

Portfolio

Intro

안녕하세요, 박성욱입니다.

1995. 03.10

respectwo@naver.com / 01027524565 / [Github](#)



확장성이 뛰어난 시스템을 만들고 싶습니다.

단순히 동작하는 서비스가 아니라, 변화에 유연하게 대응할 수 있는 확장 가능한 시스템을 만들고자 합니다.

이를 위해 서비스 구조를 깊이 있게 이해하고, 유지보수성과 확장성까지 고려한 개발을 지향합니다.

현실적으로 최선의 선택을 할 수 있는 개발자입니다.

이상적인 기술 선택보다 현재 상황과 제약 속에서 가장 효과적인 해결책을 찾는 것이 더 중요하다고 생각합니다.

환경, 인프라, 팀 역량 등을 종합적으로 고려해 안정성과 효율 사이에서 균형 잡힌 개발을 추구합니다.

희망 직무

백엔드&서버 엔지니어

주요 기술



Java

하 중 상

- 객체 지향 프로그래밍, 디자인 패턴을 이해하고 적용하는 서버 사이드 어플리케이션 개발이 가능합니다.



Spring Boot

하 중 상

- Spring Boot 기반의 웹 어플리케이션 아키텍처를 설계하고 REST API 및 비즈니스 로직을 구현할 수 있습니다.



Database

하 중 상

- 관계형/비관계형 DB를 사용해 본 경험과 효율적인 쿼리 작성, 데이터 모델링에 대해 고민한 경험을 가지고 있습니다.



Linux

하 중 상

- 리눅스 환경에서 어플리케이션을 배포하고 서비스 운영을 위한 시스템 관리가 가능합니다.

경력

삼성 청년 소프트웨어 아카데미(SSAFY)

2024. 07 ~ (진행 중)

국민은행 IT's your Life

2023. 03 - 2023. 05 (3개월)

학력

동아대학교

2014 - 2022 (경제학과)

자격증

정보처리기사

2024.12 (한국산업인력공단)

SQLD(SQL 개발자)

2023.12 (한국데이터산업진흥원)



실시간 CS 퀴즈 배틀 서비스

[프로젝트 Github](#)

[소개 영상](#)

작업 기간

2025. 04 - 2025. 05 (5주)

사용 기술 스택

Spring Boot Java Docker
 Nginx MySQL MongoDB
 Redis Kafka

역할

Backend

게임 내 ELO 랭킹 시스템 구축

Kafka 메시지 발행 및 수신 구조 구축

Infra

MongoDB 기반 랭킹 스냅샷 저장 구조 설계

Gitlab CI/CD 기반 선택적 서비스 파이프라인 구축

주요 기능 및 개요

사용자가 실시간으로 퀴즈에 참여해 점수를 겨루는 **멀티플레이 게임 서비스**입니다.

게임은 빠른 응답과 정답 여부에 따라 점수가 결정됩니다.

게임 종료 후에는 실시간으로 갱신된 랭킹과 변화를 함께 제공됩니다. 해당 게임을 통해 경쟁과 성장의 재미를 동시에 경험할 수 있습니다.

구성원

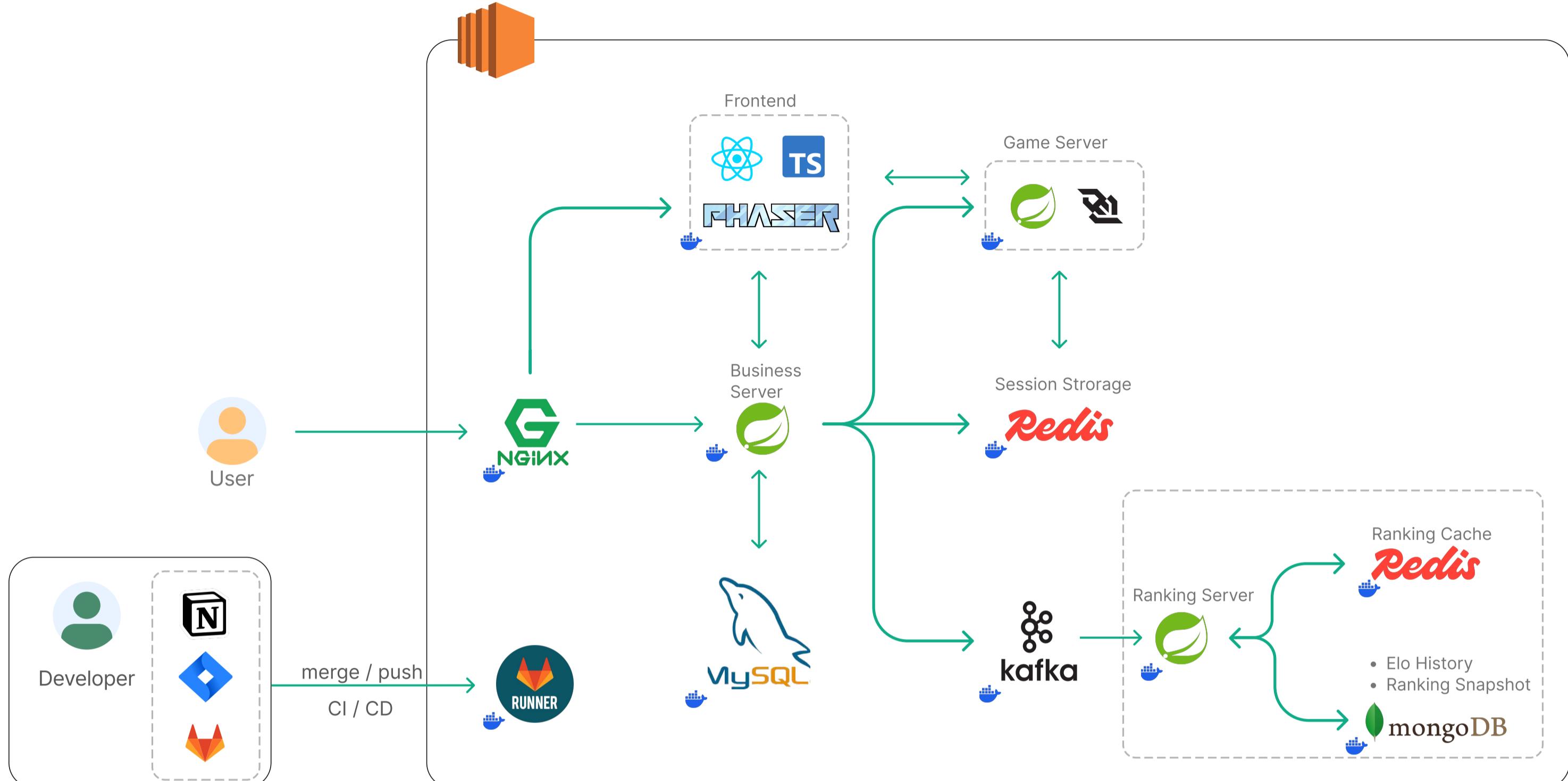
BE (박성욱 외 2인)

FE (3인)

프로젝트 회고

- 실시간성과 구조적 분리를 동시에 고려해 첫 **MSA 아키텍처**를 직접 설계하고 구현하는 경험을 했습니다.
- MySQL, Redis, MongoDB** 등 각 데이터베이스의 특성을 고려해 설계하고 활용하며, **데이터 처리 구조**에 대한 이해할 수 있었습니다.
- Kafka 메시지 공통 DTO를 GitHub Packages로 배포하고, 외부 모듈로 주입하여 **서비스 간 일관된 메시지 형식**을 유지했습니다. 이를 통해 협업 과정에서의 의존성 관리와 공통 DTO 버전 관리의 중요성을 배웠습니다.

시스템 아키텍쳐



게임 - 비즈니스 - 랭킹 분리 MSA 아키텍쳐 구성

짧은 개발 기간과 제한된 리소스를 고려해 현실적인 기능 분리를 기준으로 한 **MSA 아키텍처**를 설계하였고 게임 / 비즈니스 / 랭킹의 3개 서비스를 중심으로 구조를 구성했습니다.

- **비즈니스 서버**
 - 사용자 정보 저장, RDB 연동 등 **핵심 도메인** 로직을 처리
 - 게임의 실시간 통신은 대부분 게임 서버에서 처리되므로 비즈니스 서버는 상대적으로 **저부하 중심의 역할**을 담당
- **게임 서버**
 - 실시간 퀴즈 진행, 클라이언트와의 WebSocket 통신 등 **속도와 반응성이 중요한 기능**을 전담
- **랭킹 서버**
 - 실시간 랭킹 조회와 ELO 점수 계산 등 **연산량이 높고 트래픽이 집중되는 기능**을 분리
 - Kafka를 통해 비동기적으로 메시지를 수신하고 Redis에 반영하며 장애 격리성과 확장성 확보

초기에는 랭킹 스냅샷 기능을 별도 로그 서버로 구성하는 방안도 검토했지만 추가 가공이 필요하지 않다는 점과 운영 리소스를 고려해 랭킹 서버 내부 기능으로 통합하여 처리했습니다.

이처럼 필요한 기능만을 적절히 분리한 실용적 MSA 구성을 통해 **확장성과 유지보수성이 뛰어난 안정적인 아키텍처**를 완성할 수 있었습니다.

랭킹 시스템 설계하기

나의 랭킹				
16	나나혜원	1469	— 0	
1	이상해찌	1668	▲ 5	
2	꼬부기	1638	▼ 1	
3	팬텀	1625	▲ 5	
4	옥스리브로도	1624	▼ 2	
5	파블로피카츄	1597	▼ 2	
6	나몰빼미	1586	▲ 3	
7	모네덕후	1568	▼ 3	
8	괴력몬	1559	▼ 3	
9	잠만보	1538	▼ 2	
10	영가	1495	— 0	

BITEBYTE의 랭킹 시스템

ZMSCORE rank:elo		
	value	score
1	member:23	1300
2	member:1	1328
3	member:41	1338
4	member:32	1353
5	member:25	1387
6	member:33	1425
7	member:4	1442
8	member:11	1469
9	member:62	1469

Redis의 Sorted Set

기능 구현에 앞서 **요구사항을 정확히 파악하는 것이 가장 중요하다는 점**을 이번 랭킹 시스템 설계를 통해 체감할 수 있었습니다.
초기 설계단계에는 랭킹 시스템을 구현하는데 아래 옵션을 고려하였습니다.

- A) 상위 N명의 유저의 순위만 제공하는 구조
- B) 전체 유저 점수를 캐싱해 전체 유저의 순위를 확인할 수 있는 구조
- C) 배치 처리 기반으로 주기적으로 순위를 계산해 제공하는 구조

처음에는 메모리 효율을 위해 A안을 선택했고, Redis Sorted Set에서 N번째 점수를 기준으로 새로운 점수가 포함될지 판단하는 구조로 설계했습니다.

그러나 이후 “내 순위 변동을 실시간으로 확인할 수 있어야 한다”는 요구사항이 새롭게 생기면서,
전체 유저의 점수를 Redis에 캐싱하고 ZRANK 명령어로 순위를 조회하는 B안으로 방향을 전환하게 되었습니다.

랭킹 시스템 파이프라인 구축

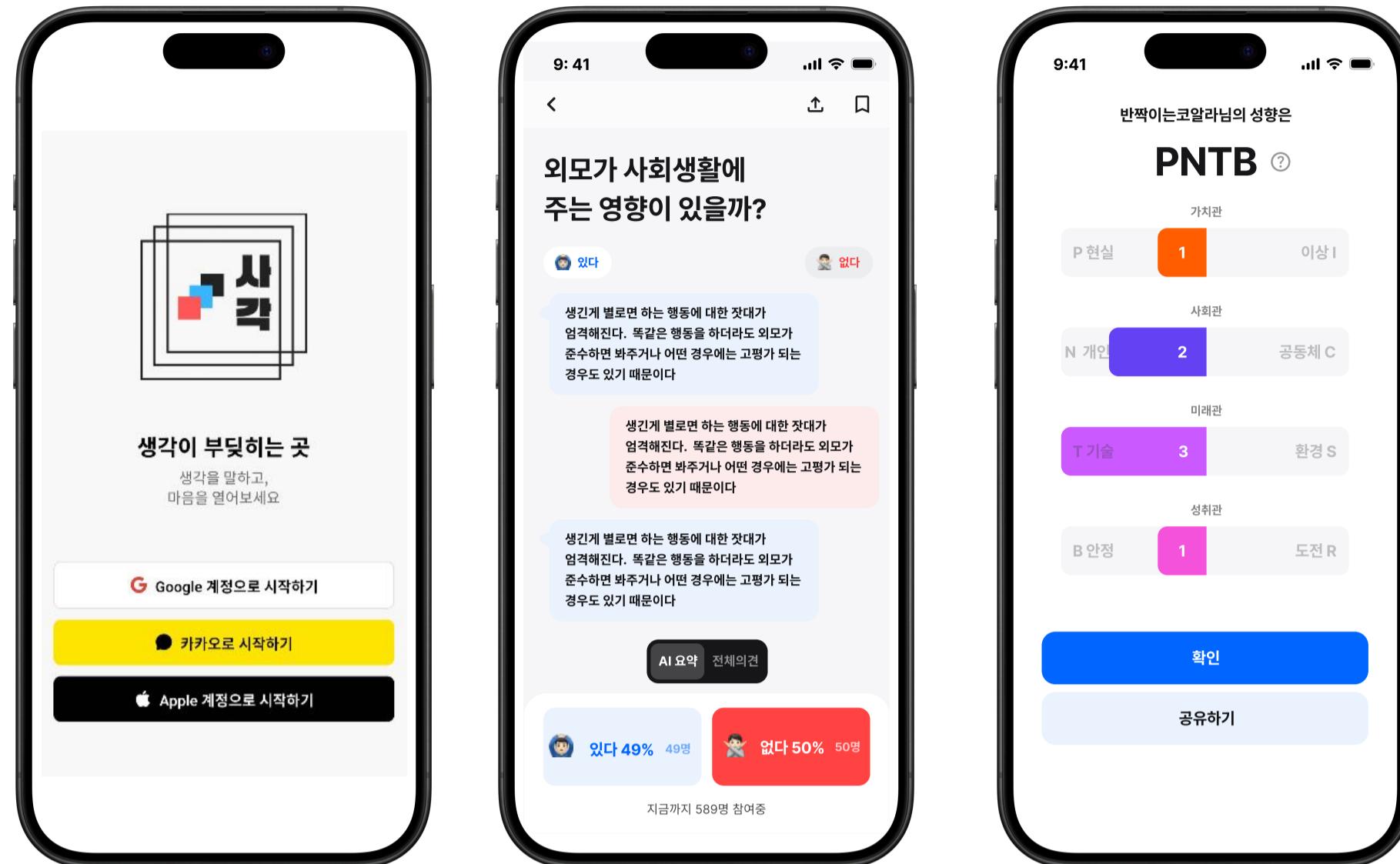
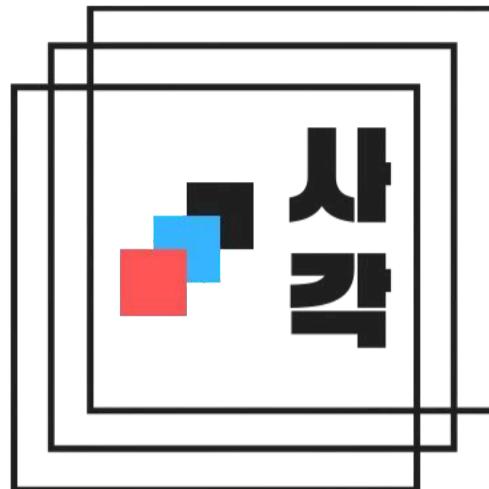


실시간 퀴즈의 결과를 기반으로 모든 유저의 실시간 순위를 파악할 수 있는 **랭킹 시스템 파이프라인**을 구축했습니다.

소켓 서버에서 수신한 게임 결과를 API 서버가 **Kafka**로 발행하고,
랭킹 서버가 이를 수신하여 ELO 알고리즘으로 점수를 계산하고 모든 유저의 점수를 **Redis의 Sorted Set**으로
저장해 빠르게 랭킹을 조회할 수 있게 만들었습니다.

또한 **특정 날짜 대비 순위 변화 추이**를 확인해야 하는 요구사항이 있었기 때문에

Redis의 실시간 데이터를 하루 단위로 **MongoDB**에 스냅샷 저장하는 **스케줄링** 구조를 도입하여
사용자별 랭킹 변화 기록을 안정적으로 추적할 수 있는 구조를 완성했습니다.



모바일 앱 기반 논쟁 플랫폼

프로젝트 Github

작업 기간

2025. 02 - 2025. 04 (7주)

사용 기술 스택

Spring Boot Java Docker
 Nginx MySQL AWS RDS
 Redis AWS S3

역할

Backend

토론 도메인 설계 및 기능 구현

S3 Presigned URL 기반 이미지 업로드 기능 구현

GCP 기반 Firebase 인증 구현

Infra

Docker 환경 인프라 아키텍처 설계

Expo 안드로이드 빌드 및 배포

Jenkins CI / CD 구축

주요 기능 및 개요

사용자가 찬반 의견을 선택하고, 자신의 주장을 펼치며 토론에 참여하는 참여형 토론 플랫폼입니다.

성향 테스트 기반의 개인화된 자유게시판, 실시간 발언 토론방, 투표 결과에 따른 통계 제공, 그리고 AI 요약 기능을 통해 사용자는 다양한 시각을 접하고 자신의 관점을 확장할 수 있습니다.

구성원

BE (박성욱 외 2인)

FE (3인)

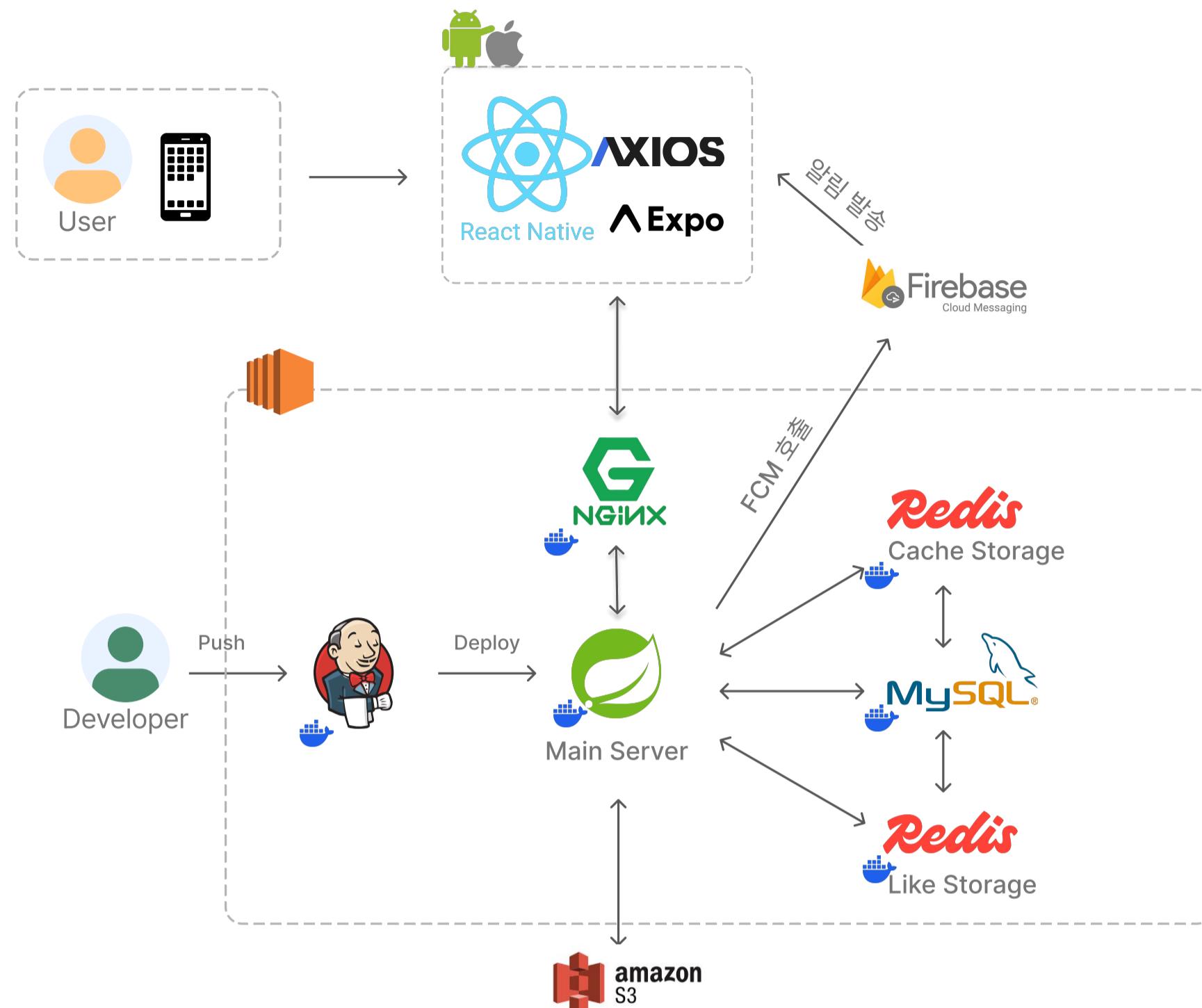
프로젝트 회고

- 모바일 앱 기반 첫 프로젝트로 클라이언트-서버 간 구조적 연결에 대한 이해 확장했습니다.
- 좋아요순 / 최신순 / 댓글순 등 다양한 정렬 조건을 지원하기 위해, **QueryDSL** 기반 복합 페이지네이션 쿼리를 구현했습니다.
- React Native와 Expo Native 기능 간의 존속 충돌로 인해 안드로이드 빌드 환경에서 많은 이슈를 겪었으며, 이를 해결하며 **네이티브 모듈에 대한 이해**를 심화했습니다.
- EC2 내 MySQL를 프로젝트 종료 이후 AWS RDS로 **マイグレーション**하는 경험을 할 수 있었습니다.

Square

Project - 2

시스템 아키텍쳐



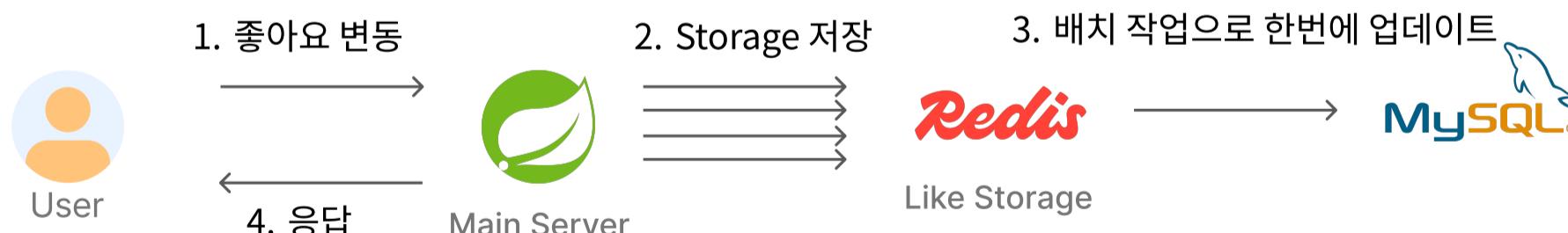
좋아요 수 저장 전략

아키텍처 설계 과정에서 게시글의 좋아요 수를 어떻게 저장할지에 대한 팀 내 논의가 있었습니다.

초기에는 게시글 정보와 함께 좋아요 수를 RDB에 저장하기로 했지만 좋아요나 조회수처럼 변동이 잦은 데이터는 고빈도 쓰기 작업으로 인한 성능 저하 가능성이 제기되었습니다.

이에 가변 데이터를 처리하는 별도의 Redis를 운용해 **Write-back 전략**을 이용하였습니다.

저장



조회

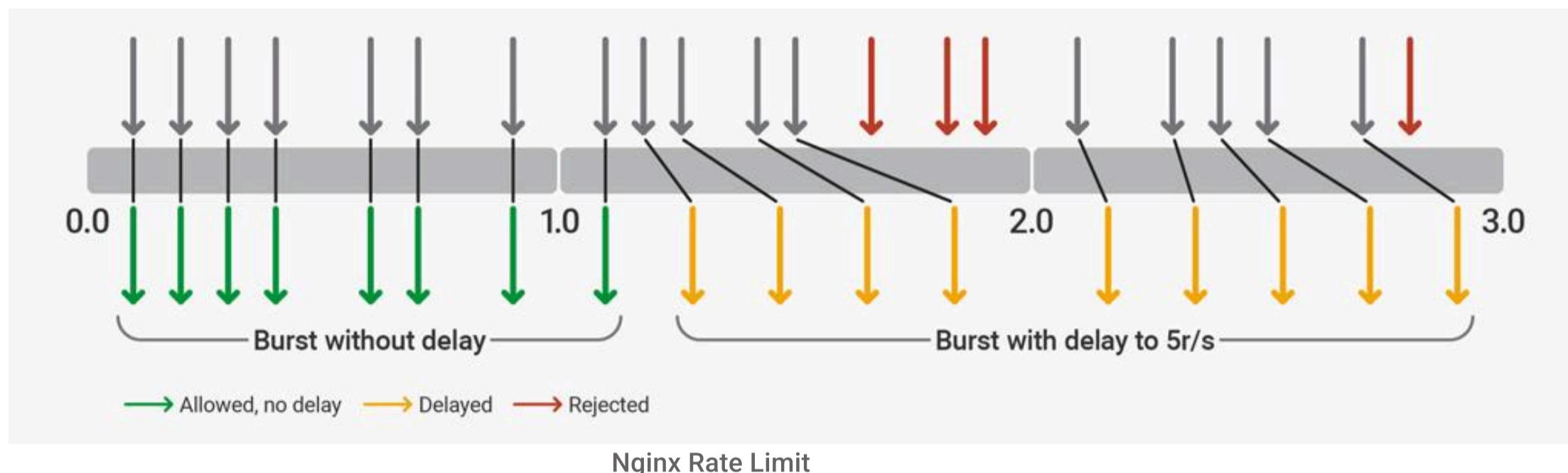


너무 많은 좋아요 요청 제어하기

설계 단계 외에도 로직적으로 많은 요청을 제어할 수 있는 방법이 필요했습니다.

클라이언트 단에서 처리하는 것 외에 인프라, 서버 두 가지 레이어에서 아래 방안을 적용하였습니다.

Infra Layer



```
limit_req_zone $binary_remote_addr zone=like_api_zone:10m rate=1r/s;
```

Nginx Limit Zone 전역 설정

```
location /api/likes {
    limit_req zone=like_api_zone burst=5 nodelay;
```

좋아요 API 엔드포인트에 적용

API Gateway 역할을 수행하는 Nginx에서 rate=1r/s (초당 1회)를 허용하는 Limit 영역을 정의하고 좋아요를 처리하는 API의 엔드포인트에 적용하였습니다.

초당 1회의 요청을 허용하지만, burst=5 옵션을 통해 **순간적으로 최대 5개의 요청까지 허용하여 사용자 인터랙션을 자연스럽게 처리하고,** 비정상적으로 과도한 요청은 거부하여 **서비스 안정성을 유지되게 설정하였습니다.**

Server Layer

```
String lockKey = String.format("Like:rate_limit:%d:%d", user.getId(), targetType.name(), targetId);
Boolean isAllowed = batchRedisTemplate.opsForValue().setIfAbsent(lockKey, value: "1", timeout: 2, TimeUnit.SECONDS);

if (!Boolean.TRUE.equals(isAllowed)) {
    throw new CustomException(ExceptionCode.TOO_MANY_REQUEST);
}
```

좋아요를 처리하는 toggleLikeInRedis 메서드

좋아요를 처리하는 메서드 내의 핵심 로직을 수행하기 전에

lockKey라는 Redis 키를 먼저 생성합니다.

이 키에는 유저가 어떤 타겟에 좋아요 요청을 보냈는지 정보가 저장되며,

isAllowed 플래그를 통해 동일한 요청이 2초 이내에 발생했는지 여부를 판별합니다.

만약 중복 요청이라면 CustomException을 발생시켜 서버 레이어에서도 안정적으로 중복 요청을 차단하였습니다.

QueryDSL 기반 페이지네이션 구현

```

@Override 1 usage 1 sungwook *
public List<Opinion> findOpinionsByLikes(Long debateId, boolean isLeft, Long nextCursorId, I
    QOpinion op = QOpinion.opinion;
    BooleanBuilder builder = new BooleanBuilder();
    builder.and(op.debate.id.eq(debateId));
    builder.and(op.left.eq(isLeft));
    builder.and(op.valid.isTrue());

    if (nextCursorLikes != null && nextCursorId != null) {
        builder.and(
            op.likeCount.lt(nextCursorLikes)
            .or(op.likeCount.eq(nextCursorLikes).and(op.id.lt(nextCursorId)))
        );
    }
    return queryFactory
        .selectFrom(op)
        .join(op.user).fetchJoin()
        .where(builder)
        .orderBy(op.likeCount.desc(), op.id.desc())
        .limit(limit)
        .fetch();
}

```

좋아요 순 조회를 수행하는 QueryDSL 코드

정렬 기준이 다양한 토론글 목록 페이지에서, 정렬 안정성과 성능을 모두 만족하는 페이지네이션을 구현하기 위해 **QueryDSL**을 사용했습니다.

기존 offset 기반 페이지네이션은 동점 항목 간의 순서가 요청마다 바뀌어 데이터가 누락 또는 중복으로 조회되는 문제가 발생할 수 있었습니다.

이를 해결하기 위해 likeCount와 id를 함께 정렬 기준으로 사용하는 커서 기반 페이지네이션을 도입하고, 정렬 조건에 따라 동적으로 쿼리를 구성하기 위해 **BooleanBuilder**를 활용했습니다.

N+1 이슈 해결하기

 반짝이는코알라 PNTB

생긴게 별로면 하는 행동에 대한 잣대가
엄격해진다. 똑같은 행동을 하더라도 외모가
준수하면 봐주거나 어떤 경우에는 고평가는 되는
경우도 있기 때문이다

♡ 1,203

5분 전

서비스 내의 토론 댓글 예시

서비스 내에서 토론글 목록을 조회할 때 각 게시글에 작성자의 닉네임과 프로필 이미지가 함께 노출되어야 했습니다.

그래서 게시글과 연관된 작성자 정보를 함께 조회해야 했고, **JPA의 Lazy Loading** 설정으로 인해 게시글 수만큼 작성자 정보를 조회하는 **N+1 쿼리 이슈**가 발생했습니다.

이를 해결하기 위해 QueryDSL을 사용할 때 **fetchJoin**을 적용하여 게시글과 작성자 정보를 하나의 쿼리로 함께 조회하도록 최적화했습니다.
결과적으로 쿼리 수를 줄여 **DB 부하를 낮추고 API 응답 속도** 또한 개선할 수 있었습니다.

```
-- 게시글 목록 조회 (1회)
SELECT * FROM opinion WHERE debate_id = 1;

-- 게시글 수만큼 반복 (10개 게시글이면 아래 쿼리 10번)
SELECT * FROM user WHERE user_id = ?;
```

Lazy Loading의 쿼리문

```
-- 게시글 + 작성자 정보를 한 번에 가져옴
SELECT o.*, u.*
FROM opinion o
    JOIN user u ON o.user_id = u.id
WHERE o.debate_id = 1;
```

fetchJoin의 쿼리문