Portfolio

이수호

angelsuho0303@gmail.com
github.com/angelSuho
https://resume.suho.info

Projects

WithFestival (위드 페스티벌) https://www.withfestival.site

대학교 축제 플랫폼, WithFestival은 대학교의 모든 학생들을 대상으로 한 통합 축제 플랫폼입니다.

https://github.com/millicon/wf-back

- 개발 기간: 23.04 ~
- 팀 구성: 프론트엔드 2인, 백엔드 3인

기술 스택

Back-end

Java17, gradle, SpringBoot 3.x. Spring Data JPA, QueryDSL, Spring Security, MariaDB, H2, JUnit5, Mockito

Infra

NCP - Server, ObjectStorage, Nginx, Redis, Docker, Github Actions, Apache JMeter

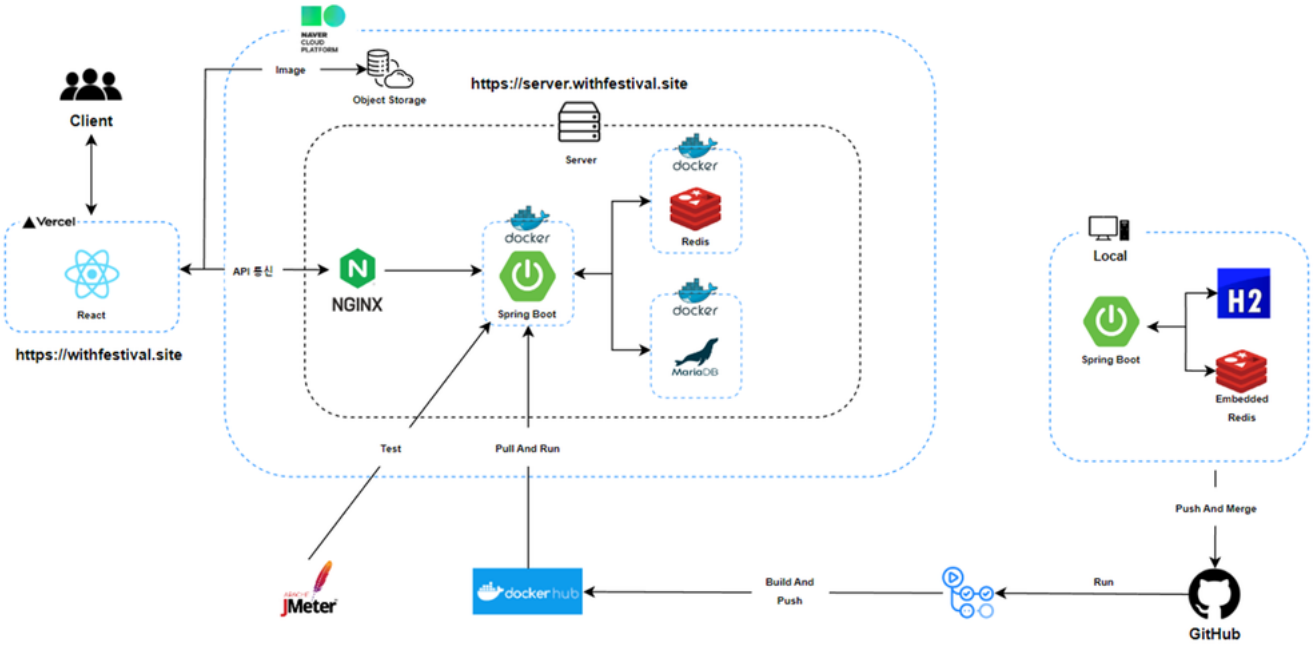
Etc

Git, Discord, Slack

담당 작업

- 대나무 숲, 안내사항 도메인 비즈니스 로직 구현
- 도메인 별 통합 테스트 로직 구현
- Github Actions CI/CD 파이프라인 구축
- 운영 환경에서 발생하는 예외 로그 감지 및 메신저 알림 연동
- 방문자 별 조회수 측정 로직 구현

System Architecure



주요 내용

1) 피크 시간동안 요청 쿼리 갯수를 줄이기 위한 조회수 로직 최적화 (링크)

상황

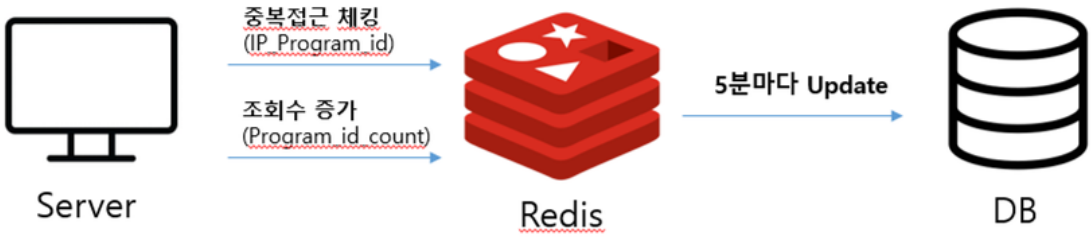
- 웹페이지 조회수 추적 로직은 사용자 방문 시 DB에 업데이트 쿼리를 통해 조회수를 갱신합니다.
- 일상적인 트래픽에는 문제가 없으나, 축제와 같은 이벤트 중 특정 시간(예: 가수 공연 소개 시간)에 방문자 수가 급증할 경우, 이 방식이 서버에 과부하를 초래할 수 있다는 우려가 있습니다.

과정

- 세션, 쿠키, 로컬 스토리지, 그리고 Redis 등을 활용하여 방문자를 검증하고 조회수를 증가시키는 로직의 응답 시간을 측정하기 위한 부하 테스트를 실시하였습니다. (링크)
- 처음 시도 했던 흐름은 레디스에서 중복 접근 체크를 한 뒤에 바로 DB에 조회수를 바로 증가 쿼리를 발생시키는 방법을 적용했습니다.
- 하지만, 사용자가 플랫폼을 사용할 때마다, 매번 조회수를 증가 시키는 쿼리를 날려야 하기 때문에 트래픽이 점점 증가할수록 DB I/O과정에서 부하가 걸립니다.

해결

- 일정 시간동안 각 게시물 별 조회수를 IP주소_도메인이름_ID 형태로 Redis에 저장시켜두고, Scheduler를 통해 DB에 redis 키에 대한 값들을 증가시켜주는 방식으로 구현하였습니다.
- 이렇게함으로써 DB에 발생하는 쿼리를 **5분마다 1회**로 단축할 수 있었습니다.



2) 운영 환경에서 발생하는 예외를 즉시 감지

상황

- 운영 환경의 예외 처리 로그를 수동으로 확인하는 기존 방식은 매번 서버에 들어가 확인할 수도 없고, 실시간으로 예외 발생 사실을 알 수 없었습니다.
- 때문에 플랫폼에 발생하는 오류 대응 시간을 늦추고 사용자 만족도를 저하시켰습니다.

과정

- 예외 대응 프로세스를 개선하기 위해, 예외 유형을 분류하고 각각에 대한 로깅 전략을 확립했습니다.
- 이를 통해 **예외 발생 시 로그를 자동으로 기록**하고, 실시간으로 **메신저**(디스코드)를 통해 개발 팀에 알림을 전송하는 파이프라인을 구축하였습니다.

해결

- 새로운 로깅 및 모니터링 시스템을 통해 운영 환경의 예외를 신속하게 탐지하고 적절히 대응할 수 있게 되었습니다.

3) 단위 테스트와 통합 테스트 범위에 대한 고민과 결정

상황

- 이번 축제 플랫폼 개발 전, 저희 팀은 테스트 코드의 중요성을 상대적으로 가볍게 여기며 주로 개발 기간 후반에 작성하는 경향이 있었습니다.
- 그러나 이번 프로젝트는 실제 운영 환경에 배포될 예정이었기에, 높은 안정성과 예측 가능성을 갖추고 유지보수가 용이한 코드를 제공하는 것이 중요하다고 판단하였습니다.

과정

- 이상적으로는 컨트롤러, 서비스, 리포지토리, 그리고 엔티티 등 모든 계층에 대해 테스트 케이스를 작성하는 것이 좋지만, 제한된 시간 안에 이를 완성하는 것은 현실적으로 어려웠습니다.
- 팀원들과의 협의를 통해 우선적으로 **엔티티, 서비스, 리포지토리 (특히 Querydsl 쿼리 작성 부분)에 대한 단위 테스트와, 서버를 띄우며 전체 요청 흐름을 검증하는 통합 테스트를 중점적으로 작성**하기로 결정하였습니다.

해결

- 노력의 결과, 서비스 운영 중 발생하는 이슈의 수가 이전 프로젝트들에 비해 절반 이상 줄어든 것을 확인할 수 있었습니다. 더불어, **jacoco로 측정한 테스트 커버리지는 80%**를 달성하였고, 기한을 준수하면서도 비즈니스 로직의 안정성과 정확성을 보장하는 높은 수준의 테스트를 작성할 수 있었습니다.
- 이 경험을 통해 테스트의 중요성과 효과를 명확하게 인식하였습니다.

TrendPick (트렌드픽)

TrendPick은 사용자가 원하는 스타일과 유사한 제품을 추천해주는 패션 거래 플랫폼입니다.

🔗 https://github.com/TrandPick/TrendPick_Pro

- 개발 기간: 23.05~23.08
- 팀 구성: 백엔드 4인

기술 스택

Back-end

Java17, Gradle, SpringBoot 3.x. Spring Data JPA, H2, Spring Security, Spring Batch, JUnit5, Mockito

Infra

NCP - Server, ObjectStorage, Nginx, Redis, Docker, Github Actions,
Kubernates, Confluent Kafka, Pinpoint, Apache JMeter,

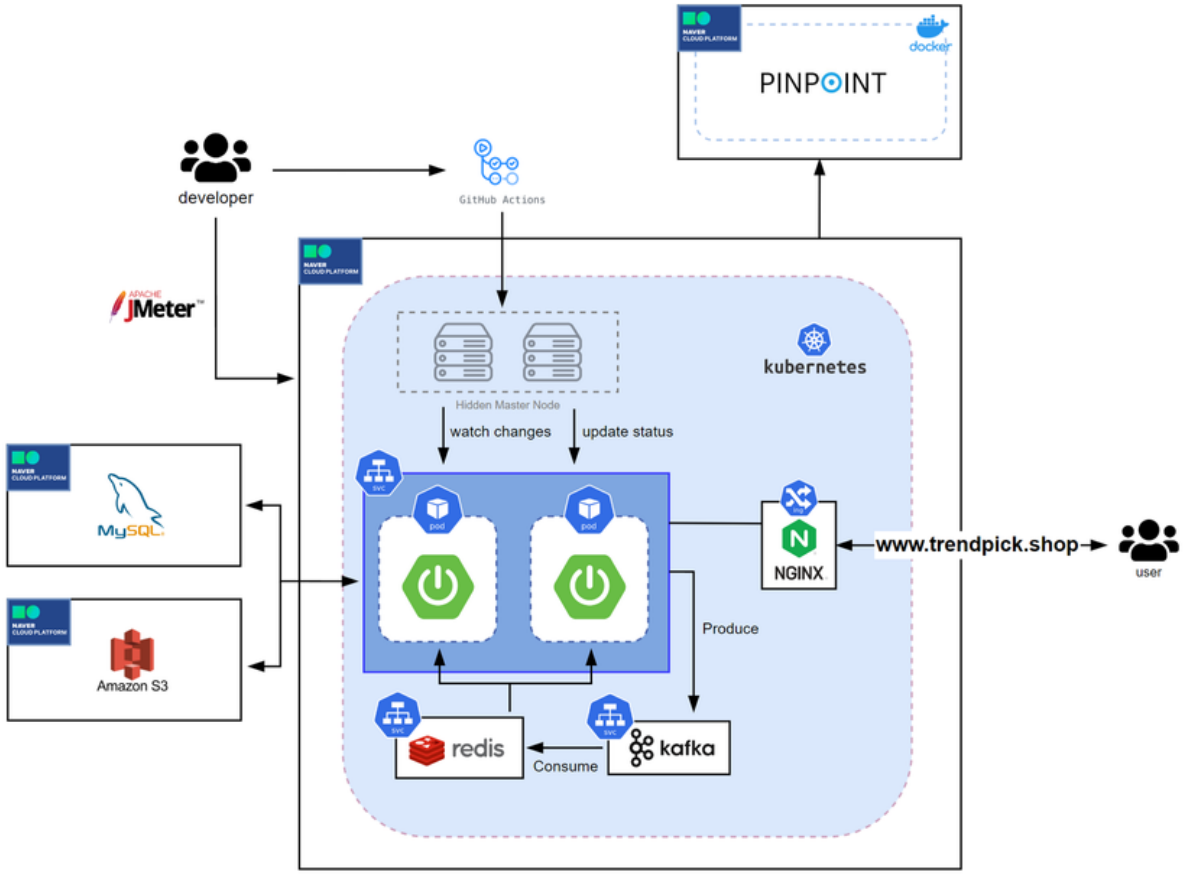
Etc

Git, Discord, Slack

담당 작업

- 상품, 태그, 주문 도메인 로직 구현
- Redis를 활용한 재고에 대한 동시성 문제 해결
- 쿠버네티스 환경, Github Actions CI/CD 파이프라인 구축
- NCP 서버 구축, S3 API를 이용한 파일 업로드 구현
- 서버 모니터링을 위한 Pinpoint 툴 도입

System Architecure



주요 내용

1) 쿠버네티스로 배포, 부하분산

상황

- 팀 회의에서는 쿠버네티스를 이용한 배포의 필요성에 대해 깊이 고민하였습니다.
- 쿠버네티스로 배포를 하게 되었을 때 맞이하는 몇몇의 상황들을 해결해보고 싶었기 때문입니다.
  1. pod의 수 증가에 따른 시스템 처리 용량의 변화를 관찰하여, 쿠버네티스 클러스터의 확장성을 검증
  2. 여러 pod에서 서비스가 실행될 때 발생할 수 있는 재고 관리의 동시성 문제에 대응하기 위해, 분산 데이터 관리 전략이나 동기화 메커니즘을 고려
- 다음과 같은 상황을 공부해보고 실제로 위와 같은 상황에 어떠한 해결 방법을 적용할 수 있을지 미리 구현을 해볼 수 있다는 점이 있었기 때문에 쿠버네티스로 배포를 진행하게 되었습니다.

과정

- 예측 불가능한 트래픽을 대비하기 위해 어플리케이션 서버를 두개의 pod로 실행하고, 이외 다른 인프라 컨테이너를 별도의 pod로 배포하였습니다.
- 이 구성의 목적은 어플리케이션의 확장성과 높은 가용성을 확보하며, 동시에 부하 분산을 통해 트래픽 증가에 효율적으로 대응하는 것입니다.

해결

- pod들을 쿠버네티스의 ReplicaSet을 통해 관리하고, 배포과정 중에서도 쿠버네티스의 롤링 업데이트 기능을 활용하여 일부 pod를 먼저 업데이트하고, 나머지 pod들도 순차적으로 업데이트함으로써 무중단 배포를 구성했습니다.
- 서버 다운시에도 Redis 가용성을 보장하기 위해, 별도의 pod에 배치 했습니다. 이를 통해 서버 장애 발생 시에도 Redis 서비스의 연속성을 유지하여 위기 분산을 할 수 있도록 했습니다.

2) Redis 분산락을 활용한 재고에 대한 동시성 문제 해결

상황

- 플랫폼이 쿠버네티스의 두 개의 pod로 운영됨에 따라, 동일한 재고 값을 유지하기 위한 동기화 과정이 필수적으로 요구되었습니다.

과정

- 초기에는 카프카의 단일 파티션 구성을 통해 동시성 문제를 해결하려고 시도했습니다. 단일 파티션 구성에서는 파티션 내 메시지 순서가 보장되어 동시성 처리가 가능했습니다.
- 그러나, 이 방식은 모든 메시지가 하나의 파티션에 쌓이게 되어 처리 성능이 크게 저하되며, 만약 파티션이 다운되면 데이터 유실의 위험이 존재했습니다. 멀티 파티션 구성을 시도했으나, 이 경우에는 파티션 간 순서 보장이 불가능해 문제가 해결되지 않았습니다.
- 이러한 상황에서 다른 해결책을 모색하게 되었고, DB 락을 활용하는 방안을 고려했습니다. 프로젝트의 환경은 단일 DB였기 때문에, 비관락 방식을 적용하여 재고 동기화 로직을 구현할 수 있었으나, 앞으로의 분산 DB 구축을 고려하면서 분산 락을 적용할 필요성을 느꼈습니다.

해결

- Redis의 분산락(Redisson) 기능을 도입하여 주문 처리의 동시성 문제를 해결하기로 결정했습니다.
- 이 과정에서 요청 순서에 따라 락 획득 순서를 정의하고, 데드락 방지를 위해 락 타임아웃을 설정하는 방법을 적용했습니다.
- 과도한 트래픽으로 인한 서버 부하를 방지하기 위해, Kafka 요청 처리 시 현재 진행 중인 트래픽 양을 확인하고, 이를 기반으로 메시지 전송에 인원 제한을 적용하여 서버 부하를 관리했습니다. (링크)
- 이러한 접근 방식을 통해, Redis 분산락을 활용하여 재고 값의 동시성 문제를 성공적으로 해결할 수 있었습니다.