안녕하세요. 김승진입니다.
Backend Engineer

Phone.
Email. ohksj77@naver.com
GitHub. github.com/ohksj77
Blog. ohksj77.tistory.com

Introduce

안정적인 데이터 처리를 고민하고 설계하는 백엔드 엔지니어입니다.

- 대용량의 데이터를 인프라 변경 없이 처리하기 위해 API 서버가 자체 부하 분산하도록 설계해 부하를 크게 줄였습니다.
- 실시간 시스템을 성능 테스트와 함께 설계하여 하나의 인스턴스로 1000명의 사용자를 수용함을 검증했습니다.

어려움을 극복하는데 적극적으로 기여하고자 합니다.

- 오픈소스의 문제를 직접 해결하였으며 동아리의 행사를 주도적으로 활성화하는 등 적극적으로 문제를 해결해 왔습니다. 주도적으로 문제를 정의하며 마주한 어려움을 헤쳐나가고 함께 목표를 확실히 이루고 싶습니다.

Skill

Backend: Java, Spring, JPA

DB: MySQL, MongoDB, Redis

Infra: Docker, AWS (EC2, RDS), RabbitMQ

Career

* 파란 글꼴은 블로그 포스트로 연결됩니다.

EA Korea FC Online / Server Software Engineer 인턴 [2025.01 ~ 2025.02]

주요 사용 기술 : Node.js, Redis, MongoDB, RabbitMQ, Docker

어드민 서버 개발로 테스터의 생산성에 기여했으며 사내에 없던 새로운 서버 부하 분산 시스템과 알고리즘을 설계했습니다.

아이템 확률 검증을 위한 대량 개봉 프로젝트

* 아이템 개봉이란 아이템을 사용해 정해진 확률을 기반으로 랜덤한 결과(선수, 재화 등)를 계정에 획득하는 것을 의미합니다.

1. **대용량 데이터 처리로 발생하는 부하를 RPC 기반 API 서버 자체 로드밸런싱으로 해결**
   - **요구사항**
     - 실제 확률을 검증하기 위해 어드민에서 실 서비스에서 사용되는 메서드를 호출해 100만 건을 개봉
   - **로드밸런싱 시스템 설계 시도의 배경: 서버 자원 사용률 이슈와 인프라 수정의 제약**
     - 한 번에 모두 처리하는 방법은 OOM이 발생하며 chunk 단위 처리는 하드웨어 자원 사용률 이슈 발생
       - Memory 평균: 60% / CPU 평균: 100%
     - API Gateway의 어드민 API 서버로의 요청은 MQ 폴링 구조와 prefetch 설정에 의해 균등한 분배 불가
   - **로드밸런싱 알고리즘 설계**
     - 부하를 고려해 최근에 처리한 2개의 서버는 처리하지 않도록 서버마다 식별자를 부여해 구현
   - **순차 처리가 불가피하여 RPC 선택 및 로드밸런싱 흐름 설계**
     - 하나의 어드민 API 서버에서 RPC로 반복 요청하며 어드민 API 서버들은 RPC 큐에 요청 폴링 수행
     - 요청은 하나의 서버가 수신하여 구현해둔 자체 알고리즘으로 작업 처리 여부를 판단해 chunk 단위 개봉

- **구현 이후 테스트 결과**
  - 다중 서버에서 100만 건이 안정적으로 9분 내에 처리됨
    - Memory 평균: 20% / CPU 평균: 45%

프로젝트 성과

- 어드민에 대량 아이템 개봉 기능을 추가하여 게임 클라이언트에 직접 접속하여 10건씩 개봉하던 불편함을 해소

Project

이길저길 길치들을 위한 경로 제공 및 만남 관리 서비스

GitHub 저장소: github.com/HongDam-org/TWTW

주요 사용 기술 : Java, Spring, JPA, Redis, MySQL, RabbitMQ, Docker, GitHub Actions

백엔드(70%), 인프라(100%): [위치 공유, 알림, 친구, 약속장소] 도메인 / [OpenAPI, 알림, 데드레터] 공통 기능 / AWS 배포

1. **실시간 양방향 위치 공유 시스템 설계 및 지속적으로 갱신되는 위치 좌표 저장 고민**
   - **실시간 통신 기술 선택: Websocket, API 폴링, SSE 비교 분석**
     - [실시간성 / 양방향 지원 / 업데이트 빈도 적합성(각 다른 시점, 3초 주기)] 로 요구사항 기반 기준 비교
     - 위 비교와 **성능 테스트 수행 결과**의 하드웨어 자원 사용률, 처리량, 커넥션 안전성 기반으로 STOMP 선택
   - **queue 활용 전략 고민과 Message Broker 선택**
     - 사용자 그룹당 평균 1.3/s 의 요청이 오기에 여러 개의 queue로 분산하는 것이 좋을 것이라 판단
     - 그룹별로만 데이터를 공유하기에 그룹별로 Binding 하며 커넥션 기반 동적 queue 생성/삭제 고려
     - 옵션으로 커넥션 기반 동적인 queue 관리가 가능하며 동적 Binding 확장 가능한 RabbitMQ 선택
   - **지속적으로 갱신되는 위치 좌표 저장 고민과 Redis Geo 선택**
     - O(log(n)) 의 거리 연산이 가능한 Redis Geo를 DB 활용에 비해 성능적 개선을 기대하며 선택
   - **목표 tps 검증**
     - 모든 API와 함께 테스트하여 EC2 1대로 1000명의 사용자를 수용할 수 있는 환경을 구축함

2. **Like 쿼리 개선을 위한 n-gram parser 기반 Full Text 인덱스 도입**
   - **구현 초반 사용한 Like 쿼리와 문제점**
     - 닉네임의 Like 기반 검색 쿼리가 500만 row의 데이터 기준으로 평균 2.63s 소요
     - 문자열 검색을 지원하는 MySQL의 Full Text 인덱스로 Like 기반 쿼리를 대체하고자 함
   - **match against 쿼리 캐시 에러 발생**
     - n-gram parser와 함께 boolean mode로 match 연산 시 쿼리 캐시 공간 부족 에러 발생
   - **첫 번째 시도: 닉네임 길이 제한**
     - 쿼리시 고려할 경우의 수를 줄이고자 닉네임 최대 길이 8로 제한하니 29s 이상 걸리는 롱 쿼리 발생
     - query profiling 확인하여 FULLTEXT initialization 과정에서 29s 소요됨을 확인
   - **두 번째 시도: n-gram parser 특징을 고려한 쿼리 수정**
     - [ngram_token_size = 2] 임을 확인하며 검색어의 길이가 2보다 긴 것이 원인이라 추측
     - 검색어 string을 길이 2씩 분할하여 쿼리하도록 수정해 평균 0.53s로 개선

3. **OpenAPI RateLimit 이슈 대응과 호출 비용 절감을 위한 캐시 적용 및 전략**
   - **불필요한 OpenAPI 호출 고민과 캐싱 도입**
     - 데이터가 자주 변경되지 않는 OpenAPI를 지속적으로 호출, 간헐적으로 RateLimit 이슈 발생
     - OpenAPI 응답 데이터를 캐싱하여 Cache Aside 전략으로 OpenAPI 호출을 줄이고자 함
   - **OpenAPI 특징에 맞는 캐시 갱신 전략**

        - 동일 로직의 @Cacheable를 사용하는 API와 @CachePut을 사용하는 API로 분리
        - OpenAPI 데이터가 변경됨을 즉시 알 수 없어 사용자의 새로고침 요청 시 OpenAPI를 활용하고자 함
        - 새로고침 요청 시 @CachePut 사용 API를 호출하여 OpenAPI 직접 호출 이후 캐시 갱신

4. **FCM 알림 발송 비동기 처리를 위한 MQ 활용과 데드레터 전략 수립**
    - **비동기 처리 고려**
        - FCM의 응답은 사용하지 않으며 성능을 고려해 아키텍처의 RabbitMQ로 비동기 처리하기로 결정
        - 알림이 여러 API에서 사용되며 전반적인 응답 Latency 에 영향을 주고 있었던 상황
    - **데드레터와 재시도 전략 수립**
        - DB에 진행 상태를 저장하였으며 에러를 파악하기 위해 데드레터 발생 시 Slack 알림 발송
        - 관리자 권한 API를 통해 데드레터 DB PK를 받아 DB 조회 후 큐로 재전송 가능하도록 구성
        - 일시적인 에러는 재시도로 처리하고자 retry queue에서 최소 10초 간격으로 지수 백오프 재시도

5. **무분별한 Mock 사용 제거를 위한 Strategy 패턴을 적용한 테스트 더블**
    - **단위 테스트 중 Mock 문제와 코드 구조 고민**
        - Service Layer 테스트 시 Mock 사용이 많았으며, 저장 이후 목록 조회 등의 로직 동적 테스트 불가
    - **Fake객체와 Strategy 패턴으로 구조 개선**
        - JpaRepository를 상속받는 인터페이스와 Fake객체의 공통 인터페이스를 만들어 상속받도록 수정
        - PK를 통한 조회가 잦았기에 Fake 구현체는 Map<PK, Entity> 를 가지며 실제 동작과 동일하게 구현
        - @TestConfiguration 클래스에 @Primary와 함께 공통 인터페이스 타입으로 Fake 구현체 빈 등록

Other Projects

Showpot 내한 공연 통합 정보 및 알림 서비스 GitHub
예상 트래픽 패턴을 성능테스트하여 인프라 비용을 90% 절감시켰으며 3개월 간 서비스를 운영했습니다.

GitRank v2 깃허브 기여도 랭킹 서비스 GitHub
공모전으로 인한 과한 기술 선택의 아쉬움 해소와 학습을 위해 Java 기반 서버를 Kotlin으로 리팩토링했습니다.

Open Source

rabbitmq/rabbitmq-java-client [2024.11]
PR#1469 / Release v5.23.0
requeue 메트릭 추가 및 해당 메트릭 수집 기능 추가
* PR#1476에서 제 작업이 merge 되었습니다.

quartz-scheduler/quartz [2024.11]
PR#1260 / PR#1261
다중 misfired trigger를 retrieve 중 예외 시 롤백 및 재처리로 인한 무한 실패 이슈를 에러 핸들링으로 해결

Education

한국공학대학교 / 컴퓨터공학부 소프트웨어전공 [2020.03 ~ 2024.02 / 졸업] * 구) 한국산업기술대학교

Activity

2025 DND 해커톤
[2025.05] * 개인상 수상

한국공학대학교 UMC 4기 운영진
[2023.03 ~ 2023.08] * 대학 IT 연합 동아리

교내 프로그래밍 동아리 씨부엉 운영진
[2022.03 ~ 2022.12]

Award

TUKOREA SW-PowerUp
[2023.12] 한국공학대학교 컴퓨터공학부
최우수상 * 졸업작품 관련 수상

한이음 ICT 멘토링 공모전
[2023.12] 한국정보산업연합회
한국정보산업연합회장상(입선)

SW 캡스톤 디자인 콘테스트
[2022.10] 한국공학대학교 컴퓨터공학부
Pre-캡스톤 디자인 부문 동상