



BACKEND

어려운 개념 5가지, 코드로 이해하기

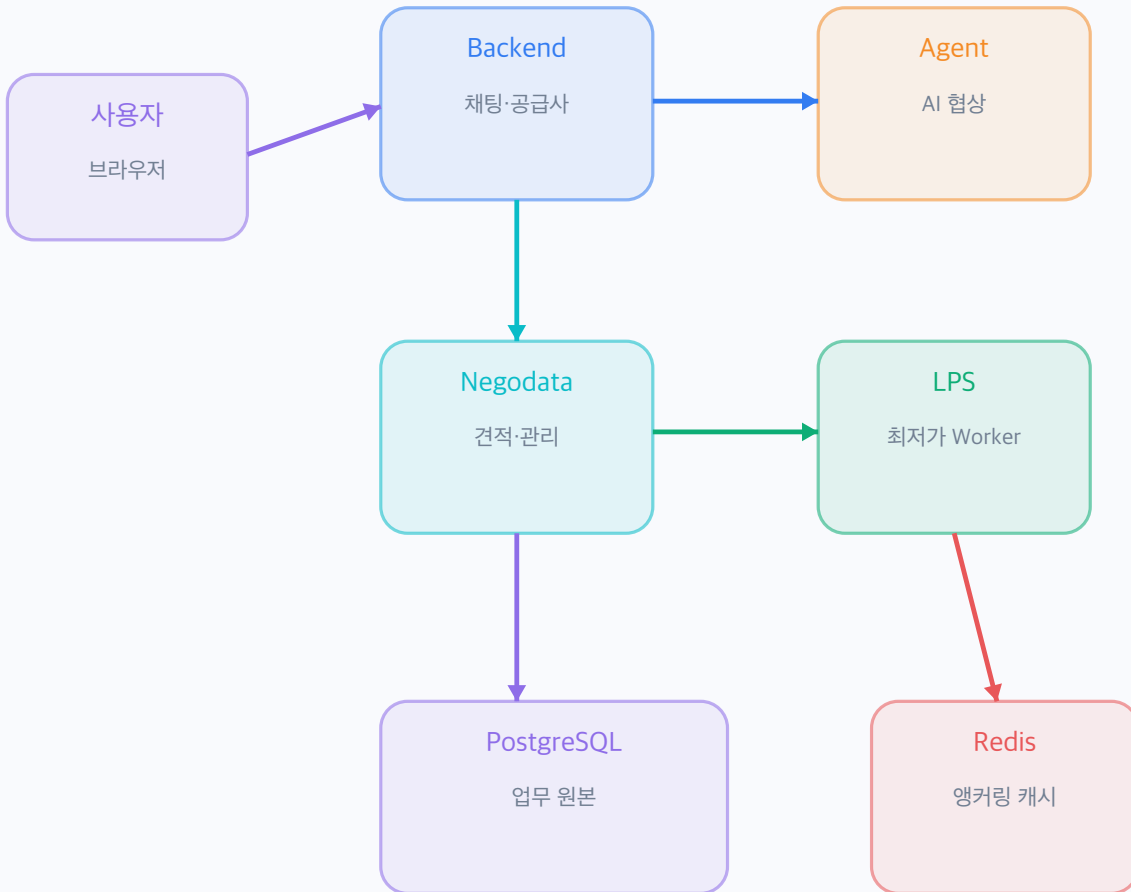
분산 시스템 · 트랜잭션/동시성 · 멀티테넌시
캐시 정합성 · 스케줄러/배치

이 문서는 이렇게 읽어요

- ① 일상 비유로 개념 잡기
- ② 실제 서비스 흐름을 그림으로 보기
- ③ 프로젝트 코드에서 구현 확인하기
- ④ 없을 때 생기는 문제와 비교하기

먼저, 서비스 지도를 봅시다

다섯 개념은 따로 노는 것이 아니라, 한 요청이 여러 서비스와 저장소를 지나면서 함께 작동합니다.



① 경계가 생기면 '분산 시스템'

서비스 A가 서비스 B를 네트워크로 호출하는 순간, 지연·타임아웃·부분 실패를 다뤄야 합니다.

② 여러 실행자가 만나면 '동시성'

사용자 클릭과 스케줄러가 같은 견적을 동시에 마감할 수 있어, DB가 최종 심판 역할을 합니다.

③ 빠르게 읽되 원본을 지키면 '캐시'

Redis와 프로세스 메모리는 복사본입니다. PostgreSQL과 설정 파일이 원본입니다.

④ 회사별 경계를 지키면 '멀티테넌시'

요청 헤더에서 회사 ID를 결정하고, 회사별 설정·엔진을 선택합니다.

분산 시스템 — 개념부터

여러 독립 실행 단위가 네트워크를 통해 하나의 업무를 완성하는 시스템

정확한 정의

프로세스·컨테이너·서버가 각자 메모리와 실행 상태를 가지고, HTTP나 메시지로 통신하는 구조입니다. 한 서비스의 함수 호출과 달리 상대의 상태를 직접 볼 수 없고, 네트워크 응답만으로 결과를 추론해야 합니다.

왜 어려운가: 네트워크에는 네 가지 결과가 있습니다

성공

상대가 처리했고 응답도 받음

명확한 실패

상대가 오류 응답을 보냄

연결 실패

상대에게 요청이 도착하지 않음

애매한 타임아웃

처리는 됐지만 응답만 늦었을 수도 있음

대표적인 대응 수단

Timeout

얼마나 기다릴지 상한을 둡니다.
짧으면 정상 요청도 실패하고, 길면
자원이 오래 묶입니다.

Retry

일시 실패를 다시 시도합니다. 단,
중복 처리에 안전한 작업에서만
제한적으로 사용합니다.

Idempotency

같은 요청을 여러 번 보내도 결과가
한 번 처리한 것과 같도록 만듭니다.

Fallback

주 서비스가 실패하면 대체 경로·기본값·이전 데이터를
사용합니다. 대체 결과가 업무적으로 허용될 때만
가능합니다.

Circuit breaker

실패가 계속되는 서비스를 잠시 호출하지 않아 연쇄 장애를
막습니다. 현재 프로젝트에는 명시적 구현이 없습니다.

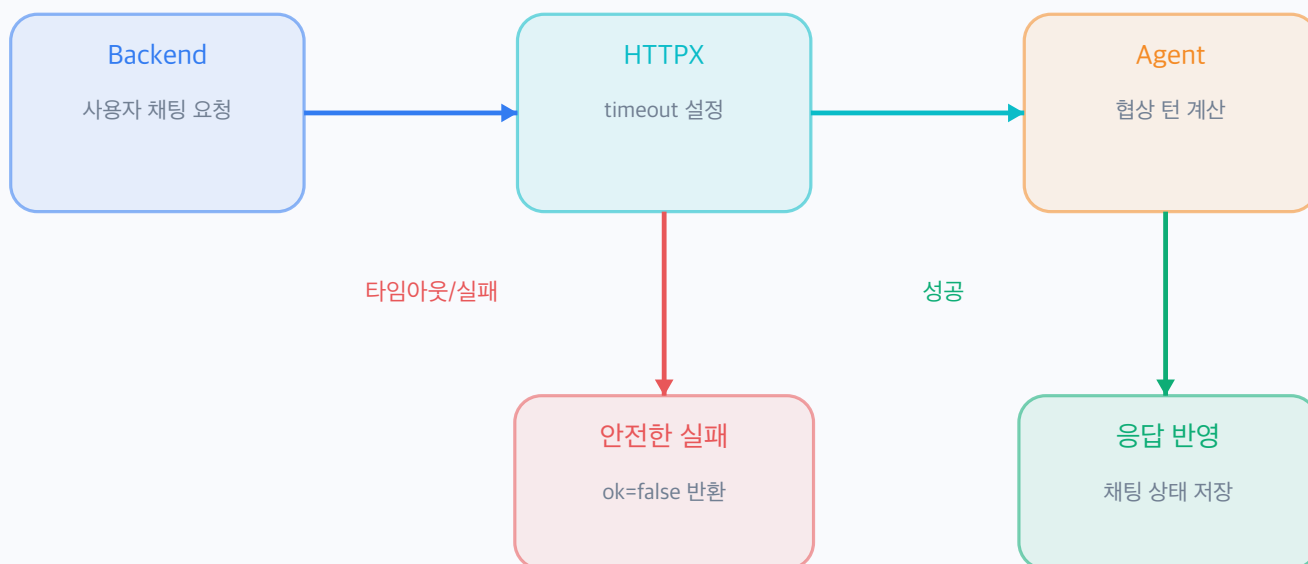
분산 시스템과 장애 대응

한 프로그램이 아니라 여러 서비스가 네트워크로 협력하는 구조

쉬운 비유

한 식당 안에서 주방과 홀 직원이 말로 협업하는 것이 단일 시스템이라면, 분산 시스템은 서로 다른 건물의 팀이 전화로 협업하는 것입니다. 전화는 늦거나 끊길 수 있고, 상대가 일을 끝냈는데 답만 못 받을 수도 있습니다.

프로젝트의 대표 흐름



중요한 함정: 타임아웃 ≠ 상대가 아무 일도 안 함

Agent가 DB 상태를 이미 전진시킨 직후 응답만 늦었을 수 있습니다. 그래서 무조건 재시도하면 같은 턴을 두 번 처리할 위험이 있습니다. 코드가 `timed_out`을 따로 표시하는 이유입니다.

이 장치가 없으면

Agent가 느린 순간 Backend 요청도 끝없이 대기합니다. 무작정 재시도하면 협상 step이 두 번 전진할 수 있습니다.

현재 방식

HTTP timeout을 두고 성공/일반 실패/타임아웃을 구분합니다. 호출 경계에서 예외를 응답 객체로 변환합니다.

분산 시스템 — 실제 코드

서비스 경계마다 timeout, fallback, best-effort 정책이 다릅니다.

Agent 호출: 타임아웃을 별도 상태로 반환

`backend/services/agent_client.py · lines 81-91`

```
async with httpx.AsyncClient(
    base_url=agent_config.base_url,
    timeout=agent_config.timeout_sec,
) as cli:
    resp = await cli.post("/v1/chat", json=body, headers=headers)

except httpx.TimeoutException as ex:
    # Agent가 이미 처리했을 수 있어 단순 재시도는 위험
    return AgentTurn(ok=False, timed_out=True)
except Exception:
    return AgentTurn(ok=False)
```

카탈로그 변경 알림: 핵심 업무를 막지 않는 best-effort

`negodata/backend/services/agent_notify.py · lines 15-26`

```
try:
    async with httpx.AsyncClient(timeout=3.0) as cli:
        await cli.post(f"{base}/v1/catalog-refresh-all")
except Exception as ex:
    # 알림 실패는 카드 작업을 막지 않는다
    LOG.w(f"[agent_notify] 변경 알림 실패(무시): {ex}")
```

어떻게 정책을 고르나요?

결제처럼 반드시 성공해야 하는 작업은 실패를 호출자에게 알려 재처리해야 합니다. 반면 '캐시 무효화 알림'처럼 보조적인 작업은 실패해도 핵심 카드 변경을 성공시킬 수 있습니다. 모든 외부 호출을 똑같이 재시도하면 안 됩니다.

기억할 단어

timeout

partial failure

best-effort

idempotency

트랜잭션·동시성 — 개념부터

데이터의 일관성을 지키는 작업 단위와, 동시에 실행되는 요청을 제어하는 방법

트랜잭션의 ACID

A · Atomicity

전부 성공하거나 전부 취소

C · Consistency

규칙을 만족하는 상태로 이동

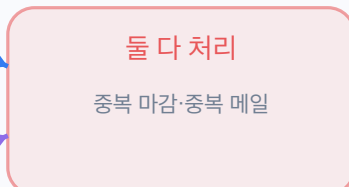
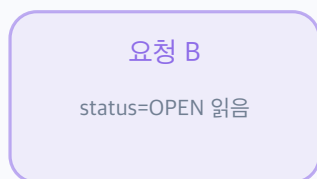
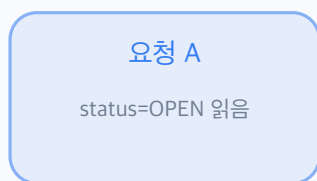
I · Isolation

동시 작업의 중간 상태를 서로 숨김

D · Durability

COMMIT된 결과는 장애 후에도 보존

동시성 문제는 어떻게 생기나?



해결 ① 비관적 잠금

SELECT ... FOR UPDATE로
먼저 행을 잠급니다.
명확하지만 잠금 대기과
.....

해결 ② 조건부 갱신

UPDATE ... WHERE
status=OPEN 후 rowcount를
확인합니다. 상태 검사와
.....

격리 수준과 잠금은 만능이 아님

격리를 높이면 안전성은 커지지만 동시 처리량이 줄고 대기·교착 가능성이 커집니다. 업무 규칙에 맞는 최소 범위의 트랜잭션과 조건부 상태 전이가 실용적입니다.

COMMIT / ROLLBACK

COMMIT은 변경 확정, ROLLBACK은 현재 트랜잭션의 미확정 변경 취소입니다. 외부 이메일 발송은 DB rollback으로 되돌릴 수 없다는 점도 중요합니다.

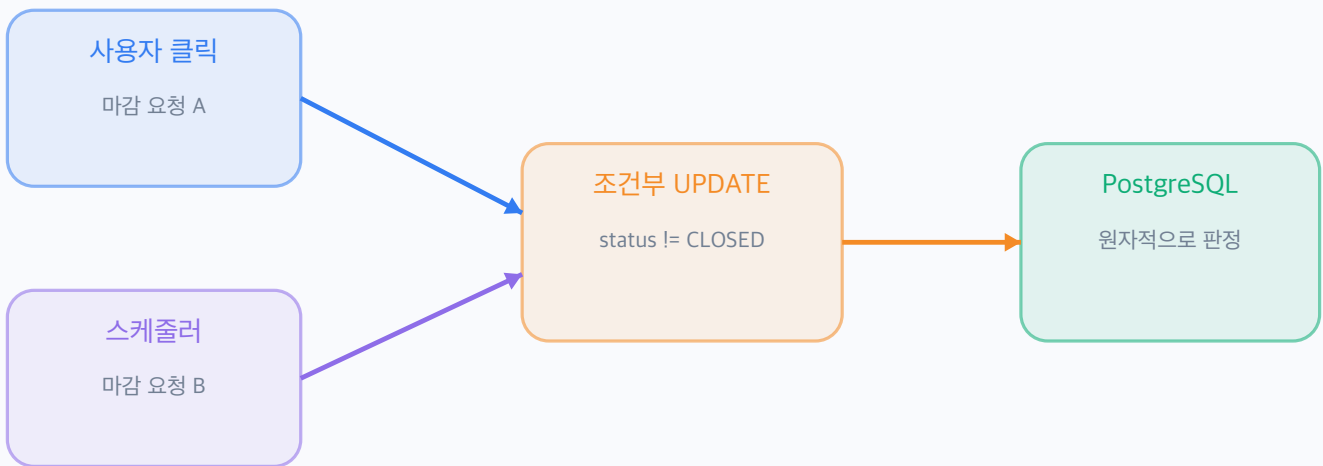
DB 트랜잭션과 동시성 제어

여러 작업을 하나로 묶고, 동시에 온 요청 중 한 명만 통과시키는 기술

트랜잭션이란?

은행 이체에서 '내 계좌 차감'과 '상대 계좌 증가'가 둘 다 성공하거나 둘 다 취소되어야 하듯, 관련 DB 변경을 하나의 작업 단위로 묶는 것입니다. 중간에 실패하면 rollback합니다.

동시 마감 문제



승자

영향받은 행 수(rowcount) = 1
마감 판정 권한 획득

패자/재요청

rowcount = 0
이미 닫혔으므로 추가 처리 중단

단순 SELECT 후 UPDATE

두 요청이 동시에 OPEN을 읽으면 둘 다 마감·낙찰 로직을 실행할 수 있습니다. 이메일도 두 번 발송될 수 있습니다.

조건부 UPDATE

DB가 상태 검사와 변경을 한 문장으로 처리합니다. 먼저 성공한 요청만 rowcount=1을 받습니다.

트랜잭션·동시성 — 실제 코드

애플리케이션의 if문보다 DB의 원자적 UPDATE가 강한 최종 방어선입니다.

조건부 상태 전이: 마감 권한 선점

negodata/backend/crud/quotation_crud.py · lines 482-500

```
query = (
    update(quotations)
    .where(
        quotations.qt_id == qt_id,
        quotations.status != QuotationStatus.CLOSED.value,
        quotations.deleted == False,
    )
    .values(
        status=QuotationStatus.CLOSED.value,
        updated_at=GTime.UTC(),
    )
)
return await DB_SESSION_MNG.add_with_rowcount(cdb, query)
```

트랜잭션 실패 시 rollback

negodata/backend/common/database/db_session_manager.py · lines 116-127

```
try:
    await db.commit()
    return ErrorType.SUCCESS
except IntegrityError:
    await db.rollback()
    return ErrorType.DB_ALREADY_SAME_KEY
except Exception:
    await db.rollback()
    raise
```

왜 rowcount를 보나요?

UPDATE가 여러 없이 실행됐다는 사실만으로는 내가 상태를 바꿨는지 알 수 없습니다. WHERE 조건에 맞는 행이 없으면 SQL은 정상 실행되지만 변경 행은 0개입니다. 그래서 1이면 승자, 0이면 이미 다른 요청이 처리한 것으로 판단합니다.

실무 체크

트랜잭션은 짧게 유지하고, 외부 HTTP-이메일처럼 오래 걸리는 작업을 DB 트랜잭션 안에 오래 붙잡아 두지 않습니다.

멀티테넌시 — 개념부터

하나의 애플리케이션을 여러 고객사가 공유하면서 논리적으로 격리하는 설계

Tenant란?

서비스를 사용하는 독립 고객 단위입니다. 이 프로젝트에서는 주로 '회사'가 tenant입니다. 같은 API와 서버를 쓰더라도 회사별 데이터, 설정, 권한, 협상 정책이 섞이면 안 됩니다.

대표적인 데이터 격리 모델

DB 분리

회사마다 별도 DB
격리 강함 · 운영비 높음

Schema 분리

한 DB 안에서 schema 분리
중간 수준의 격리와 비용

Row 공유

같은 테이블 + tenant_id
효율적 · 쿼리 누락 위험

격리는 DB만의 문제가 아닙니다

식별

이 요청이 어느 회사 것인지 신뢰할 수 있게 결정

인가

그 사용자가 해당 회사 자원에 접근 가능한지 확인

데이터

모든 조회·수정 쿼리에 회사 범위 적용

설정

회사별 정책·브랜딩·카드 선택

캐시

캐시 key에 tenant를 포함해 회사 간 충돌 방지

자원

한 회사의 과부하가 다른 회사에 미치는 영향 제한

가장 흔한 사고

쿼리의 WHERE tenant_id 조건 누락, 공유 캐시 key에 tenant_id 누락, 사용자가 body로 보낸 tenant_id를 그대로 신뢰하는 경우입니다. 그래서 tenant context를 요청 초기에 확정하고 자동 전달하는 구조가 중요합니다.

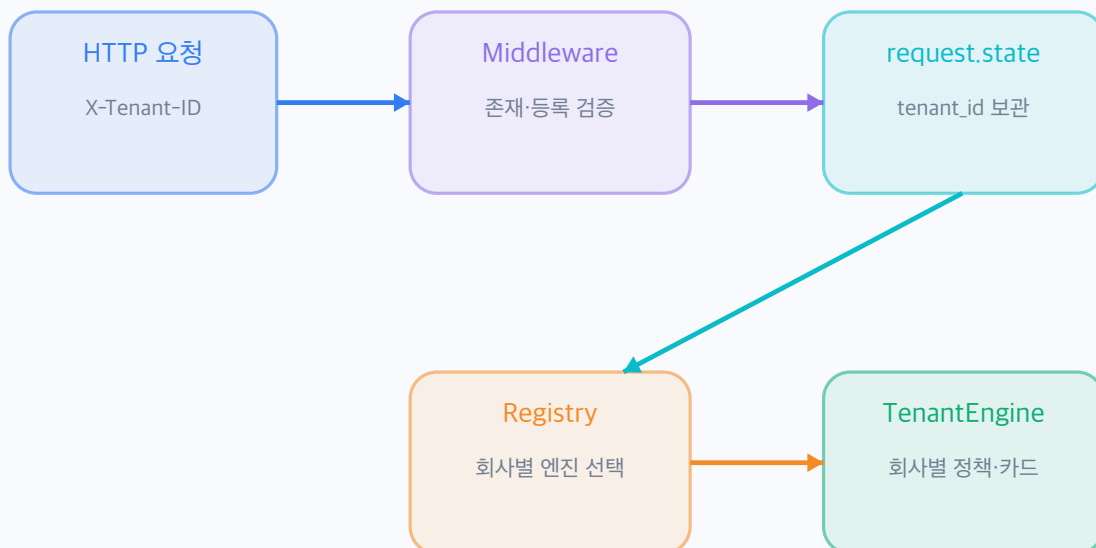
멀티테넌시

하나의 시스템을 여러 회사가 쓰되, 설정과 데이터의 경계를 지키는 구조

쉬운 비유

한 오피스 빌딩을 여러 회사가 함께 사용하지만 출입카드가 자기 회사 층만 열어주는 구조입니다. 서버는 공유하되, 요청마다 '어느 회사의 요청인지'를 먼저 확정해야 합니다.

요청이 회사별 엔진을 찾는 과정



보안 핵심

tenant_id를 요청 body에서 받으면 사용자가 다른 회사 ID를 넣어 위조할 수 있습니다. 이 프로젝트는 헤더/경로에서 결정한 값을 middleware가 request.state에 넣고, 뒤의 코드가 그것만 사용합니다.

멀티테넌시 경계가 약하면

A회사 요청이 B회사 카드·설정·협상 엔진을 사용할 수 있습니다. 이는 단순 버그가 아니라 데이터 유출 사고입니다.

현재 방식

요청 시작점에서 tenant를 검증하고, Registry가 해당 회사의 설정과 엔진을 해석합니다.

식별 → 검증 → request.state 전달 → 회사별 엔진 선택의 4단계

Middleware: tenant를 요청 경계에서 확정

[agent/router/middleware/tenant_middleware.py · lines 35-69](#)

```
tenant_id = request.headers.get("X-Tenant-ID")

if not tenant_id:
    return JSONResponse(status_code=400, ...)

if not tenant_registry.is_registered(tenant_id):
    return JSONResponse(status_code=404, ...)

request.state.tenant_id = tenant_id
return await call_next(request)
```

Dependency: 검증된 tenant로 엔진 조회

[agent/router/deps.py · lines 13-21](#)

```
async def get_tenant_engine(request: Request) -> TenantEngine:
    tenant_id = getattr(request.state, "tenant_id", None)
    if not tenant_id:
        raise EXCEPTION_TENANT_HEADER_MISSING
    return await tenant_registry.get_engine(tenant_id)
```

Registry: 프로세스 메모리에서 회사별 엔진 재사용

[agent/tenancy/registry.py · lines 85-98](#)

```
cached = self._engines.get(tenant_id)
if cached is not None:
    return cached

async with self._locks[tenant_id]:
    cached = self._engines.get(tenant_id)
    if cached is not None:
        return cached
    engine = await self._build(tenant_id)
    self._engines[tenant_id] = engine
    return engine
```

캐시 정합성 — 개념부터

비싼 계산·DB·외부 호출의 결과를 가까운 곳에 복사해 재사용하는 기술

기본 용어

Hit

캐시에 값이 있어 원본을 읽지 않음

Miss

값이 없어 원본을 읽고 캐시를 채움

TTL

값이 자동 만료될 때까지의 시간

Stale

원본은 바뀌었지만 캐시는 옛 값인 상태

대표적인 읽기·쓰기 패턴

Cache-aside

앱이 캐시를 먼저 조회하고 miss이면 DB를 읽어 캐시에 저장합니다. 단순하고 가장 흔하지만 무효화를 앱이 책임집니다.

Write-through

쓰기 때 캐시와 원본을 함께 갱신합니다. 읽기는 안정적이지만 쓰기 지연과 두 저장소의 부분 실패를 다뤄야 합니다.

Write-behind

캐시에 먼저 쓰고 DB는 나중에 반영합니다. 빠르지만 캐시 장애 시 데이터 유실 위험이 있어 업무 원본에는 신중해야 합니다.

Negative cache

‘결과 없음’도 잠깐 저장합니다. 반복 실패 비용을 줄이지만 너무 긴 TTL은 새로 생긴 데이터를 늦게 발견하게 합니다.

캐시에서 자주 생기는 문제

Invalidation

원본 변경 후 어떤 key를 언제 삭제·갱신할지 결정하기 어렵습니다.

Stampede

인기 key가 만료되는 순간 많은 요청이 동시에 DB로 몰립니다.

Key 설계

tenant·버전 등이 빠지면 서로 다른 데이터가 같은 key를 공유합니다.

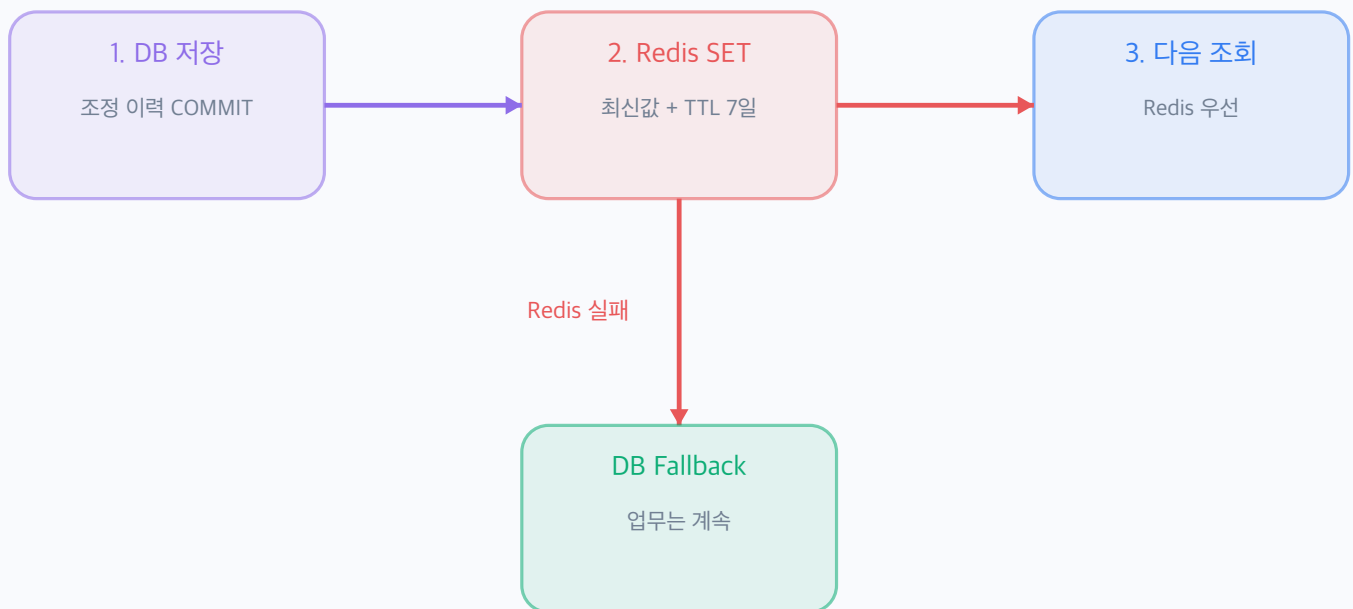
캐시 정합성과 무효화

빠른 복사본이 원본과 다른 값을 갖지 않도록 관리하는 문제

캐시는 복사본

도서관 검색대의 메모가 캐시이고, 원본 장부가 DB라고 생각하면 쉽습니다. 메모는 빠르지만 오래된 정보일 수 있습니다. 정합성이란 메모와 장부가 의미상 같은 상태를 유지하는 것입니다.

앵커링 값의 저장·조회 순서



stale 데이터란?

DB에는 새 값 600이 저장됐는데 Redis SET이 실패해 캐시에 옛 값 55가 남은 상태입니다. 캐시 miss와 달리 값이 존재하므로 더 위험합니다. TTL과 주간 re-SET으로 회복합니다.

캐시만 믿으면

Redis 장애가 업무 장애가 되고, 오래된 값이 실제 제안을 왜곡할 수 있습니다. Redis 유실 시 원본도 사라집니다.

원본 DB + 파생 캐시

Redis 장애 시 DB를 읽고, TTL과 reconciliation으로 오래된 복사본을 교정합니다.

캐시 정합성 — 실제 코드

Cache-aside, TTL, DB fallback, reconciliation이 한 세트로 작동합니다.

Cache-aside: Redis miss → DB → Redis backfill

`schedules/anchoring/src/anchoring/reader.py · lines 33-42`

```
cached = await get_value(company_id, supplier_type, price_range)
if cached is not None:
    return cached

value = await get_latest_adjusted_value(
    db, company_id, supplier_type, price_range
)
if value is None:
    value = get_base_anchoring_value(price_range)

await set_value(..., value, nx=True)
return value
```

Redis 장애는 cache miss로 취급

`schedules/anchoring/src/anchoring/redis_client.py · lines 69-84`

```
try:
    raw = await _client.get(anchor_key(...))
    if raw is None:
        return None
    return int(raw)
except Exception as ex:
    _note_failure("get", anchor_key(...), ex)
    return None # 호출측이 DB fallback
```

두 겹의 회복 장치

TTL 7일은 오래된 키가 영원히 남는 것을 막습니다. 주간 reconciliation은 DB의 최신 조정값을 Redis에 다시 SET하여, DB commit 뒤 Redis 갱신에 실패했던 값도 교정합니다.

현재 견적 생성 경로의 예외

Negotdata의 실제 견적 생성은 Redis를 사용하지 않고 PostgreSQL의 `anchoring.current_values` View를 일괄 조회합니다. Redis는 현재 anchoring 배치 내부 캐시입니다.

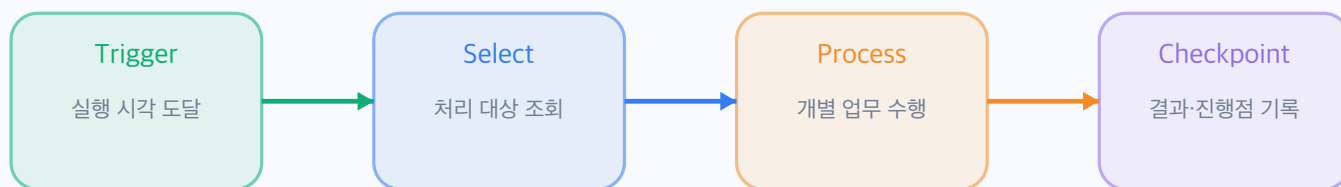
스케줄러·배치 — 개념부터

시간 규칙으로 작업을 시작하고, 많은 데이터를 사용자 요청 밖에서 처리하는 방식

둘의 차이

스케줄러는 '언제 실행할지'를 결정합니다. 배치 Job은 '무엇을 어떻게 처리할지'를 구현합니다. CronTrigger가 알람시계라면 close_expired_quotations는 알람이 울렸을 때 수행할 실제 업무입니다.

Job의 생명주기



운영에서 반드시 결정할 것

중복 실행

이전 Job이 안 끝났는데 다음 시각이 오면?

Misfire

서버가 꺼져 실행 시각을 놓쳤다면?

부분 실패

100건 중 73번째가 실패하면 어디부터 재개?

재시도

즉시 재시도, 다음 tick, 운영자 재처리 중 무엇?

멱등성

같은 대상을 다시 처리해도 중복 효과가 없는가?

관측성

처리량·실패 대상·소요시간을 로그와 지표로 남기는가?

스케줄러만으로 정확성은 보장되지 않음

max_instances=1은 한 프로세스 안의 중복을 막을 뿐입니다. 서버가 여러 대면 각 서버가 Job을 실행할 수 있으므로 DB 조건부 갱신, 분산 락, 전용 Worker 같은 추가 방어가 필요합니다.

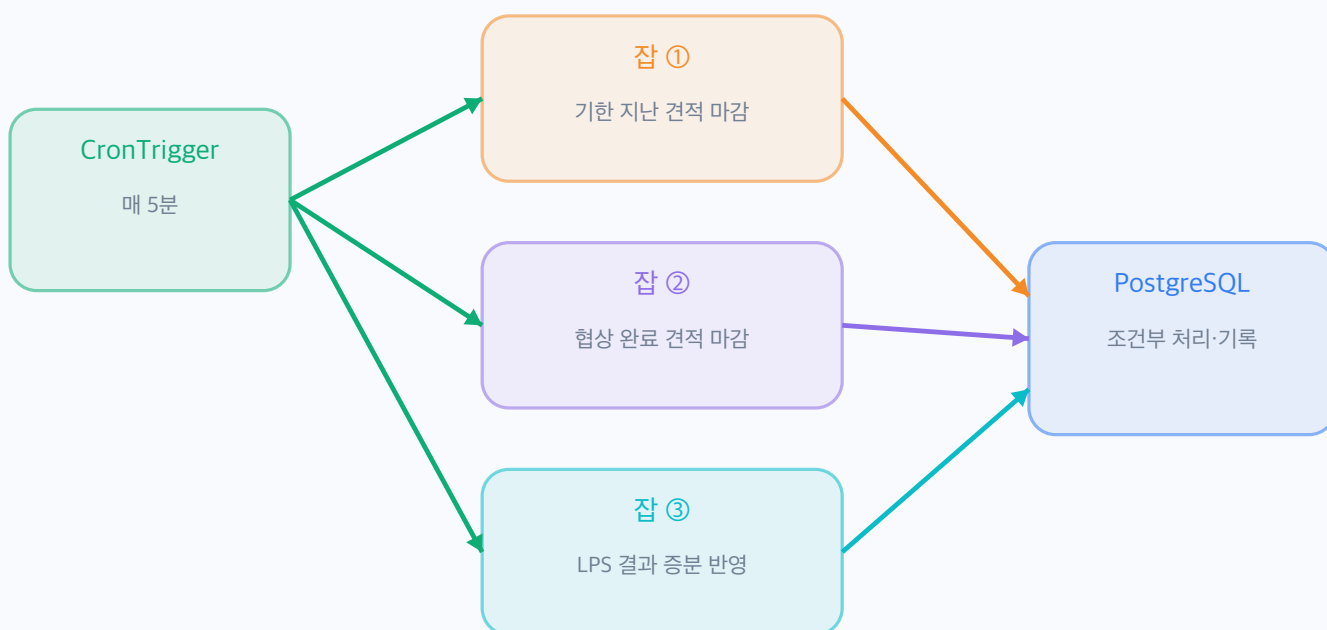
스케줄러와 배치 안정성

사용자 요청 없이 정해진 시간마다 반복 업무를 수행하는 백그라운드 실행

쉬운 비유

API가 손님이 주문할 때 움직이는 직원이라면, 스케줄러는 매 5분마다 마감 시간이 지난 주문을 확인하는 당직자입니다. 사람이 요청하지 않아도 시간이 되면 일을 시작합니다.

5분 tick의 세 가지 작업



중복 실행 방지 장치

SCHEDULER_ENABLED=1인 프로세스 하나만 잡을 등록합니다. 각 잡은 max_instances=1이고, 밀린 실행은 coalesce=True로 한번만 실행합니다. 그래도 다중 서버 가능성을 고려해 DB의 조건부 UPDATE가 마지막 방어선입니다.

안정 장치가 없으면

서버 Worker 수만큼 같은 잡이 실행되고, 같은 건적을 여러 번 마감하거나 알림을 중복 발송할 수 있습니다.

현재 방식

실행 프로세스 제한 + 잡 중복 제한 + DB 동시성 가드를 겹쳐 사용합니다.

스케줄러·배치 — 실제 코드

‘언제 실행할지’와 ‘무엇을 안전하게 처리할지’를 분리합니다.

APScheduler 등록: 5분, 중복 방지, 지연 허용

[negodata/backend/scheduler/_init_.py · lines 37-69](#)

```
_scheduler = AsyncIOScheduler(timezone="Asia/Seoul")
_scheduler.add_job(
    jobs.close_expired_quotations,
    CronTrigger(minute="*/5"),
    id="close_expired_quotations",
    coalesce=True,          # 밀린 실행은 1번만
    misfire_grace_time=600, # 10분 내 지연 실행 허용
    max_instances=1,        # 같은 잡 동시 실행 금지
)
```

Job: 대상 조회와 개별 마감 처리를 분리

[negodata/backend/scheduler/jobs.py · lines 39-61](#)

```
err_type, qt_ids = await DB_SESSION_MNG.execute_lambda(
    quotations.DBType(),
    DBWRType.DB_READ.value,
    lambda s: crud.list_due_for_close(s, now),
)
if err_type != ErrorType.SUCCESS:
    return 0

results = await _close_each(service, qt_ids)
return sum(results.values())
```

멱등성(idempotency)

같은 잡을 두 번 실행해도 최종 결과가 한 번 실행한 것과 같도록 만드는 성질입니다. 대상 조회가 중복될 수 있어도 `claim_for_close`의 조건부 UPDATE가 두 번째 처리를 `rowcount=0`으로 막습니다.

실패한 tick은 어떻게 되나요?

LPS 동기화는 watermark 기반 증분 처리라 실패한 회차의 데이터가 다음 5분 tick에서 다시 대상이 됩니다. 스케줄러 자체 재시도보다 데이터 설계를 통해 회복합니다.

다섯 기술이 한 장면에서 만나는 순간

예: 협상이 모두 끝난 건적을 스케줄러가 자동 마감하는 동안 담당자가 수동 마감을 클릭했다.



핵심 관점

어려운 백엔드 기술은 '라이브러리 이름'보다 경계와 실패를 다루는 방법입니다. 네트워크 경계, 회사 경계, 트랜잭션 경계, 캐시의 원본 경계를 명확하게 설계하는 것이 핵심입니다.

초보자를 위한 한 줄 사전

분산 시스템

여러 프로세스·서버가 네트워크로 협력하는 시스템

부분 실패

전체 중 일부 서비스만 실패한 상태

Timeout

응답을 무한히 기다리지 않고 정해진 시간에 포기하는 제한

Best-effort

실패해도 핵심 업무는 성공시키는 보조 작업 정책

트랜잭션

여러 DB 변경을 모두 성공 또는 모두 취소하는 작업 단위

Rollback

실패했을 때 트랜잭션의 변경을 되돌리는 것

Race condition

실행 순서에 따라 결과가 달라지는 동시성 문제

원자적 연산

중간 상태가 보이지 않도록 한 번에 처리되는 연산

멀티테넌시

한 시스템을 여러 고객사가 격리된 상태로 공유하는 구조

Cache-aside

캐시를 먼저 보고, miss이면 원본 조회 후 캐시를 채우는 패턴

TTL

캐시 값이 자동 만료되기까지의 시간

Stale

원본보다 오래되어 현재와 맞지 않는 캐시 상태

Invalidation

원본 변경 시 캐시를 삭제하거나 무효화하는 것

Reconciliation

원본과 복사본을 비교·재적재해 다시 맞추는 작업

Scheduler

정해진 시간 규칙에 따라 작업을 실행하는 도구

Batch

사용자 요청과 별개로 데이터 묶음을 주기적으로 처리하는 작업

멱등성

같은 작업을 반복해도 최종 결과가 달라지지 않는 성질

추천 복습 순서

트랜잭션·동시성 → 캐시 정합성 → 스케줄러 → 멀티테넌시 → 분산 시스템 순으로 다시 보면, 작은 DB 작업에서 전체 서비스 구조로 이해가 확장됩니다.