도전하는 백엔드, 장종인입니다.

자기소개

백엔드 엔지니어 장종인입니다.

"자바 플레이그라운드 with TDD, 클린 코드"라는 NEXTSTEP 강의를 수강하며 백엔드 엔지니어링에 대한 깊은 흥미를 발견하고, 이 분야로의 전향을 결심했습니다. 이 과정에서의 몰입과 프로젝트 참여는 백엔드 개발에 대한 저의 열정을 확인시켜 주었으며, 지속적인 학습과 개선을 통해 이 분야에서 전문성을 쌓아가고 있습니다.

새로운 기술에 대한 깊은 호기심을 바탕으로, 매일의 대중교통 이용 시간을 활용해 kakao tech, NAVER D2와 같은 기술 블로그를 탐색하며 최신 기술 트렌드를 지속적으로 탐구해 왔습니다. 2023년 9월에는 kakao tech에서 공개한 "신뢰성 있는 카프카 애플리케이션을 만드는 3가지 방법" 영상에 주목하여 얻은 인사이트를 토대로, 부트캠프의 팀 프로젝트에서 카프카를 활용해 채팅 API의 메시지 영속성 문제를 해결하는 성과를 달성했습니다.

네트워크 엔지니어로서 3년간의 경험 동안 다수의 고객사와 긴밀히 협력하여 각기 다른 온프레미스 환경에 맞는 VPN 설치 및 구성 작업을 수행했습니다. 이 경험은 네트워크의 거시적인 이해뿐만 아니라, 다양한 역할의 엔지니어와의 협업을 통한 커뮤니케이션 능력 향상에 큰 도움이 되었습니다. 이러한 기술적 배경과 팀원들과의 원활한 소통 능력을 바탕으로 팀 내 협력을 강화하고, 프로젝트의 성공적인 수행을 이끌어 내고자 합니다.

Email : biz.jongin@gmail.com
Phone : 010-7470-8906
Github : https://github.com/lavi15
Blog : https://velog.io/@lavi15

사용 기술

Backend
Java, Java Spring, JPA, MySQL, Oracle, MongDB

DevOps
AWS - EC2, S3, RDS, ECS, Route53, ELB, SNS
Terraform, Jenkins, Docker, Nginx

Frontnd
HTML5, CSS3(SCSS), JS(ES6), React

Tools & Collaboration
IntelliJ, Git, Sourcetree, Jira, Slack

프로젝트

DependenciesVersionHelper github | blog

기간 : 24.01 ~ 24.02

개요 : Spring Boot는 관리하는 의존성에 대해 버전 명시를 자동으로 처리할 수 있음에도 불구하고 이 사실을 인지하지 못해 수동으로 버전을 기입하거나, 해당 의존성이 자동 관리되는지 확인하는 데 시간을 소모하고 있습니다. 이러한 문제를 해결하고 개발자들의 업무 효율성을 향상시키기 위해, Spring Boot가 관리하는 의존성의 버전을 자동으로 식별하여 필요 없는 경우 제거해주는 IntelliJ 플러그인을 개발하였습니다. 개발 과정에서 불필요한 시간 낭비를 줄이고, 보다 직관적이고 효율적인 개발 환경을 제공하는 것을 목표로 합니다.

기술 스택 : Java, IntelliJ Platform Plugin SDK

주요 기능

- 프로젝트 내의 모든 Gradle 파일 파싱
- 해당 Gradle 파일의 의존성 중 Spring Boot가 관리하는 의존성에 해당하고 버전이 기입되어 있다면 버전을 제거
- 해당 Gradle 파일의 의존성 중 Spring Boot가 관리하는 의존성에 해당하지 않고 버전이 기입되지 않다면 버전 확인 및 기입을 지원

성과

- Spring Boot 의존성 관리 프로세스 자동화

역할

- 멀티 Gradle 파일 조회 및 수정 기능 개발
- Spring Boot가 관리하지 않고 버전이 기입되지 않는 의존성 처리 관련 개발
- Kotlin Dsl으로 작성된 gradle 파일도 동작하도록 개발

개선 및 문제 해결

Spring Boot Version 파싱

문제 : Spring Boot Version을 build.gradle이 아닌 다른 파일에서 관리 시 파싱 불가.

원인 : build.gradle의 내용 확인하여 Spring Boot 버전을 파싱.

대안

1. 모든 파일에 Spring Boot 버전이 있는지 확인.
2. 사용자가 사용하는 버전을 기입.
3. IntelliJ의 Libraries를 읽어 버전 확인.

연구 : IntelliJ Platform Plugin SDK

검토

1. 모든 파일 검색
   - 프로젝트의 복잡성에 비례하여 소요 시간이 증가
   - 버전 정보가 저장된 파일의 형식이 다양할 수 있어 타입별 로직이 필요
2. 사용자 입력
   - 사용상 불편함 발생
3. IntelliJ의 Libraries 정보 활용
   - 일관된 로직으로 버전 정보를 파싱 가능

※ 소요 시간이 적고 사용자 편의성이 높은 3번으로 진행

해결

1. IntelliJ 프로젝트의 모든 Libraries 정보를 파싱.
2. Libraries 중 Spring Boot의 버전을 식별 및 추출.

고찰 : 서비스 로직의 설계와 사용자 편의성을 고민하며, 가장 적합한 로직을 선별할 수 있는 기회였습니다. 플랫폼의 특정 API를 이해하고 활용하는 데 여러 어려움을 겪었으나, 새로운 기술을 배우고 적용하는 데 있어 큰 동기부여가 되었습니다.

개선 및 문제 해결

멀티 Gradle 파일 지원 불가

문제 : 멀티 Gradle 파일 지원 불가.

원인 : 플러그인이 Root 모듈의 Gradle 파일만 읽도록 구현.

대안

1. 현재 보고 있는 파일만 읽음.
2. 프로젝트 내의 모든 Gradle 파일을 읽음.

연구 : IntelliJ Platform Plugin SDK

검토

1. 현재 보고 있는 파일만 읽기.
   - 사용자가 원하는 파일만 사용 가능.
   - 프로젝트 전체의 의존성 관리 상황을 파악하기 어려움.
2. 프로젝트 내의 모든 Gradle 파일 읽기
   - 프로젝트 내의 모든 모듈의 의존성 정보를 종합적으로 파악할 수 있음.
   - 프로젝트 내의 모든 모듈의 의존성 정보를 종합적으로 파악할 수 있음.

※ 1번의 경우 유저의 실수로 인한 파일 검사 누락이 발생할 수 있고,
단순 파일명 확인으로 프로젝트 크기가 커짐에 따라 증가 되는
소요 시간의 크지 않아 2번으로 결정

해결

1. IntelliJ 프로젝트의 모든 build.gradle, build.gralde.kts 파일을 찾음
2. 해당 파일들을 탭으로 분리하여 각각의 dependencies를 보여줌.

평가 : 기존 UI를 활용해 탭 형식으로 각 Gradle 파일의 dependency를 보여주는 방식으로 변경했으나, 10개 이상의 다수의 Gradle 파일이 존재할 경우 UI 사용성에 불편함이 발생. UI 디자인에 대한 추가적인 개선 필요

프로젝트

상품 조회 및 구매 API 개발 github | blog

기간 : 23.12 ~ 24.01 (약 7주)

개요 : 개인 프로젝트로 레이어드 아키텍처를 기반으로 하여 CI/CD 파이프라인 구축부터 실제 서비스 배포까지 모든 과정을 직접 수행하며, 전체 소프트웨어 개발 주기에 대한 깊이 있는 학습을 목표로 했습니다.

기술 스택 : Java, Spring Boot, MySQL, JPA, Redis

주요 기능

- 잔액 충전 / 잔액 조회
- 상품 조회(전체 상품 / 인기 상품)
- 주문 / 결제

성과

- 동시성 이슈 해결 방안에 대한 고찰
- CI/CD 구축 및 모니터링까지 다양한 키워드에 대한 고찰

개선 및 문제 해결

재고 관리 동시성 문제

문제 : 동시에 재고 차감 API를 호출할 때, 재고 수량의 차감이 원활하지 않음

원인 : 하나의 프로세스가 재고를 변경하기 전, 다른 프로세스가 동시에 같은 재고에 접근하여 변경을 시도

대안

1. DB를 이용한 비관적락
2. Redis를 이용한 분산락

연구 : 동시성 이슈 해결을 위한 락킹 메커니즘

검토

1. DB를 이용한 비관적락
   - 기존 DB에서 지원하여 별도의 도구 추가 불필요
   - 처리 속도가 느림
2. Redis를 이용한 분산락
   - Redis 추가로 관리 포인트 및 비용 증가
   - 처리 속도가 빠름

※ 실제 사용하지 않는 API로 관리 포인트가 늘지 않는 1번으로 하는게 맞아 보이나, 사용자가 많고 처리 속도가 중요하다고 가정하여 2번으로 진행

해결

1. @Transactional이 붙은 메서드 내에서 재고 차감 시 마다 key가 product에 대한 값의 존재 여부를 확인하는 락킹 처리 구현
2. Transactional commit이 되기전 언락되어 다른 프로세스에서 재고에 접근 가능한 이슈 발생
3. 처리 순서를 Locking > Transactional start > Business logic > Transactional commit > Unlocking 로 변경하여 해결
4. 단일 차감 시 모든 차감 로직을 블록하여 productId를 기준으로 락킹 로직 변경
5. productId가 (1, 2, 4)와 (2, 1, 4) 같이 역순으로 두개의 차감 API 호출 시 데드락 문제 발생
6. productId 기반 정렬 후 락 걸기로 데드락 문제 해결

고찰 : 분산 락과 비관적 락 같은 다양한 동시성 제어 기술을 비교 분석하고 적용해보면서, 기술적 선택이 실제 서비스 성능과 사용자 경험에 미치는 영향을 고민할 수 있는 경험이었습니다. 또한, 프로젝트를 진행하면서 발생하는 예기치 않은 문제들을 해결하기 위해 다양한 기술 문서를 참고하고, 커뮤니티의 지식을 활용하는 등 지속적인 학습의 중요성을 체감할 수 있었습니다.

개선 및 문제 해결

인기 상품 조회

문제 : 인기 상품(3일간 주문이 가장 많은 상품) 조회 API의 성능이 낮음

원인 : API 요청 시 마다 조회 쿼리를 실행

대안

1. Redis에 인기 상품 목록을 별도로 저장하고 매일 00시에 갱신

연구 : 스케줄링, 어플리케이션 레벨의 캐싱

검토

1. Redis에 인기 상품 목록 저장
   - 기존 락을 위해 Redis를 도입하여 추가적인 구축 불필요
   - Redis의 경우 메모리에 값을 저장하여 읽어 오는 속도가 빠름

※ 매번 쿼리를 통해 인기 상품을 조회하는 로직에 비해 구축된 Redis를 이용하여 저장된 값을 가져오는게 성능적으로 이득으로 판단되어 진행

해결

1. 인기 상품 목록을 Redis에 저장하고 매일 00시에 갱신하는 스케줄을 설정
2. 성능 테스트를 위해 메모리 1GB, CPU 2코어 사양의 도커 컨테이너를 구성하고, nGrinder를 이용해 부하 테스트를 진행
   (1000번의 요청중 약 75% 성공, 컨테이너의 CPU, 메모리 사용량이 100% 후 다운)
3. Redis와 네트워크 통신으로 인한 시간 지연을 줄이고자 어플리케이션 레벨의 캐시 사용을 고려
4. @Cacheable 어노테이션을 사용하여 캐시에 저장하고 매일 00시에 갱신하는 스케줄을 설정
5. 동일한 사양의 도커 컨테이너에서 nGrinder를 이용한 부하 테스트를 재실행
   (1000번의 요청 중 100% 성공, 평균 응답 속도 약 8.5배 증가)

고찰 : 초기에는 Redis를 활용한 접근 방식이 성능 개선에 기여할 것으로 예상했으나, 실제로는 어플리케이션 레벨의 캐싱이 더 효과적인 해결책이 됨을 발견했습니다. 이 과정에서 추가 비용 발생과 기술적 복잡성 증가에도 불구하고 성능이 기대에 미치지 못하는 경우가 있음을 목격했습니다. 따라서, 기술을 도입하기 전에는 해당 기술의 장단점을 철저히 검토하고, 실질적인 필요성을 바탕으로 결정을 내리는 접근 방식이 중요하다는 것을 깨달았습니다. 기술이 제공하는 이점이 명확하고, 해당 기술 없이는 해결할 수 없는 문제 상황에서만 새로운 기술을 도입하고자 합니다.