도를 추측하지 않는다. 확인이 필요하면 기획을 읽거나 사용자에게 묻는다.
- 결함이나 위험을 보고하기 전에 그 경로가 실제 애플리케이션 코드에서 도달 가능한지 먼저 확인한다.

## 스키마

모든 환경에서 synchronize: false. 스키마 변경은 반드시 마이그레이션으로만 반영한다.

작업별 규칙 — 해당 작업을 시작하기 전에 읽는다 (CLAUDE.md)
10-typescript · 20-nestjs-api · 30-typeorm · 40-migrations · 50-testing · 60-documentation · 70-refactoring
```

프레스런 백엔드 저장소의 실제 규약 파일(00-invariants.md) 일부

프레스런 — Harness Engineering 기반 AI 개발 프로세스 구축 (계속)

작업 절차

Agent Skills. 작업 유형에 따라 필요한 절차를 수행하도록 커스텀 스킬을 구성해, AI가 바로 코드를 쓰는 대신 기존 구조와 요구사항을 먼저 확인하게 했습니다.

domain-feature: 기획·DB 구조 확인 → 정책 확정 → TDD → 구현 → 검증

code-change: 기존 코드와 영향 범위 분석 → 수정 → 검증

이외에도 성능 분석, 기획 문서 대조, 독립 작업 환경 구성 같은 절차를 스킬로 분리했습니다.



기능 하나를 만드는 순서와 단계별 확인 지점

프레스런 — Harness Engineering 기반 AI 개발 프로세스 구축 (계속)

판단 범위

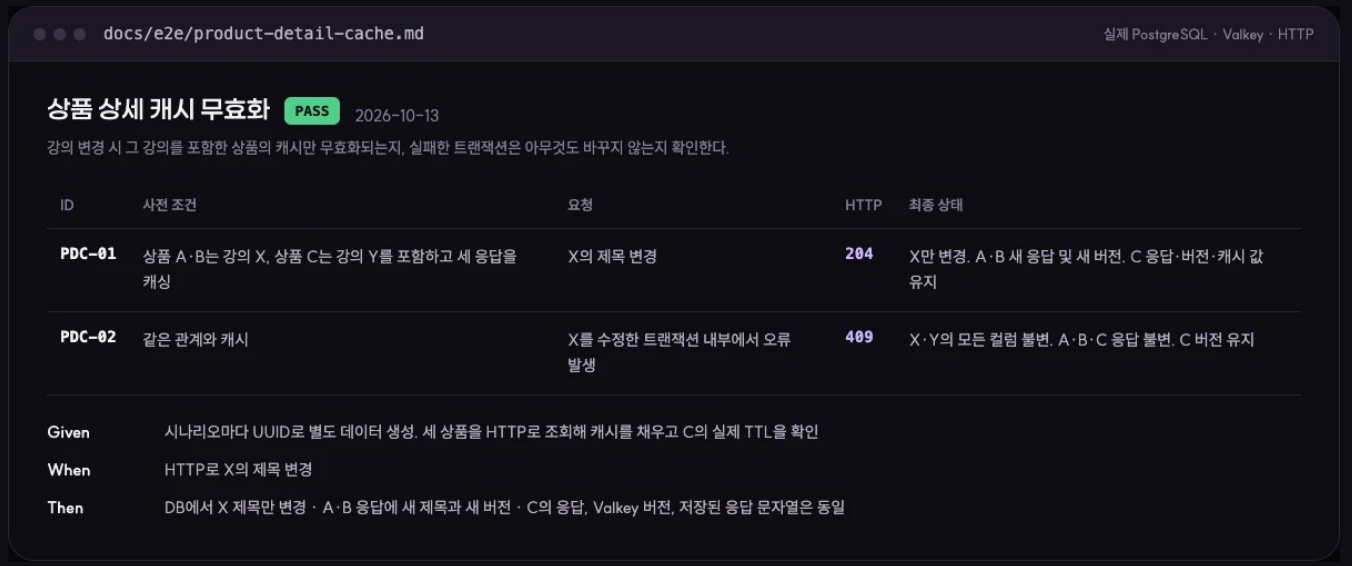
Execution Guardrails. 가장 중요하게 관리한 부분은 AI가 구현할 수 있는 영역과 사람이 결정해야 하는 영역을 구분하는 것이었습니다. 게시판 접근 권한, 강사 노출 조건, 비회원 작성 가능 여부처럼 기획이 불명확한 부분은 AI가 임의로 결정하지 않고 확인할 사항을 정리하도록 했습니다.

확정된 정책은 별도 판정 함수와 테스트로 구현해, 기획이 바뀌어도 그 로직을 중심으로 수정할 수 있게 했습니다. 엔티티·DTO 같은 반복 구현은 AI에 맡기되, 트랜잭션 경계, 수강권 연장 동시 요청 같은 동시성 처리, 캐시 무효화 시점, 외부 API 오류 처리 등 핵심 비즈니스 로직은 직접 검토했습니다.

품질 검증

TDD & Quality Gates. AI의 작업 완료 보고만으로 구현이 끝났다고 판단하지 않고 검증 절차를 정의했습니다.

요구사항 확인 → 실패 테스트 작성 → 구현 → Test·Build·Lint → 리팩터링 → 실제 동작 확인



docs/e2e/product-detail-cache.md

실제 PostgreSQL · Valkey · HTTP

상품 상세 캐시 무효화 PASS 2026-10-13

강의 변경 시 그 강의를 포함한 상품의 캐시만 무효화되는지, 실패한 트랜잭션은 아무것도 바꾸지 않는지 확인한다.

ID	사전 조건	요청	HTTP	최종 상태
PDC-01	상품 A·B는 강의 X, 상품 C는 강의 Y를 포함하고 세 응답을 캐싱	X의 제목 변경	204	X만 변경. A·B 새 응답 및 새 버전. C 응답·버전·캐시 값 유지
PDC-02	같은 관계와 캐시	X를 수정한 트랜잭션 내부에서 오류 발생	409	X·Y의 모든 컬럼 불변. A·B·C 응답 불변. C 버전 유지

Given 시나리오마다 UUID로 별도 데이터 생성. 세 상품을 HTTP로 조회해 캐시를 채우고 C의 실제 TTL을 확인

When HTTP로 X의 제목 변경

Then DB에서 X 제목만 변경 · A·B 응답에 새 제목과 새 버전 · C의 응답, Valkey 버전, 저장된 응답 문자열은 동일

실제 PostgreSQL·Valkey·HTTP로 확인한 E2E 검증 기록

프레스런 — Harness Engineering 기반 AI 개발 프로세스 구축 (계속)

품질 검증 (계속)

DB 관련 기능은 SQL로 저장 결과를 확인하고, 여러 기능이 연결된 흐름은 PostgreSQL·Valkey 환경에서 HTTP 요청 전후의 데이터를 비교했습니다. 문서 규약과 별개로, AI 도구의 Hooks로 지시 없는 커밋 같은 일부 규약 위반을 감지해 경고하도록 했습니다.

결과

규약과 작업별 스킬로 AI 개발 절차를 체계화했습니다.

- 프로젝트 규약과 작업별 Skills를 활용한 AI 개발 절차 체계화
- 불명확한 요구사항을 정책과 테스트로 구체화해 실제 서비스에 반영
- 프레스런 TF에서 착수 한 달 만에 신규 플랫폼 오픈

AI를 활용한 개발에서 중요한 것은 코드 생성 속도뿐 아니라, 무엇을 구현할지 판단하고 결과를 검증할 수 있는 기준을 마련하는 것이었습니다.

개인 프로젝트 — Harness 기반 Agentic Development Workflow

개인 프로젝트에서는 프레스런에서 정리한 Harness Engineering 방식을 활용해, 기획부터 프론트엔드·백엔드·인프라 구현, 검증과 개선까지 AI 에이전트와 역할을 나누어 진행했습니다. 프로젝트 방향과 기술적 의사결정은 직접 맡고, 세부 설계와 반복적인 구현에 AI를 활용하는 방식입니다.

청사진

Blueprint. 프로젝트의 목적과 핵심 기능, 구현 범위는 직접 정의했습니다. Slack과 Docker Desktop 재현 프로젝트에서는 원본 애플리케이션의 동작을 분석해 재현할 기능과 제외할 범위를 문서화하고, 직접 확인한 내용과 추정한 내용의 구분, 원본 소스·비공개 API 사용 금지 같은 제약조건을 정했습니다.

세부 설계

Collaborative Design. 방향이 정해지면 AI와 아키텍처, 데이터 구조, 기능별 처리 방식을 구체화했습니다. AI가 제안한 설계안을 기술적 제약과 요구사항에 맞춰 검토하고 구현 방향을 결정했습니다.

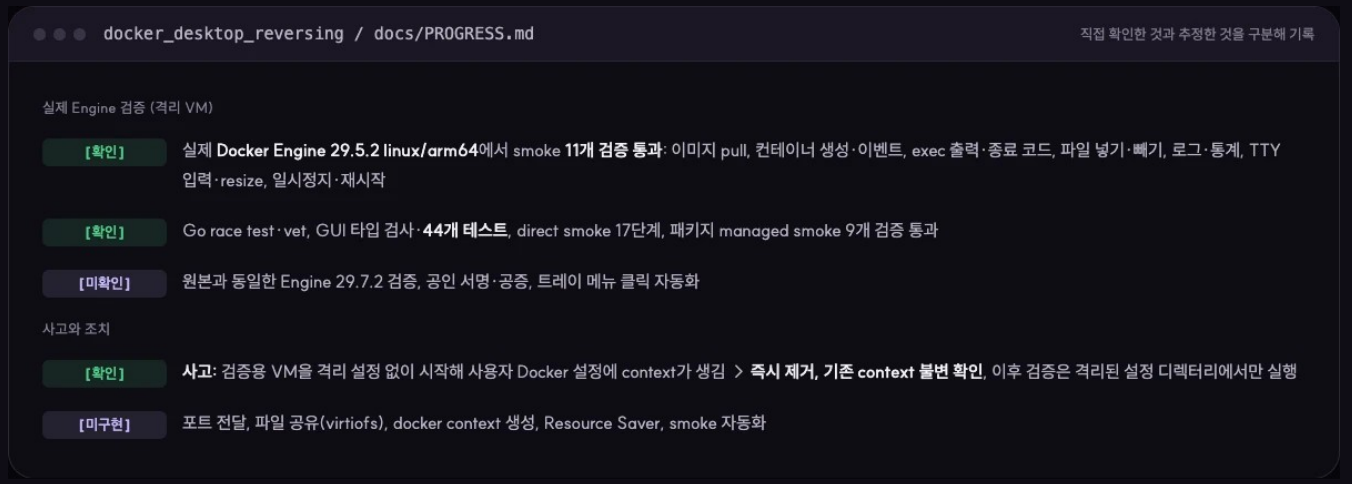
구현 위임

Agentic Implementation. Claude Code와 Codex에 구현을 맡기고 프로젝트 규약과 Agent Skills를 적용했습니다. 정해진 절차에 따라 기존 코드 분석, 기능 구현, 테스트와 수정을 수행하게 해 프론트엔드·백엔드·인프라 작업에 AI를 활용했습니다.

개인 프로젝트 — Harness 기반 Agentic Development Workflow (계속)

결과 검증

Review & Validation. AI의 작업 완료 보고보다 실제 실행 결과를 기준으로 구현 상태를 확인했습니다. Slack 재현 프로젝트에서는 서버 동작과 UI 구현을 원본과 비교해 일치하지 않는 부분을 따로 기록했습니다. Docker Desktop 재현 프로젝트에서는 실제 Docker Engine과의 연동을 검증했고, 검증 중 로컬 Docker 설정이 바뀌는 문제를 확인한 뒤 격리된 설정 환경에서 테스트하도록 개선했습니다.



Docker Desktop 재현 프로젝트의 진행 기록(PROGRESS.md) 일부

개인 프로젝트 — Harness 기반 Agentic Development Workflow (계속)

반복 개선

Iterative Improvement. 기능 구현 이후에도 AI로 코드 구조와 기능의 개선 가능성을 분석했습니다. UI/UX는 화면 구성과 사용자 흐름을 직접 확인하고, AI의 개선 제안과 비교하면서 수정 방향을 결정했습니다.

결과

청사진은 직접 수립하고, 세부 설계는 AI와 협업하며, 구현을 맡긴 뒤 검증과 개선을 반복하는 개발 방식을 적용했습니다.

AI의 역할은 코드 생성에 그치지 않고 설계 대안 검토와 개선점 탐색까지 넓혔고, 최종 의사결정과 결과 검증은 직접 수행했습니다.

염영인

Node.js · TypeScript 백엔드 엔지니어

Email	duaduddls1919@gmail.com
GitHub	https://github.com/Yeom-YoungIn
기술 블로그	https://threezerosin.tistory.com/category