채기훈

Backend Developer

읽기 쉬운 코드와 검증 가능한 구조를 통해 팀이 빠르게 이해하고 안전하게
수정할 수 있는 시스템을 만드는 2년차 백엔드 개발자입니다.

chae0738@naver.com · 010-6278-8593

github.com/Hun425 · velog.io/@chae0738

Portfolio · 2026.03

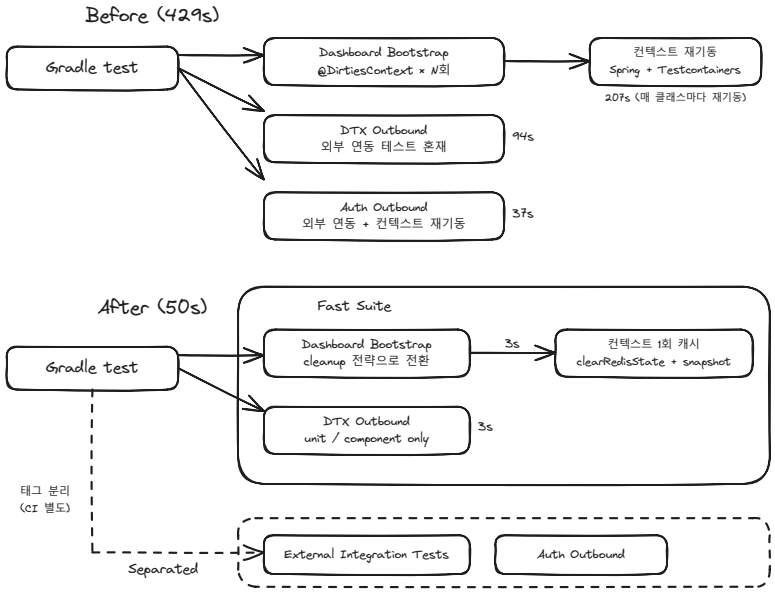
목차

테스트 구조 최적화를 통한 CI 피드백 루프 개선

비욘드메디슨 · Clickless · 2026.03

성능 개선 | Kotlin | Spring Boot | Testcontainers | H2 | Redis

아키텍처



기술 선택 이유

전체 테스트 실행 시간 429초 중 실제 테스트 로직 비용은 미미했고, **병목은 @DirtiesContext로 인한 Spring 컨텍스트 재기동**이었습니다. baseline 측정 스크립트를 만들어 모듈별 비용을 정량화한 후, 가장 비싼 모듈부터 순차 최적화하는 전략을 선택했습니다.

테스트 삭제나 assertion 약화가 아닌, **"같은 의미를 더 싸게 유지"**하는 것을 원칙으로 세웠습니다.

트러블슈팅

Dashboard Bootstrap: 207s → 3s

문제: 매 테스트 클래스마다 Spring Boot + Testcontainers가 재기동되어 207초 소요.

해결: cleanup 경로 구축. Redis는 clearRedisState(), DB는 snapshot restore. maxParallelForks=1로 캐시 보호.

DTX Outbound: 94s → 3s

문제: 외부 연동 테스트가 기본 test에 혼재.

해결: unit/component/bootstrap/external 4단계 분류, 외부 테스트는 전용 태스크로 분리.

성과

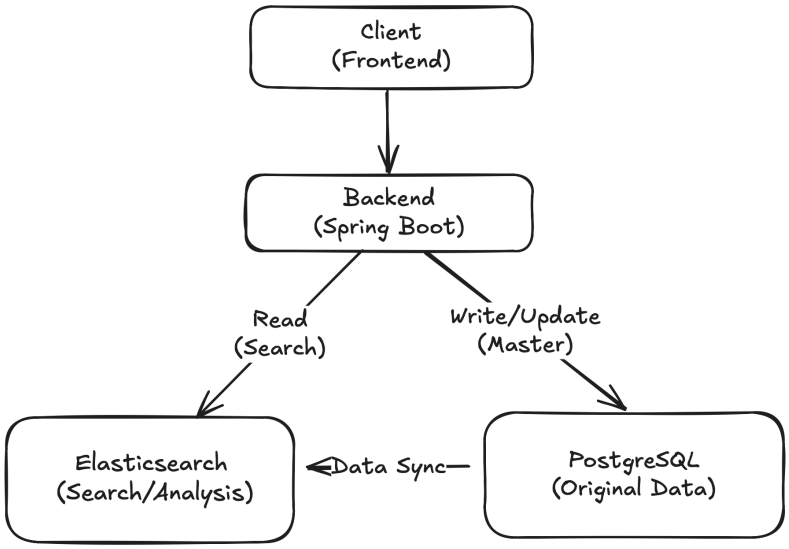
전체 테스트	Dashboard Bootstrap	DTX Outbound	Auth Outbound
~~429s~~ → 50s	~~207s~~ → 3s	~~94s~~ → 3s	~~37s~~ → 1s
-88%	-98%	-97%	-97%

Elasticsearch 도입을 통한 대용량 데이터 검색 성능 최적화

아이씨티웨이 · 대기정책 시스템 · 2024.12 - 2025.10

성능 개선 | Java | Spring Boot | Elasticsearch | PostgreSQL | Kafka

아키텍처



기술 선택 이유

약 2,260만 건 차량 데이터에서 `LIKE '%검색어%'`가 B-Tree 인덱스를 무력화, `OFFSET 999990`이 전체 데이터를 읽고 버리는 구조였습니다. GIN 인덱스 + FTS도 검토했으나 OFFSET 구조적 한계는 해결 불가. **역색인 기반으로 접근 방식을 전환**하기 위해 Elasticsearch를 선택하고, 원본은 PostgreSQL에 유지하는 하이브리드 구조를 설계했습니다.

트러블슈팅

2,260만 건 마이그레이션

문제: 한번에 이관 시 메모리 초과 및 타임아웃.

해결: 5만 건 청크 배치 이관. LIKE → Bool + Wildcard Query 전환.

Deep Pagination 제한

문제: max_result_window(10,000) 제한.

해결: 제한 해제 + 인덱스 설정 최적화.

성과

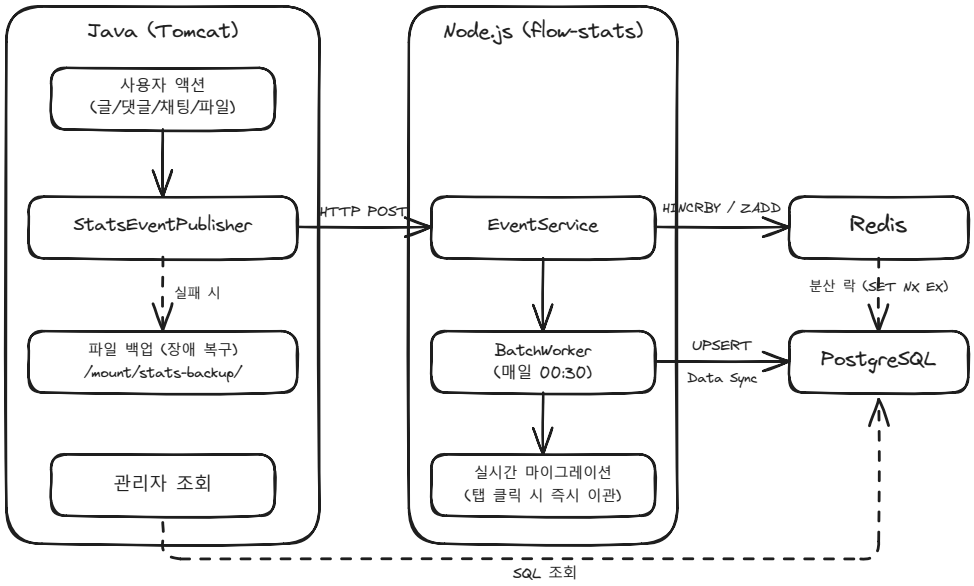
검색 응답 (마지막 페이지)	PostgreSQL 부하
~~20s~~ → **2s**	**70% 감소**
-90%	-70%

실시간 집계 파이프라인 설계 및 구축

마드라스체크 · 삼성화재 통계 고도화 · 2025.10 - 2026.03

아키텍처 설계 | Java | Node.js | Redis | PostgreSQL | Docker

아키텍처



기술 선택 이유

기존 통계 조회는 원본 테이블을 매번 조인, 데이터 누적 시 **20초 이상** 소요. 매 액션마다 DB INSERT 시 WAS 부하 급증.

Kafka/RabbitMQ는 **오버엔지니어링**으로 판단. 기존 Redis를 버퍼로, 별도 Node.js로 **WAS 부하 분리**하는 구조 선택.

Redis HINCRBY/ZADD NX 원자적 연산으로 동시성 해결, 키 패턴을 PK와 일치시켜 이관 비용 최소화.

트러블슈팅

실시간성 vs 정렬 충돌

문제: Redis만으로는 SQL ORDER BY 정렬 불가.

해결: 배치 + 실시간 마이그레이션 이중 구조. 탭 클릭 시 Redis → PostgreSQL 즉시 이관 후 SQL 조회.

동시 마이그레이션 중복

문제: 동시 탭 클릭 시 중복 집계.

해결: Redis SET NX EX 분산 락.

전송 실패 시 유실

문제: HTTP 전송 실패 시 이벤트 유실.

해결: 로컬 파일 백업 → 자정 배치 복구.

성과

통계 조회

~~20s~~ → **1s 이내**

-95%

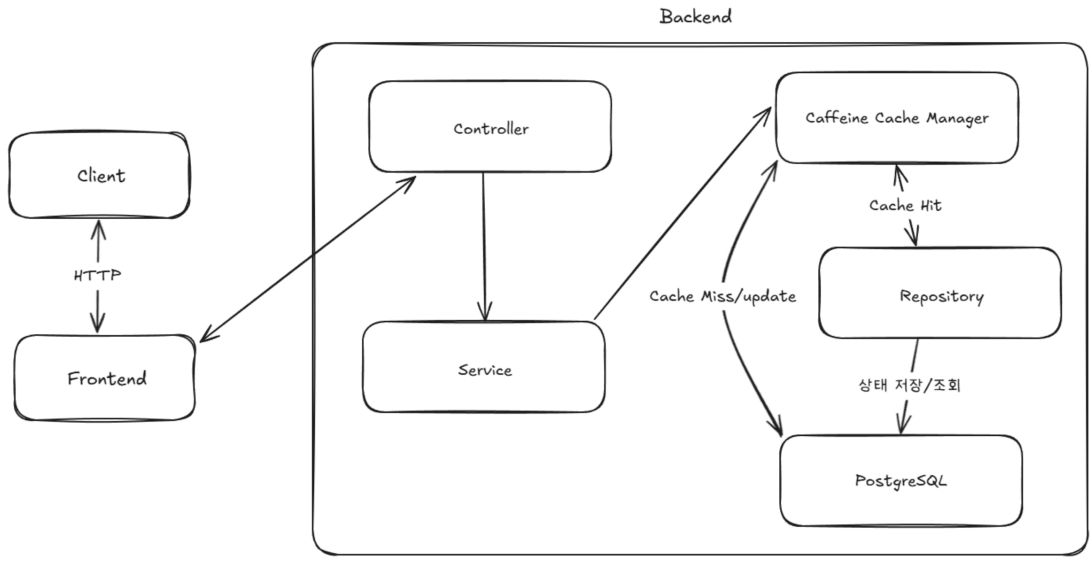
기존 원본 테이블 조인 방식에서 사전 집계 테이블 조회로 전환하여 통계 페이지 응답 시간 대폭 단축. WAS와 집계 로직을 완전 분리하여 기존 서비스 안정성 유지. 장애 시에도 파일 백업 → 배치 복구 경로로 데이터 정합성 보장.

Caffeine 캐시 적용을 통한 퀴즈 API 성능 최적화

팀 프로젝트 · Cram · 2025.03

성능 개선 | Spring Boot | Caffeine | Redis | k6

아키텍처



기술 선택 이유

퀴즈 조회 API가 반복적으로 DB 접근하여 성능 저하. 인기 퀴즈는 **조회 빈도 높고 변경 빈도 낮은** 특성.

Redis는 세션 관리에 사용 중, 조회 전용 데이터에 네트워크 hop 불필요. **로컬 캐시(Caffeine)**가 적합하다고 판단.

@CacheEvict로 캐시 무효화, k6로 전후 정량 검증.

트러블슈팅

캐시 일관성 관리

문제: 데이터 수정/삭제 시 캐시에 이전 데이터 잔존.

해결: @Cacheable로 캐싱, 변경 시 @CacheEvict로 즉시 무효화.

성과

API 응답 속도

70% 단축

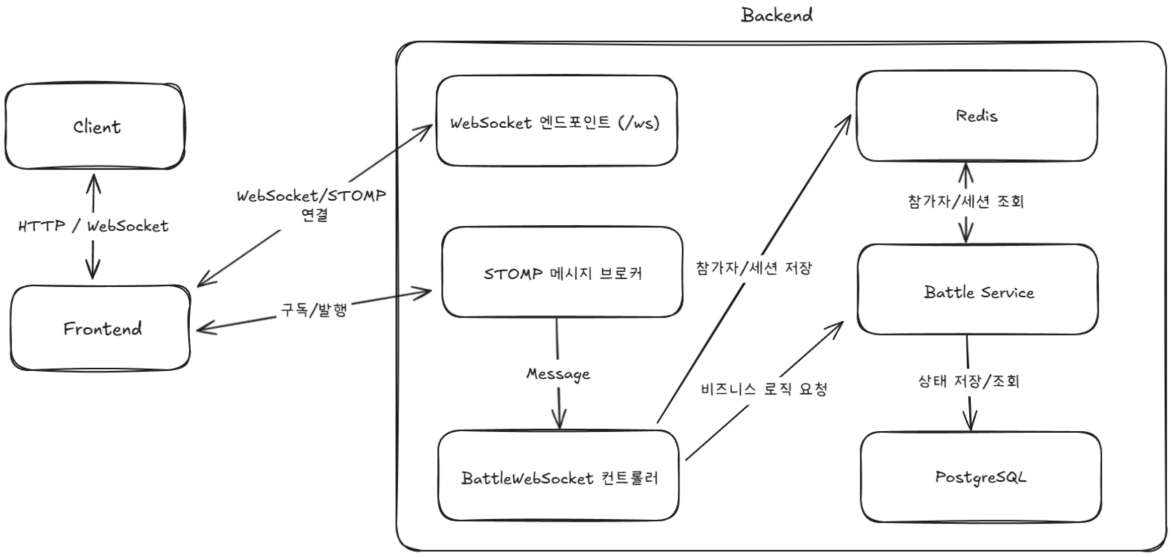
-70%

Spring WebSocket을 활용한 실시간 퀴즈 배틀 기능 구현

팀 프로젝트 · Cram · 2025.03

실시간 설계 | Spring Boot | WebSocket | STOMP | Redis

아키텍처



기술 선택 이유

사용자 간 실시간 상호작용(배틀 참가, 정답 제출, 점수 변화) 동기화 필요.
HTTP 폴링은 **트래픽 낭비 + 실시간성 한계**.

SSE는 양방향 불가. **WebSocket**을 선택하고 STOMP 프로토콜로 배틀방 단위 구독/발행 패턴 구현.

트러블슈팅

동시 정답 제출 race condition

문제: 동시 정답 제출 시 점수/순위 계산 오류.

해결: @Synchronized + BattleRoomStatus Enum 관리. Redis 분산락 확장 가능 설계.

배틀방 메시지 격리

문제: 전체 브로드캐스팅으로 다른 방에도 메시지 전달.

해결: /sub/battle/room/{roomId}로 채널 분리.

성과

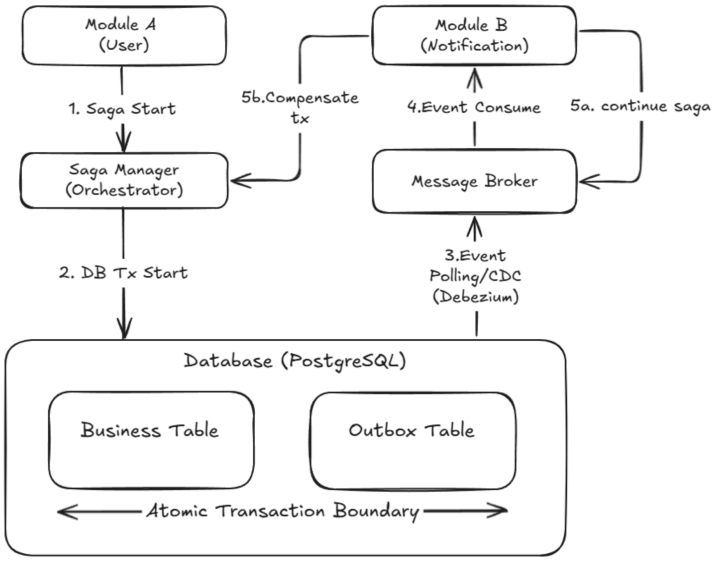
실시간 퀴즈 배틀 기능 구현 완료. 폴링 대비 서버/네트워크 부하 감소 및 실시간성 보장. WebSocket + STOMP 기반 메시징 아키텍처 구축 경험 확보.

Saga 및 Outbox 패턴 기반의 분산 트랜잭션 구현

개인 프로젝트 · AlgoReport · 2025.04

아키텍처 설계 | Kotlin | Spring Boot | Kafka | PostgreSQL | Debezium

아키텍처



기술 선택 이유

여러 모듈(가입 → 프로필 → 알림)에서 단일 ACID 트랜잭션으로 정합성 보장 불가.

2PC는 단일 장애점 + 가용성 저하로 제외. **Saga(Orchestration)**로 단계별 실행 + 보상 트랜잭션으로 최종 일관성 보장.

Outbox + Debezium으로 트랜잭션 커밋과 이벤트 발행의 원자성 확보. At-Least-Once 보장.

성과

특정 모듈 장애가 전체 시스템의 데이터 불일치로 확산되는 것을 방지하여 회복탄력성 향상. 모듈 간 결합도를 낮추어 유지보수성 및 확장성 확보.