



이동훈 | Donghoon Lee



[donghoony](#)



<https://blog.hoony.me>



aru0504@naver.com



[010 2687 3552](tel:010-2687-3552)

안녕하세요, 백엔드 개발자 이동훈입니다.

알고 있는 지식을 아낌없이 나누며 즐거움을 느낍니다.

좋은 코드는 좋은 문화에서 쉽게 탄생한다고 생각합니다.

사람의 실수를 빠르게 바로잡고, 코드로 해결하며 성취감을 느낍니다.

교육

우아한테크코스 | 6기 웹 백엔드 과정, 2024.02 – 2024.11

건국대학교 서울캠퍼스 | 컴퓨터공학부, 2019.03 – 2026.02 졸업예정

GPA **4.3**/4.5 | 전공 **4.47**/4.5, 전공 61학점

기술

| 배움에서 즐거움을 얻습니다. 갖고 있는 지식을 바탕으로 빠르게 학습합니다.

Java, Spring Boot(Web, Data JPA), MySQL

AWS(VPC, EC2, ELB), Docker, GitHub Actions

Prometheus, Grafana

활동 | 알고리즘 문제해결을 좋아합니다. 대회에 출전하고 동아리를 이끌기도 했습니다.

AlKon | 알고리즘 동아리 초대 회장 2023.03 – 2023.12

ICPC | 국제 대학생 프로그래밍 대회 본선 진출 2022, 2023

KUPC | 교내 프로그래밍 경진대회 개최, 출제 및 검수 2019, 2022, 2023



리뷰미

| 동료 개발자로부터 나를 발견해요

 [리뷰미 URL](#) |  [GitHub 저장소](#)

‘나는 어떤 개발자일까?’로부터 출발한 동료 리뷰 서비스입니다

리뷰미는 함께 협업한 팀원으로부터의 피드백으로부터 나의 강점을 알아갈 수 있는 서비스입니다. 직접 리뷰 그룹을 만들어 리뷰 작성 링크를 생성하고, 받은 리뷰를 손쉽게 확인할 수 있습니다. 몰랐던 장점을 동료로부터 발견하거나, 강점을 재확인함으로써 내가 어떤 개발자인지 견고하게 할 수 있습니다.

프론트엔드 4명, 백엔드 4명이 함께 개발합니다

프론트엔드와 백엔드 사이의 협업도 중요합니다. 프론트엔드와의 페어 프로그래밍을 자주 제안했습니다. 실제로 몇 개의 기능은 프론트엔드와 함께 코드를 확인하며 구현되었습니다.

팀원 모두가 힘껏 노를 저었습니다. 방향타는 사용자가 잡았습니다

초기에 기획했던 회원 기반 기능은 구현되지 않은 채 비회원 기반의 서비스를 운영하고 있습니다. 몇 번의 실험 결과 ‘질 좋은 리뷰’, ‘나의 리뷰 모아보기’를 향한 관심이 로그인의 수요보다 높았기 때문입니다. 사용자가 좋아할 것으로 짐작하는 것과, 실제로 사용자가 원하는 기능이 서로 다를 수 있음을 알았습니다.

기술 스택

Java, Spring Boot, MySQL, Docker, AWS, GitHub Actions

개발 기간

2024.07 - 진행 중

REVIEW ME

리뷰 연결 / [리뷰 작성](#)

리뷰미

아루를 리뷰해주세요!

시간관리 능력

이제, 선택한 순간들을 바탕으로 아루에 대한 리뷰를 작성해볼게요.

시간 관리 능력에서 어느 부분이 인상 깊었는지 선택해주세요. (1개 ~ 2개)

- ☒ 프로젝트의 일정과 주요 마일스톤을 설정하여 체계적으로 일정을 관리해요.
- ☒ 일정에 따라 마감 기한을 잘 지켜요.
- ☒ 업무의 중요도와 긴급성을 고려하여 우선 순위를 정하고, 그에 따라 작업을 분배해요.

REVIEW ME

리뷰 연결 / [리뷰 목록](#)

리뷰미

아루가 받은 13개의 리뷰 목록이에요

목록보기

모아보기


2024-10-28

백엔드 단에서 큰 일이 일어나면 앞장서서 문제의 원인을 파악하고, 해결하는 아루의 모습이 인상 깊었어요. 책임감있게 끝까지 파헤쳐서...

💡 문제 해결 능력 (예: 프로젝트 중 만난 버그/오류를 분석하고 이를 해결하는 능력)

🔧 기술적 역량, 전문 지식 (예: 요구 사항을 이해하고 이를 구현하는 능력)

Q. 위에서 선택한 사항과 관련된 경험을 구체적으로 적어 주세요.

형광펜 ① 

- 특히 아루가 이해가 되지 않는 부분이 있으면 구체적으로 어떤 부분에서 막혔는지 먼저 물어봐주고, 다시 차근차근 설명해줘서 편안하게 질문할 수 있었어요. "왜? 이런 것도 이해 못 해?"가 아니라, **언제나 친절하게 "이해되지 않으면 다시 설명해줄게"**라는 마인드로 대해줘서 **질문하는 입장에서 너무나 고맙웠어요**. 아루 덕분에 부담 없이 배우고 성장할 수 있는 환경이 만들어져서 감사해요!
- 서비스의 요구사항은 계속해서 늘어나는 상황에서 '좋은 코드'를 지키는 일이 얼마나 어려운지 체감했던 이번 데모 기간이었습니다. 그 때 마다 아루가 코드의 기본을 지키는 자세를 상기시켜주어서 고맙웠습니다. 여러 요구사항들로 인해 그 기본을 놓치고 있었다는 것도 아루 덕분에 발견하게 되었어요. 본인의 코드에서만 그걸 적용하는 게 아니라 끊임없이 주위에 이 중요성을 상기시키고, **어떻게 하면 개선할 수 있을지** 본인의 팁을 공유해주는 팀원이 있다는 것이 정말 소중한 경험입니다.
- 아루를 보면 볼 수록, 개발과 관련되어 굉장히 넓은 범위로 호기심이 있고 이를 깊이 파고들며 그로인해 가지고 있는 지식이 굉장히 풍부해요. 색약자를 생각해서 차트 색깔을 말할 때는 프론트 크루들도 미처 생각하지 못한 웹 접근성이라 정말 깜짝 놀랐어요.
- 아루의 기술 역량과 풍부한 지식은 팀의 전체적인 수준을 높이는 데 기여하고 있어요. 관련 인프라 지식으로 CI/CD를 구축하는 데 큰 도움



도메인과 강결합돼 확장에 어려움을 겪은 타입을 추상화, DB 스키마 변화 무중단 마이그레이션

[중복 테이블 생성 PR](#)

[레거시 코드 제거 PR](#)

초기 기획, 구현 단계에서는 리뷰 질문에 대한 답변이 주관식/객관식으로만 이루어져 있을 것으로 판단해 단순 if-else로 해당 질문/답변 타입을 처리했습니다.

하지만 발전 과정에서 이런 단순 분기 처리는 확장에 걸림돌이 되었습니다. 1점 - 5점 사이의 스케일 타입 답변을 도입할 때 변경해야 하는 부분이 많았습니다. 이를 해결하기 위해 답변 클래스를 추상화했고, 이는 DB 테이블 스키마의 변화로 이어졌습니다.

기존 애플리케이션의 동작을 보장하면서 데이터를 옮기기 위해 고민했습니다. 리뷰는 **작성 이후 수정될 수 없다**는 서비스의 정책을 활용해 돌파구를 찾았습니다. 총 두 번의 배포로 추상화를 적용하고, 레거시 코드를 덜어낼 수 있었습니다. DB 마이그레이션은 아래와 같은 단계로 이루어졌습니다.

1. 구 테이블과 신 테이블의 Auto-increment를 일정값으로 설정
2. 애플리케이션에서 데이터 삽입 시, 구 테이블과 신 테이블에 모두 삽입하도록 수정
3. 설정한 Auto-increment 값보다 작은 ID를 가지는 행을 신 테이블에 정제해 이동
4. 데이터의 정합성을 확인한 뒤, 구 테이블 및 애플리케이션에서의 레거시 코드 삭제

이후 애플리케이션에서는 @Inheritance를 활용했습니다. 추상 클래스인 Answer를 상속하여 TextAnswer와 같이 구현했습니다. 구체 타입에 의존하지 않고 추상화된 타입에 대해서만 확인하니 훨씬 유연한 코드가 되었습니다.

```
private void validateAnswer(List<TextAnswer> textAnswers,
                             List<CheckboxAnswer> checkboxAnswers) {
    textAnswers.forEach(textAnswerValidator::validate);
    checkboxAnswers.forEach(checkBoxAnswerValidator::validate);
}
```



```
private void validateAnswer(List<Answer> answers) {
    for (Answer answer : answers) {
        AnswerValidator validator =
            answerValidatorFactory.getAnswerValidator(answer.getClass());
        validator.validate(answer);
    }
}
```

답변 내용을 검증할 때 | 요구사항 변경에 취약한 구조 개선

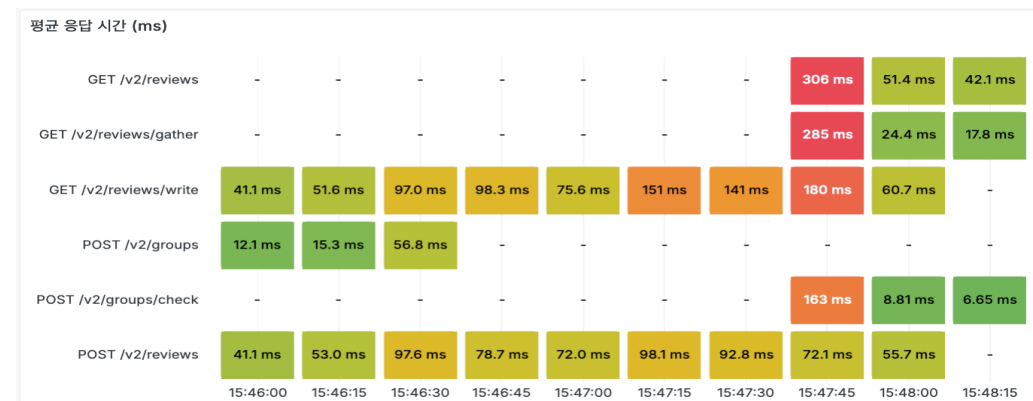


부하 테스트를 기반으로 병목 확인 및 성능 향상 [부하 테스트 결과](#)

Scale-out은 사용자를 많이 감당할 수 있는 방법 중 하나이지만, 근본적인 해결 방법은 아닙니다. 하나의 서버가 많은 사용자를 감당할수록 서버 증설에 드는 비용을 아낄 수 있습니다. 현재 상황에서 많은 사람들이 서비스를 사용할 때, 어디까지 감당할 수 있을지 확인해 보고 개선할 부분을 찾아보았습니다. K6로 부하 테스트를, Grafana로 그 경과를 확인했습니다.

성능을 측정하기 위한 척도가 필요해 시나리오를 작성했습니다. 실제 서비스를 사용하는 것처럼 일련의 핵심 기능을 한 번 이상 수행합니다. 모든 API 요청에 대해 1초 내로 응답을 받도록 목표를 설정했습니다.

가상 사용자가 늘어날수록 병목이 눈에 띄었습니다. 현재 운영 서버의 CPU/RAM 설정에 맞게 스레드 풀, DB 커넥션 풀 크기를 설정했습니다. 조회 병목에 해당하는 데이터는 업데이트가 일어나지 않아 캐시를 적용했습니다. 부하 테스트를 처음 진행했을 때는 VU 300으로도 600ms가 넘는 응답 속도를 보였었는데, 다양한 최적화를 적용해 VU 700도 1초 안의 응답속도로 감당할 수 있게 되었습니다.



조회 캐시를 적용하기 전(위)과 후(아래) 응답속도 비교 | VU 700

(t4g.small 인스턴스 기준)



도메인 연관관계 및 경계 설정, 생명주기를 기반으로 연관관계 도입

[관련 블로그 글](#)

객체지향과 데이터베이스 사이의 패러다임 불일치를 해소하기 위해 ORM이 등장했습니다. 프로젝트 초기에는 대부분의 객체에 @OneToMany, @ManyToOne 과 같은 연관관계를 두었고, 의존이 서로 얽힌 하나의 큰 도메인이 되었습니다. 팀원 모두 기능 구현, 테스트 작성에 어려움을 겪었습니다. 객체지향을 최대한 반영하려고 했지만, 몸집이 커진 객체 간의 상호작용은 유연한 코드를 작성하는 데 어려움을 불러왔습니다.

팀원과 수많은 좌절과 고민 끝에, 결국 생명주기를 기준으로 생성과 소멸을 함께 할 때만 객체로 참조하기로 했습니다. 그 외에는 id를 사용해 간접적으로 참조합니다. 예를 들어, 리뷰가 등록될 때 답변도 함께 작성되므로 리뷰와 답변은 객체 참조를, 리뷰와 리뷰 그룹은 저장되는 시기가 다르므로 id를 참조합니다. 객체를 따라가면 자연스럽게 하나의 도메인도 여러 경계로 나뉘었습니다.

소프트웨어를 개발하는 데 유연하게 변경할 수 있는 것이 중요한 지표가 된다는 것을 알았습니다. **객체지향이 좋은 코드가 아니라, 좋은 코드를 향한 길 중에 객체지향이 존재한다는 것**을 깨달은 소중한 시행착오입니다.

팀 내 CI/CD 및 Github Actions 세미나 진행, 백엔드 CI/CD 워크플로우 작성

[관련 블로그 글](#)

프로젝트가 시작되고 얼마 지나지 않아, 컴파일되지 않는 코드가 머지되는 일이 있었습니다. 코드리뷰를 받더라도, 그 코드가 충분히 믿을 만한지는 별개의 사안이었습니다. 사람의 실수로 불완전한 코드가 존재하는 상황이 존재할 수 있었습니다. CI의 필요성을 느꼈고, GitHub Actions를 도입해 자동화했습니다.

프론트엔드와 백엔드가 같은 레포지토리 아래에서 개발이 진행되었습니다. 이때 각 분야의 워크플로우가 다른 분야에 영향을 미칠 수 있었습니다. 서로의 코드는 디렉토리로 분리돼 있었기 때문에, 워크플로우 조건에 디렉토리를 명시해 하위 디렉토리의 변경을 감지하도록 작성했습니다.

AWS 보안 그룹의 인바운드 정책으로 인해 외부에서 SSH를 사용해 개발/운영 서버 접근이 불가능했습니다. GitHub에서 제공하는 Self-Hosted Runner로 CI/CD 워크플로우를 구성했습니다. CI 과정에서 코드를 컴파일하고 테스트하며, CD에서 코드 빌드, 배포가 이루어집니다.



API-First Design 도입 실패와 대안 제시 [OpenAPI-Generator 도입 시도 PR](#)

리뷰미는 Swagger를 활용해 API 문서를 작성했습니다. 프로덕션 코드에 문서를 위한 코드가 포함된다는 단점도 있었지만, 빠르게 문서를 작성해 공유할 수 있다는 점, 비용이 RestDocs에 비해 크지 않다는 점 등을 고려해 채택했습니다.

하지만 시간이 지나며 API 문서가 노선, Swagger 등 다양한 곳에서 공유되고 있었습니다. 여러 출처에 존재하는 문서는 불일치를 불러왔고, 관리할 곳이 늘었습니다.

API-First Design에 대해 학습했습니다. OpenAPI 명세에 맞는 문서를 작성한 뒤 API 문서와 테스트 코드를 자동화하는 것을 시도했습니다. 하지만 현재 프로젝트의 진행 상황을 고려했을 때 팀원들에게 부담이 가중될 것이라는 의견으로 반려되었습니다.

여전히 문제 상황은 지속되었기에, 대안으로 RestDocs를 활용했습니다. API를 먼저 프론트엔드와 함께 설계한 뒤 확정합니다. 백엔드에서 최우선적으로 기능을 제외한 API 테스트 코드를 작성하고 이를 기반으로 API 문서를 자동으로 생성합니다. 문서 출처는 단 한 곳으로 줄었고, 빠르게 프론트엔드와 공유해 분야와 상관없이 기능 구현에만 집중할 수 있게 되었습니다.

[BE] feat: 리뷰 모아보기 API 구현 ✓

#806 by nayonsoso was merged 3 weeks ago • Approved 6차 스프린트: 최...

[FE] feat: 리뷰 모아보기 페이지의 공통 컴포넌트 구현 및 퍼블리싱 ✓

#790 by chysis was merged 3 weeks ago • Approved 6차 스프린트: 최...

[BE] docs: 리뷰 모아보기 RestDocs 작성 ✓

#785 by nayonsoso was merged 3 weeks ago • Approved 6차 스프린트: 최...

문서 작성 후 각 분야가 병렬적으로 개발

[BE] feat: 하이라이트 추가 및 수정 API 구현 ✓

#813 by skylar1220 was merged 3 weeks ago • Approved 6차 스프린트: 최...

[BE] docs: 답변 하이라이트 API 문서 작성 ✓

#800 by donghoony was merged 3 weeks ago • Approved 6차 스프린트: 최...

작성한 문서를 기반으로 다른 팀원이 기능을 구현하기에도 용이



다중화 환경에서의 인프라 설정 일관성 문제, 형상관리로 해결

 [관련 블로그 글](#)

서비스를 운영하면서 코드와 인프라 모두 들여다볼 일이 많았습니다. 당시 프로젝트 코드와 인프라는 서로 관리하는 방법이 달랐습니다. 코드는 Git으로 관리되고 있었지만, 인프라 설정은 별도의 중앙 설정이 존재하지 않았습니다. 설정 변경은 각각의 EC2 인스턴스에 대해 수동적으로 전파되었고, 이 과정에서 휴먼 에러가 발생할 가능성이 존재했습니다.

‘터미널에 접속하지 않고 모든 설정을 할 수 있을까?’라는 욕심이 생겼고, 이를 조금씩 실행에 옮겼습니다. 설정 정보에는 민감 정보가 존재했기에 별도의 비공개 저장소에서 작업했습니다. 애플리케이션 설정, 로그/매트릭 수집 설정, Docker 구성 등이 이곳에서 관리되었습니다. 이제는 더 이상 새로운 환경 설정을 위해 vi 명령어를 사용하지 않습니다.