

이윤하

NestJS로 결제와 비동기 작업을 다뤄온 백엔드 개발자

younha0088@gmail.com · github.com/labyrinth30

https://labyrinth30.github.io/

PROJECT 01 / WORK · PURPLE ACADEMY

Purple

주문 검색 SQL과 정기결제

교육 서비스의 결제 흐름과 관리자 백오피스를 개선했습니다.

ROLE

시스템개발부 · 백엔드 개발자

PERIOD

2026.04 — 현재

STACK

TypeScript · NestJS · MySQL · TypeORM · Redis · BullMQ

주문 검색 SQL 개선, 구독 단위 청구 경합 제어, 결제 응답 유실 복구, 셀프 재결제 약관 명시 동의를 맡았습니다.

회사 코드는 공개할 수 없어 설계와 검증 결과만 정리했습니다.

사례의 테스트 건수와 측정값은 당시 PR·개발 기록 기준입니다.

핵심 성과

- 01 주문 검색 count SQL **4.73초** → **0.89초**
 - 02 같은 구독에 **30개 요청이 동시에 와도** 청구는 **1번**
 - 03 PG 응답이 유실된 결제를 다시 조회해 **실패 안내 없이 성공으로 복구**
 - 04 서버가 대신하던 약관 동의를 **리스크로 보고해 정식 과제로 말고**, 사용자 명시 동의로 전환
-

주문 ID를 먼저 골라 검색 SQL 비용 줄이기

관리자 주문 검색이 주문마다 관련 테이블을 반복 확인해 느렸습니다.

검색어에 맞는 주문 ID를 먼저 모으는 방식으로 바꿔, 원본 검색의 count SQL을 4.73초에서 0.89초로 줄였습니다.

내가 맡은 범위

주문 검색 SQL 개선, 환불요청 검색 연결, 변경 전후 결과 대조

선택과 이유

실행 계획과 측정으로 정렬은 병목이 아님을 확인해 인덱스는 추가하지 않았습니다.

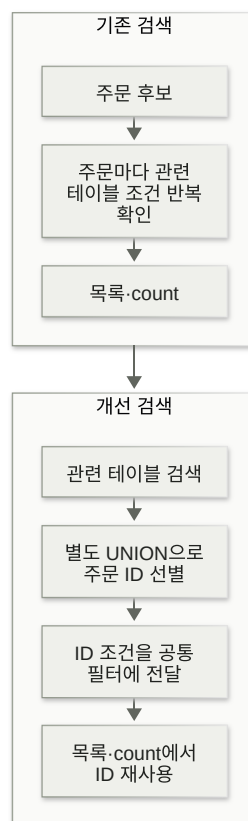
대신 검색어에 맞는 주문 ID를 별도 쿼리로 먼저 모아, 바깥 쿼리가 PK 조회만 하게 바꿨습니다.

트레이드오프

SQL 왕복과 ID 전달 비용이 생깁니다.

결과가 넓게 걸리는 1글자 검색은 1.86초에서 1.95초로 조금 느려졌습니다.

SYSTEM FLOW



관련 테이블에서 검색어에 맞는 주문 ID를 먼저 모읍니다.

목록과 count는 그 ID를 함께 씁니다.

문제

관리자 주문 검색은 회원·상품·결제 조건을 주문마다 상관 서브쿼리로 확인했습니다. 조회 기간이 길수록 이 반복 비용이 커져, 전체 기간에서 원번으로 검색하면 count SQL만 4.73초가 걸렸습니다.

해결 과정

1. 실행 계획 확인

먼저 EXPLAIN 을 돌렸습니다. 주문 테이블 전체를 훑으면서(type: ALL), 관련 테이블 조건마다 DEPENDENT SUBQUERY 가 7개 붙어 있었습니다. 주문 한 건마다 서브쿼리가 다시 실행되는 구조였습니다.

2. 인덱스 확인

주문 테이블에 생성일·유형 인덱스가 없어 정렬(filesort)이 생기는 것도 보였습니다. 하지만 정렬 없는 count(0.55초)와 정렬한 목록 20건(0.49초)이 거의 같아, 정렬은 병목이 아니라고 보고 인덱스는 추가하지 않았습니다.

3. 한 SQL 안에서 ID 먼저 고르기

o.id IN (SELECT ... UNION ...) 형태로 바꿔 봤습니다. 그런데 MySQL이 이를 다시 DEPENDENT SUBQUERY / DEPENDENT UNION 으로 실행해 실행 계획이 원래와 같았고, 시간도 4.55초로 그대로였습니다.

4. 쿼리를 둘로 나누기

검색어에 맞는 주문 ID를 별도 쿼리로 먼저 모으고, 바깥 쿼리는 그 ID로 PK 조회만 하게 했습니다. SQL 왕복은 한 번 늘었지만 원번 검색은 $0.33\text{초} + 0.56\text{초} = 0.89\text{초}$ 가 됐습니다.

주문 테이블 자체의 검색 조건은 그대로 두었고, 검색 안내 문구에는 있었지만 실제로 연결되지 않았던 환불요청 검색도 함께 연결했습니다.

측정

MySQL 8.0.42, 주문 약 11.6만 행, 전체 기간 기준입니다. 기존 count와 변경 후 ID 선별+count의 SQL 시간을 비교했습니다.

검색 조건	기존	변경 후
원번	4.73초	0.89초
구매처	9.60초	0.56초
상품명	4.45초	1.16초
성씨	4.38초	0.88초
1글자	1.86초	1.95초

다섯 조건 모두 변경 전후 결과 건수가 같았습니다. 같은 규모의 데이터로 조건마다 15번씩 다시 측정했을 때도 다섯 조건 모두 중앙값이 줄었습니다(원번 2.14초 → 0.55초).

여러 청구 경로의 경합을 구독 단위로 제어하기

정기결제 배치, 관리자 재결제, 학부모 재결제가 같은 구독을 동시에 청구할 수 있었습니다.

세 경로가 구독 단위로 같은 Redis 락을 쓰게 하고, 락을 잡은 뒤 결제 상태를 다시 확인하게 했습니다.

같은 구독에 30개 요청을 동시에 보내도 청구는 한 번만 실행됐습니다.

내가 맡은 범위

관리자·배치 청구의 구독 락 적용과 TTL 조정, 학부모 재결제의 락 키 통일과 상태 재확인

선택과 이유

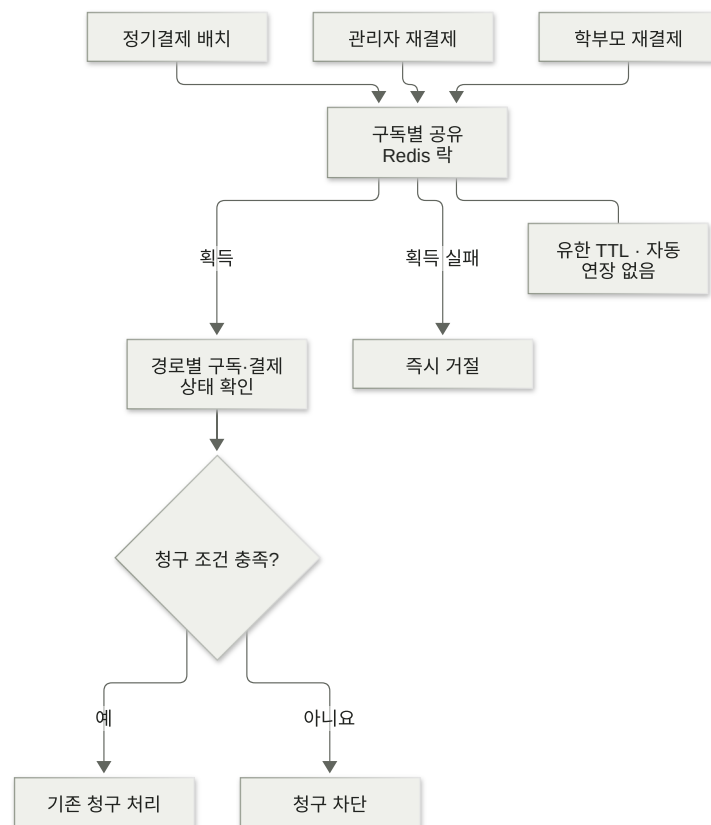
팀의 기존 공용 락을 구독 단위로 적용하고, 학부모 재결제는 락을 잡은 뒤 최신 결제 상태를 다시 확인하도록 했습니다.

트레이드오프

경합한 요청은 기다리지 않고 바로 거절됩니다.

락 유효시간보다 오래 걸리는 처리는 막지 못합니다.

SYSTEM FLOW



세 청구 경로를 구독 단위의 같은 락 키로 보호합니다.

경합한 요청은 기다리지 않고 바로 거절합니다.

문제

정기결제 배치, 관리자 수동 청구, 학부모 재결제가 같은 구독을 동시에 청구할 수 있었습니다. 관리자 API와 웹 API는 별도 프로세스라서, 경로마다 따로 막아서는 서로의 청구를 막을 수 없었습니다.

내 기여와 해결

팀의 공용 Redis 락을 관리자·배치 청구에 구독 단위로 적용하고 TTL을 조정했습니다. 학부모 재결제에도 같은 락 키 규칙을 적용해, 세 경로가 같은 구독에 대해 하나의 락을 다투게 했습니다.

락을 잡은 뒤에는 구독과 결제 상태를 다시 조회하도록 했습니다. 락을 기다리는 사이 다른 경로가 먼저 청구를 끝냈을 수 있기 때문입니다.

검증 결과

두 앱의 실제 락 코드와 Redis를 쓰고 청구 함수만 테스트용으로 바꿔, 같은 구독에 30개 요청을 동시에 보내는 시험을 20번 반복했습니다. 매번 청구는 1번만 실행되고 나머지 29개는 거절됐습니다.

서로 다른 구독은 동시에 청구됐고, 처리 중 예외가 나거나 프로세스가 강제 종료돼도 락이 풀리는 것을 확인했습니다.

CASE 03

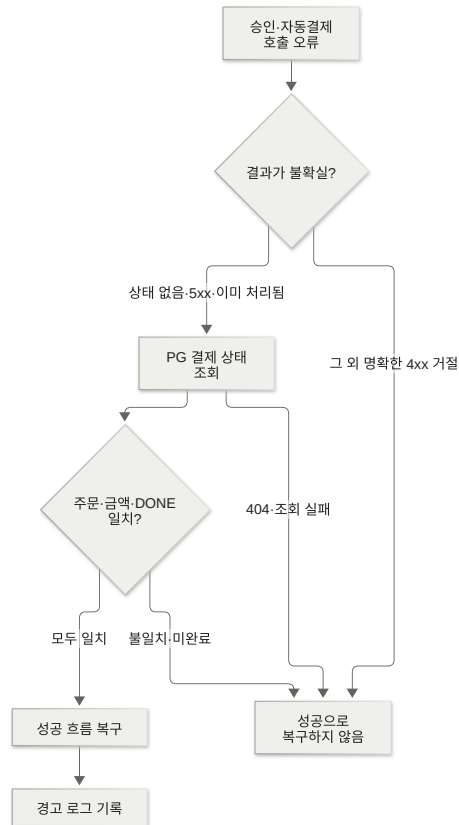
응답이 유실된 결제를 조회로 확인하고 복구하기

PG에서 결제가 끝났는데 응답만 유실되면, 서비스는 이를 실패로 보고 재청구 안내를 보낼 수 있었습니다.

결과가 불확실한 오류에서는 PG에 결제를 다시 조회하고, 주문번호·금액·완료 상태가 모두 맞을 때만 성공으로 처리했습니다.

내가 맡은 범위	카드·브랜드페이 승인과 자동결제 오류 분류, 조회 기반 복구 조건 적용
선택과 이유	결과가 불확실한 오류만 조회하고, 같은 주문번호·같은 금액·완료(DONE)가 모두 맞을 때만 기존 성공 흐름으로 넘겼습니다.
트레이드오프	해당 오류에서는 PG 조회가 한 번 더 일어납니다. 조회로도 확인되지 않으면 원래 오류를 그대로 둡니다.

SYSTEM FLOW



결과가 불확실한 오류만 PG에 다시 조회합니다.

주문번호·금액·완료 상태가 모두 맞아야 성공으로 처리하고, 아니면 원래 오류를 유지합니다.

문제

PG에서 결제가 완료됐어도 응답이 유실되면 서비스는 결과를 알 수 없습니다. 이를 바로 실패로 처리하면 실패 안내가 나가고, 같은 회차를 다시 청구할 수 있었습니다.

내 기여와 해결

카드·브랜드페이 승인과 자동결제에 **조회 기반 복구**를 적용했습니다. 상태 코드가 없는 오류, 5xx, “이미 처리된 결제” 오류가 나면 주문번호로 PG에 결제를 다시 조회합니다. 조회 결과가 **같은 주문번호, 같은 금액, 완료(DONE)** 상태일 때만 기존 성공 흐름으로 이어갑니다.

명확한 4xx 거절은 조회하지 않습니다. 404, 미완료, 주문·금액 불일치, 조회 실패는 성공으로 처리하지 않고 원래 오류를 그대로 둡니다. 성공으로 복구한 건은 로그를 남겨 나중에 추적할 수 있게 했습니다.

검증 결과

실제 관리자 주문·결제 코드와 Redis를 쓰고 PG와 DB만 테스트용으로 바꿔, PG가 결제를 완료한 뒤 응답만 유실되는 상황을 재현했습니다. **같은 결제에 10개 요청을 동시에 보내자 1개만 성공으로 복구되고 9개는 거절됐습니다.**

청구, 복구 조회, 결제 저장은 각각 1번씩만 일어났고, 실패 안내는 한 번도 나가지 않았습니다.

재결제 약관 동의를 서버가 대신하지 않게 하기

정기결제를 직접 재결제할 수 있게 만들면서, 재결제 API가 Toss 약관 동의를 거치지 않고 서버에서 대신 동의하고 넘어가던 것을 확인했습니다.

사용자가 재결제할 때는 이 때문에 오류가 났습니다.

리스크로 판단해 보고했고, 정식 과제로 맡아 사용자가 직접 약관에 동의하는 장치를 만들었습니다.

내가 맡은 범위

리스크 보고, 청구 직전 미동의 차단, 필수 약관 조회·동의 API, 재결제 화면 약관 UI

선택과 이유

서버 자동 동의를 쓰지 않고, 미동의 필수 약관이 남아 있으면 청구를 차단했습니다.

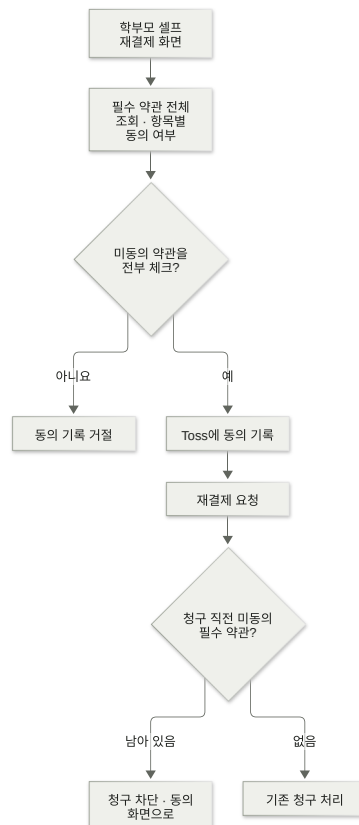
화면에 표시된 미동의 약관을 모두 체크해야만 동의를 기록합니다.

트레이드오프

사용자는 재결제 전에 약관 동의 단계를 한 번 더 거칩니다.

미동의 약관이 남아 있으면 재결제가 진행되지 않습니다.

SYSTEM FLOW



사용자가 직접 체크한 약관만 동의로 기록합니다.

청구 직전에 미동의 필수 약관이 남아 있으면 청구하지 않습니다.

문제

정기결제에 실패했을 때 학부모가 직접 재결제할 수 있는 기능을 만들면서, 재결제 API가 Toss의 약관 동의 절차를 거치지 않고 서버에서 대신 동의한 뒤 청구하고 있다는 것을 확인했습니다. 사용자가 확인하지 않은 약관에 동의가 기록되는 구조였습니다.

학부모가 직접 재결제할 때는 약관 동의가 기록되지 않은 상태라 오류가 발생했습니다. 서버의 대리 동의를 그대로 붙이면 오류는 바로 사라지지만, 같은 구조를 사용자 재결제에도 넓히는 셈이었습니다.

해결 과정

1. 보고와 과제화

이 구조를 리스크로 판단해 **경영지원팀과 경영진에 보고**했고, 정식 과제로 맡아 진행했습니다.

2. Toss 위젯의 약관 박스로 해결 시도

처음에는 Toss Payments 콘솔에서 결제 위젯의 약관 박스에 해당 약관을 **명시적으로 추가**했습니다. 그런데 개발 서버에서 QA를 해 보니, 재결제 상황에서는 그 박스에서 동의해도 **PG로 전송되지 않았습니다**. 재결제는 결제 요청 호출을 거치지 않고 저장된 결제수단으로 바로 청구하기 때문이었습니다.

3. 청구 직전 차단과 직접 동의 장치

그래서 동의를 Toss 위젯에 맡기지 않고 직접 받기로 했습니다. 재결제 API는 청구 직전에 미동의 필수 약관이 있는지만 확인하고, 남아 있으면 **청구를 차단**합니다. 이 경로에서는 서버가 동의 API를 호출하지 않습니다. 화면에서는 동의 체크를 하기 전까지 재결제 버튼을 막고, 동의 API는 화면에 표시된 미동의 약관을 모두 받은 경우에만 Toss에 동의를 기록합니다. 일부만 보내 우회하면 다시 거절합니다.

4. 동작하지 않는 Toss 박스 정리

재결제 화면에는 전송되지 않는 Toss 박스가 새 동의 체크와 함께 남아 있어 사용자를 헷갈리게 했습니다. 이 화면에서만 Toss 박스를 숨기고, 같은 형태의 자체 약관 박스로 바꿨습니다. 미동의 항목만이 아니라 **필수 약관 전체**를 보여주고, 이미 동의한 항목은 체크된 채 비활성으로 표시해 사용자가 자신이 동의한 약관을 한눈에 볼 수 있게 했습니다. 약관 전문은 Toss가 내려주는 원문 링크를 그대로 써서 사본과 원문이 어긋나지 않게 했습니다.

구현하며 확인한 점

Toss의 약관 동의 기록은 상점별로 따로 관리됩니다. 같은 고객이라도 상점이 다르면 동의 상태가 다르기 때문에, 그 주문의 결제 이력에 남은 상점 기준으로 조회하고 동의하도록 맞췄습니다. 또 Toss의 약관 조회 API는 이름과 달리 이미 동의한 약관까지 함께 내려주어, 이를 이용해 약관 전체와 항목별 동의 여부를 함께 내려주게 했습니다.

검증 결과

재결제·약관 조회·동의 관련 Jest 22건이 통과했습니다. 개발 서버에 배포한 뒤 미동의 계정에서 약관 3건이 미체크 상태로 표시되고, 전부 체크하기 전에는 결제 버튼이 비활성인 것을 확인했습니다. 전부 체크한 뒤 동의 API가 호출되고, Toss 조회에서 해당 약관이 미동의에서 동의로 바뀐 것을 직접 확인했습니다. 이미 전부 동의한 계정은 약관이 체크된 채 비활성으로 표시됐습니다.

운영 서버에 반영한 뒤, 같은 약관 미동의 오류가 더 이상 발생하지 않는 것을 확인했습니다.

GGUK

비동기 저장과 동시성 처리

Instagram 링크에서 장소를 뽑아 친구와 같이 쓰는 방에 저장하는 서비스입니다.

ROLE

백엔드 개발

PERIOD

2026.02 — 현재

STACK

TypeScript · NestJS · PostgreSQL · Drizzle · Cloud Run · Cloud Tasks

공통 응답·입력 검증·로깅, 링크 접수·워커 API와 저장, 핀·댓글 API를 맡았습니다.

AI 추출과 지오코딩은 팀원이 만든 코드를 썼습니다.

사례의 테스트 건수와 측정값은 당시 PR·개발 기록 기준입니다.

[Google Play에서 보기](#)

[App Store에서 보기](#)

[GitHub · 백엔드 코드](#)

핵심 성과

- 01 링크 접수 응답 **0.2~0.7초**, 분석이 일부 실패해도 **성공한 장소는 남기고 나머지만 재시도**
- 02 댓글 작성과 핀 삭제가 같은 행을 잠그게 해 **삭제된 핀에 댓글이 남지 않게 함**
- 03 큐 동시 실행 수를 서버 용량에 맞춰 7일간 사용자 API **429 14건 → 0건**

링크 접수와 장소 분석을 분리하고, 재시도해도 저장 결과가 남게 하기

링크는 202로 바로 접수하고, 분석이 일부만 성공해도 성공한 장소와 핀은 남긴 채 나머지만 재시도하도록 만들었습니다.

내가 맡은 범위

Cloud Tasks 접수·워커 API, OIDC 인증, 장소·출처·방 핀 저장과 재시도 처리

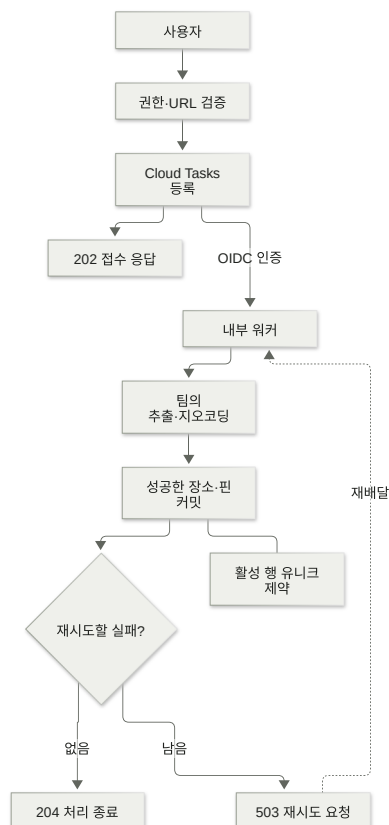
선택과 이유

성공한 장소는 먼저 커밋하고, 재시도할 실패가 남았을 때만 503을 반환했습니다.
같은 작업이 두 번 와도 활성 행 유니크 제약으로 중복 저장을 막았습니다.

트레이드오프

작업 상태 테이블과 폴링 API를 없어서 관리할 상태는 줄었지만, 202 응답만으로는 분석이 끝났는지 알 수 없습니다.

SYSTEM FLOW



접수 API와 내부 워커는 같은 Cloud Run 서비스에서 실행됩니다.
성공한 장소를 저장한 뒤 재시도할 실패가 남으면 503을 반환합니다.

문제

GGUK는 Instagram 링크에서 장소를 뽑아 방에 저장합니다. 그 사이에 스크래핑, AI 분석, 지오코딩을 거치는데, 외부 서비스가 느리거나 장소 여러 개 중 일부만 처리되는 경우가 있습니다. 링크를 받았다는 응답을 저장까지 끝났다는 뜻으로 쓰면 사용자가 보는 것과 실제 데이터가 달라집니다.

말은 일

AI 추출과 지오코딩은 팀원이 만든 코드를 쓰고, 저는 접수 API, 내부 워커, DB 저장을 연결했습니다. 접수할 때 방 멤버십과 URL을 확인하고, Cloud Tasks에 등록이 된 다음에만 202 Accepted 를 돌려줍니다. 내부 워커는 OIDC로 인증합니다.

이렇게 한 이유

원래 있던 job 테이블, jobId, 폴링 API를 없애고 접수와 처리를 나눴습니다. 중간 상태를 맞출 일이 없어진 대신, 이 API의 응답은 “받았다”는 뜻일 뿐 결과 알림은 아니게 되었습니다.

워커는 장소마다 지오코더 1순위 후보를 저장합니다. 성공한 장소·출처·핀을 먼저 커밋하고, 일시적인 지오코딩 실패가 남아 있으면 503 을 반환해 Cloud Tasks가 다시 보내게 했습니다. 이렇게 하면 같은 작업이 여러 번 들어올 수 있어서, 장소는 upsert하고 출처와 핀은 활성 행 유니크 제약으로 충돌을 처리했습니다.

코드

핀 INSERT 뒤에 붙는 충돌 처리입니다. 같은 방, 같은 장소에 삭제되지 않은 핀이 이미 있으면 넘어갑니다. 삭제된 핀은 제약 대상이 아니라서 나중에 다시 등록할 수 있습니다.

```
.onConflictDoNothing({
  target: [pins.roomId, pins.placeId],
  where: isNull(pins.deletedAt),
});
```

[원본 코드 · place-result.repository.ts 180–183행](#)

검증 결과

PR #74에서 실제 HTTP 요청과 embedded PostgreSQL로 접수→워커→저장, 부분 실패 후 재시도, 중복 배달, 삭제 후 재등록을 테스트했습니다. 유닛 147건, E2E 28건이 통과했습니다.

운영에서 링크 접수 응답은 0.2~0.7초였고, 분석 작업은 성공 73건 기준 중앙값 6.6초(p90 13.2초)가 걸렸습니다.

공개 근거

[PR #38 · enqueue-only 전환과 담당 범위 ↗](#)

[PR #74 · 방 핀 저장과 부분 실패 재시도 ↗](#)

[PR #152 · 운영 시간 측정 기록 ↗](#)

삭제된 핀에 댓글이 남지 않게 하기

댓글 작성과 핀 삭제가 같은 핀 행을 잠그도록 순서를 맞춰, 동시에 들어와도 삭제된 핀에 댓글이 남지 않게 했습니다.

내가 맡은 범위

핀 삭제 API, 댓글 생성 시 행 잠금, 삭제 트랜잭션과 권한·재등록 검증

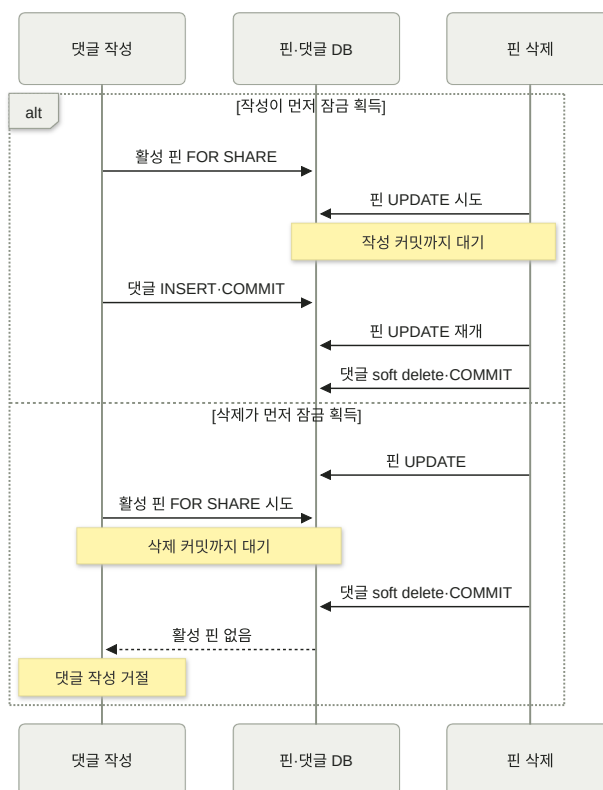
선택과 이유

댓글 작성은 FOR SHARE로 핀을 읽은 뒤 INSERT하고, 삭제는 핀을 먼저 UPDATE한 뒤 댓글을 soft delete합니다.

트레이드오프

같은 핀에 작성과 삭제가 겹치면 한쪽이 기다립니다.
댓글 작성끼리는 공유 잠금이라 동시에 진행됩니다.

SYSTEM FLOW



작성이 먼저면 삭제가 커밋을 기다렸다가 새 댓글까지 지웁니다.

삭제가 먼저면 작성이 기다린 뒤 핀이 없는 걸 보고 거절됩니다.

문제

핀과 댓글을 한 트랜잭션에서 지워도, 댓글을 먼저 지우면 그 직후에 다른 요청이 새 댓글을 넣을 수 있습니다. 그러면 핀은 삭제됐는데 댓글은 살아 있게 됩니다. 댓글을 쓰기 전에 핀이 있는지 조회하는 것만으로는 조회와 INSERT 사이에 끼어드는 삭제를 막을 수 없습니다.

말은 일

핀 삭제 API를 만들면서 “**삭제된 핀에는 활성 댓글이 없어야 한다**”를 지켜야 할 조건으로 잡았습니다. 삭제는 그 방의 핀과 댓글에만 적용하고, 여러 방이 같이 쓰는 장소 데이터는 건드리지 않았습니다.

이렇게 한 이유

댓글 작성은 트랜잭션 안에서 핀을 `SELECT ... FOR SHARE` 로 읽고 나서 댓글을 넣습니다. 핀 삭제는 핀을 먼저 UPDATE해서 잠금을 잡고, 같은 트랜잭션에서 댓글을 soft delete합니다. 삭제 쿼리에는 방 멤버 조건도 넣었습니다.

작성이 먼저 잠금을 잡으면 삭제는 작성이 커밋될 때까지 기다렸다가 방금 생긴 댓글까지 지웁니다. 삭제가 먼저면 작성이 기다린 뒤 핀이 없는 걸 확인하고 실패합니다.

같은 핀에서 작성과 삭제가 겹치면 대기가 생기지만, 댓글 작성끼리는 공유 잠금이라 서로 막지 않습니다. 애플리케이션에서 미리 조회하는 대신 DB 잠금으로 확인과 쓰기 사이를 묶었습니다.

코드

```
const [pin] = await tx
  .select({ id: pins.id })
  .from(pins)
  .where(and(eq(pins.id, pinId), isNull(pins.deletedAt)))
  .for("share");
```

[원본 코드 · comment.repository.ts 68–72행](#)

검증 결과

PR #132에서 실제 HTTP 요청으로 핀·댓글 삭제, 목록 제외, 중복 삭제, 권한·입력 검증, 삭제 후 같은 장소 재등록을 테스트했습니다. 유닛 266건, E2E 148건이 통과했습니다. 재등록은 활성 행에만 걸리는 partial unique 인덱스 덕분에 가능합니다.

공개 근거

[PR #132 · 핀 삭제, 댓글 경합과 검증 기록 ↗](#)

[고정 커밋 · 핀 우선 갱신과 댓글 삭제 ↗](#)

큐 동시 실행 수를 서버 용량에 맞추기

사용자 API와 워커가 같은 Cloud Run을 나눠 쓰는데, 큐가 서버 용량보다 많은 작업을 동시에 보내 사용자 요청이 429로 거절됐습니다.

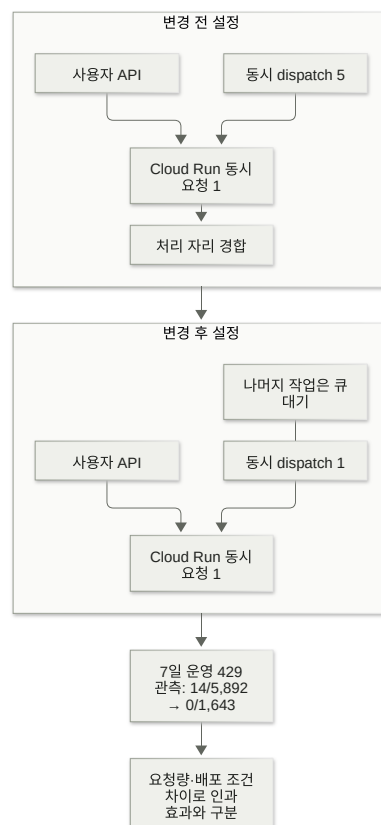
큐 동시 실행 수를 5에서 1로 맞춘 뒤 7일간 사용자 API 429는 14건에서 0건이 됐습니다.

내가 맡은 범위 운영 로그 분석, 큐 동시 dispatch 조정, 장소 추출 단계별 시간 로깅

선택과 이유 maxConcurrentDispatches를 5에서 1로 낮춰 대기는 큐에서 하게 하고, 어느 단계가 느린지 보려고 단계별 시간을 기록했습니다.

트레이드오프 큐 대기가 늘어 추출 결과가 보이기까지 더 오래 걸릴 수 있습니다.
같은 서비스를 쓰는 이상 사용자 요청이 먼저 처리되지는 않습니다.

SYSTEM FLOW



조사 당시 Cloud Run은 인스턴스 1개, 동시 요청 1개였는데 큐는 최대 5개를 동시에 보냈습니다.
큐 동시 실행 수를 서버 용량에 맞췄습니다.

문제

사용자 API와 내부 워커가 같은 Cloud Run을 나눠 쓰는데, 당시 서버는 인스턴스 1개·동시 요청 1개였고 큐는 최대 5개를 동시에 보내고 있었습니다. 2026년 9월 12일 하루에만 **사용자 API 요청 14건이 429**로 거절됐습니다.

말은 일

로그로 요청 거절과 워커 부하를 확인하고 큐의 `maxConcurrentDispatches`를 **5→1**로 낮췄습니다. 장소 추출을 스크래핑·이미지 업로드·AI 호출·지오코딩으로 나눠, 단계별 시간을 기록하는 로그도 추가했습니다.

이렇게 한 이유

링크 접수는 큐에 넣고 바로 응답하므로 워커가 조금 늦어도 사용자는 앱을 계속 쓸 수 있습니다. 반면 조회 요청이 거절되면 바로 드러납니다. 그래서 서버를 늘리기 전에 **큐를 서버 용량에 맞춰, 기다리는 일은 큐가 맡게** 했습니다. 서버 증설이나 워커 분리는 부하와 비용을 보고 정할 다음 단계로 두었습니다.

결과

변경 전후 7일씩 사용자 API 요청을 비교했습니다.

구간	사용자 API 요청	429
변경 전 7일 (9/10–9/17)	5,892건	14건
변경 후 7일 (9/17–9/24)	1,643건	0건

10월 5일까지 관측을 이어가도 **2,181건 중 429는 1건**이었습니다. 단일 처리 자리로 재현한 로컬 실험에서도 사용자 요청 거절이 **2,250건 중 1,023건에서 0건**으로 줄었고, 대신 태스크 대기 시간이 늘어나는 것을 확인했습니다.

PR #152는 인프라 코드의 Biome·TypeScript 검사를, PR #153은 유닛 테스트 285건을 통과했습니다.

공개 근거

[PR #152 · 서버 용량과 큐 동시 실행 수 ↗](#)

[PR #153 · 추출 단계별 시간 로깅 ↗](#)

그 밖의 활동

OTHER PROJECT

2025.02 – 2025.07

KOKKOK(콕콕) ↗

Mash-Up 15기 · 백엔드

친구들과 예약 일정을 공유하고 준비 상태부터 성공 여부까지 같이 확인하는 앱.

CI/CD 구축, FCM 푸시 알림, 엔티티 설계, 예약 목록·결과 조회 API, 콕 찌르기 기능을 맡았습니다.

COMMUNITY

2026.02 – 2026.09

Mash-Up 16기 | Node Team 파트장

IT에 관심 있는 개발자와 디자이너가 함께 모여 협업하고 성장하는 IT 연합동아리

2025.02 – 2025.07

Mash-Up 15기 | Node Team

2023.07 – 2024.06

GDG on Campus SKHU Organizer

Google Developers가 지원하는 대학 개발자 커뮤니티

STUDY

2025.04 – 2025.06

Kubernetes 스터디 ↗

Mash-Up 15기 Node·Spring 합동

2024.07 – 2024.08

코어 자바스크립트 스터디 ↗

8주간 매주 정리 제출과 상호 리뷰

PRESENTATION

Entity와 DTO 대신 Schema를 믿기로 했다 ↗

Mash-Up 16기 Node Team 세미나

AWS Serverless로 진짜 On-Demand 알아보기 ↗

Mash-Up 15기 Node Team 세미나

들어봤니 GraphQL ↗

GDG on Campus SKHU 테크톡

WRITING

당신의 API 때문에 유저에게 서비스 규모를 들키고 있습니다 — NestJS에서 PK 암호화하기 ↗

TypeORM의 forFeature 반복 문제 해결하기 — 서비스의 SRP를 위한 Repository ↗