고정완

https://github.com/ghojeong/resume
010-3878-1459
rhwjddhks@gmail.com

보유 기술

- Java
- JavaScript
- Kotlin
- Spring Boot
- Spring Security
- JPA
- OOP
- TDD, ATDD
- REST Docs

실무 경험

Java Spring Developer | Dreamfora 2021.08.25 –
Java, Spring, JPA, MariaDB 기반의 API 서버 개발, 인프라 구축, 모니터링을 담당하였습니다.
3 명의 개발팀으로 앱의 누적 다운로드 200 만을 달성하고, 미국 앱스토어 에디터에게 선정되었습니다.

Web Developer | Ecube Labs 2018.10.15 – 2020.12.28
TypeScript, React, NodeJS 기반으로 미국 볼티모어에 서비스하는 스마트 시티 솔루션을 개발 및 유지보수 했습니다.

교육 경험

Kotlin Reviewer | 우아한테크코스 2024.02.13 – 2024.06.24
안드로이드 6 기에서 리뷰어로 활동했습니다.

JPA Reviewer | NEXTSTEP 2023.10.14 – 2023.11.25
만들면서 배우는 JPA 를 이수하고, 2 기와 3 기에 리뷰어로 활동했습니다.
JSR 338 문서를 기반으로 QueryBuilder, EntityManager, 1 차 캐시, OneToMany 프록시를 구현했습니다.

Spring Reviewer | NEXTSTEP 2022.07.18 – 2022.09.29
만들면서 배우는 Spring for 리뷰어를 이수하고, 3 기에 리뷰어로 활동했습니다.
웹서버, MVC, JDBC, DI, AOP 관련 프레임워크를 바닐라 자바를 통해 TDD 로 구현했습니다.

Java TDD Reviewer | NEXTSTEP 2021.03 – 2021.05
Java TDD 10 기를 우수하게 완주하여, 11 기 부터 Java TDD 과정의 리뷰어로 활동했습니다.
객체지향 생활체조 원칙과 단위 테스트 작성을 습관화 하게 되었습니다.

Backend Masters | CodeSquad 2021.01 – 2021.06
Java 와 Spring 기반으로 백엔드 서비스를 만드는 부트캠프를 수료했습니다.
DB 사설망 분리, Bastion Host, IAM Role, Block Storage, Object Storage 에 관해 배웠습니다.

세미나 강사 | 한빛미디어 2019.03 – 2019.06
개발 문서화를 주제로 한빛미디어의 후원을 받아 세미나를 개최했습니다.
사전정의서, 요구사항 명세서, ERD, 프로세스 Flow Chart 문서 작성하는 법을 세미나에서 다루었습니다.

컴퓨터공학 학사 | UNIST
컴퓨터공학을 성적 우수 졸업했습니다. 전공자 수준의 CS 지식이 있습니다.
전공과목 성적: 알고리즘 A+, 데이터구조 A, 데이터베이스 A-, 운영체제 A-, 네트워크 A-

지원 동기

그란데클립의 “치열하게 토론하고 뜨겁게 실행한다” 라는 소개 문구에 끌렸습니다. 무미건조하게 업무 지시를 주고받는 문화가 아니라, 팀 혹은 고객에게 줄 수 있는 긍정적인 가치를 잡담 할 수 있는 문화라고 느꼈습니다. 손현태 님께서 합류 이유로 “좋은 팀을 함께 만들어갈 수 있다는 확신” 을 적어주셨는데, 무엇이 좋은 팀인지, 좋은 팀을 만들려면 어떻게 해야 하는지 이미 경험한 분들이 합류해 계시다는 인상을 받았습니다. 저는 그란데클립에서 팀이 성장하는 과정을 경험해보고 싶습니다. 제대로 실패하고 그만큼 함께 성장하고 싶어서 지원합니다.

자기 소개

저는 피드백을 소중히 생각합니다. 다면 평가에서 저는 동료들에게 “궂은일도 맡아 하는 이타적인 업무방식” 과 “타인의 피드백에 대한 긍정적인 수용” 을 장점으로 평가 받아왔습니다. 피드백은 행동의 개선으로 이어져야 한다고 생각하며, 개선의 결과는 개인의 성장을 넘어 팀 전체에 긍정적이어야 한다고 믿습니다. 피드백을 받았음에도 무엇을 실패했는지 원인조차 모르는 “Not Even Wrong” 상태를 굉장히 경계하며, 무작정 행동하기에 앞서 문제가 정확히 무엇인지 상황과 맥락을 파악하기 위해 노력합니다.

백엔드 개발자로써 제가 재직중인 회사에서 일궈낸 성과는 다음과 같습니다. 첫째, 웹 호스팅 업체에게 서버를 임대하여 진행하던 서비스를 AWS 클라우드로 이전하였습니다. 클라우드로 이전하며 WAS 가 죽었을 경우를 대비한 이중화를 하고 보안을 위해 DB 망 분리를 했습니다. 둘째, 단위 테스트와 인수 테스트를 도입하고 Rest Docs 를 통한 문서화를 도입했습니다. 셋째, 배포 시 서비스가 정지하는 일을 막기 위해 무중단 배포 파이프라인을 구축했습니다. 넷째, 1 정규형을 지킬 수 있도록 DB 스키마를 새로 모델링하였습니다. 다섯째, 재설계한 스키마로 기존 데이터를 마이그레이션했습니다. 여섯째, 서버의 모니터링과 로깅 시스템을 구축했습니다. 굉장히 급진적인 변화였지만, 이 모든 과정을 저의 독단으로 진행하지 않고 경영진과 비개발팀을 설득하며 진행하였습니다.

1) AWS 클라우드로 인프라를 이전

드림포라에 처음 입사했을 때, 경영진은 외주를 통해 서버 개발과 유지보수를 하고 있었습니다. 서버는 웹 호스팅 업체에서 임대한 서버로 운영되고 있었고, AWS 의 스타트업 크레딧을 활용하고 싶었던 경영진은 저에게 클라우드로의 서버 이전 업무를 맡겼습니다. 기존 서버는 월 50 만원짜리 단일 WAS 와, 150 만원 짜리 단일 DB 서버로 되어 있었고, 두 개의 서버 모두 public 하게 공개되어 있었습니다. AWS 로 이전을 하며 VPC 를 활용해 EC2 WAS 와 RDS 를 사설망으로 분리했고, 로드밸런서와 bastion 용 EC2 서버 하나를 인터넷 네트워크에 공개했습니다. 스타트업 크레딧은 메가존 클라우드를 통해 지원받았기에, 메가존 클라우드에게 필요한 권한은 IAM Role 을 부여하여 제공했습니다. 이전을 하며 망분리와 WAS 이중화, RDS 의 백업 설정을 하였고, 기존 서버가 컨테이너화 되어 있지는 않았기에 기존 서버의 운영체제 설정을 터미널을 통해 하나하나 분석하며 옮겼습니다. 다음에 또 이렇게 서버를 옮겨야 할 때 터미널로 서버 설정을 확인하기 힘들 것 같아, 분석하는 김에 서버환경과 관련한 도커 파일을 작성하였습니다.

클라우드로의 이전을 하며 겪었던 가장 큰 문제는 DNS 였습니다. 도메인 권한을 가진 서버를 가리키는 Name Server 의 TTL 은 172800 초였고, 그로 인해 48 시간 동안은 라우터의 캐시에 따라 네트워크 패킷이 새로 만든 AWS 서버와 기존의 웹 호스팅 서버 중에 어디로 갈지 모르는 상태가 되었습니다. 다행히 패킷이 길을 잃는 원인이 네트워크 라우터 캐시의 TTL 때문이라는 것을 알았기에, 과거의 웹 호스팅 서버로 패킷이 도착할 경우 새로 만든 AWS 서버의 반환값을 가져오는 중개 서버를 사이에 끼워 넣었습니다.

2) TDD 기반의 리팩터링과 REST Docs 의 도입

클라우드로 서버를 마이그레이션 한 이후에는 백엔드 서버를 리팩터링하기 시작했습니다. 외주를 통해 개발된 Spring 서버는 Controller 에 모든 로직이 존재하고, DTO 와 도메인 모델 구분이 되어있지 않았습니다. 특히 로그인을 하고 나면 서비스가 굉장히 느려지는 문제가 있었는데, 원인을 추적해보니 세션이 WAS 의 메모리가 아니라 RDB 의 테이블을 활용해 동작하고 있던 게 원인이었습니다. JSessionID 별로 row 가 새로 생성되고 있었고 심지어 인덱스가 전혀 걸려있지 않았습니다. 따라서 로그인을 하고 나면 그 이후로의 모든 API 동작은 8 천만 개 정도의 row 가 쌓인 테이블을 항상 풀스캔을 하며 동작을 하고 있었습니다. 풀스캔 때문에 느려지는 문제를 DB 의 스펙을 올려가는 방식으로 해결해왔지만, 지금과 같은 상황에서는 설사 사용자가 많지 않더라도 서버를 오래 켜둘 수록 느려지기 때문에 그대로 놔둘 수가 없다고 판단했습니다. 응급처치로 세션을 RDB 가 아닌, Redis 로 옮겼고 날짜가 반년 이상 오래된 세션들을 모두 삭제하는 작업을 진행했습니다. 응급처치 후 앱의 실제 요구사항을 살펴보니 세션이 필요한 로직이 거의 없어서, 이 참에 JWT 기반의 sessionless 형태로 서버를 리팩터링하기로 하였습니다.

외주를 통해 개발된 서버라 그런지 API 인터페이스에 관한 문서화가 전혀 되어 있지 않았습니다. 심지어 필드의 이름도 통일되어 있지 않습니다. 어떤 API 에서는 비공개 여부를 나타내는데 isPrivate 을 쓰는가 하면, 어떤 곳에서는 그냥 private 이라고 되어 있었습니다. 안드로이드 측에 제대로 된 API 문서가 제공되고 있지 않다보니, 서버의 인터페이스와 안드로이드에서 보내는 JSON 이 일치하지 않아서 발생하는 버그도 무척 많았습니다. 이 혼란을 해결하려면 서버의 코드가 아니라, 안드로이드 코드와 요구사항 문서를 보고 서버를 전부 다 새로 구현하는 게 빠르다는 판단을 했습니다. 서버 리팩터링을 위해 안드로이드 코드를 기반으로 서버가 어떤 Request 를 받아서 어떤 Response 를 보내주어야 하는지에 대한 인수 테스트를 작성했습니다. 그렇게 작성한 인수 테스트는 REST Docs 를 통해 문서화 하여, 안드로이드 동료에게 제가 실수한 부분은 없는지 한번 더 검수를 받았습니다. 작성한 인수 테스트에 확신이 들면 제가 새로 작성한 DTO 와 Controller 를 기존 서버의 로직과 연결했습니다. 기존 서버에 Layer 가 분리되어 있지 않았기 때문에 기존의 Controller 를 Service 객체로 이름을 바꾸었습니다. 안드로이드에서 사용하고 있는 모든 API 의 호출 성공 시나리오에 대한 인수 테스트를 모두 작성한 다음에는, 비대해져 있는 Service 의 로직을 도메인 객체로 옮기는 리팩터링을 하였습니다. 단위 테스트를 작성하며 리팩터링을 하였고, 리팩터링을 마치고 나니 **테스트 커버리지가 80%** 에 도달해 있었습니다.

3) 무중단 배포 파이프라인 구축

리팩터링을 마무리하여 신규 기능을 개발 속도가 빨라지고 나서는 무중단 배포 파이프라인을 구축할 필요가 생겼습니다. 그전에는 사용자에게 공지를 띄우고 배포를 하는 5 분 동안 서비스를 중지했었는데, 그런 불편함이 자주 생기지 않도록 무중단 배포 파이프라인을 구축했습니다. Dev 와 Stage 서버의 경우에는 NGINX 를 통해 포트를 바꾸는 형식의 Blue-Green 배포 방식을 취했습니다. 빌드와 배포 서버 비용을 아끼고 싶었기 때문에, 배포를 하는 모든 내용은 직접 shell script 로 작성하여 자동화 하였습니다. Production 서버의 경우에는 ECS 를 활용한 Rolling 배포 방식을 도입했습니다.

4) DB 스키마 재설계

서버를 리팩터링 했지만, DB 스키마 구조로 인한 문제는 여전히 남아있었습니다. 인덱스가 제대로 걸려있지 않았던 것도 문제였지만, 스키마가 제대로 설계되어 있지 않아서 쿼리를 작성하는 일 또한 쉽지가 않았습니다. 가령 요구사항에서는 명백히 다른 엔티티로 정규화 되어야 함에도, 하나의 테이블에 지나치게 많이 통합되어 있는 경우가 많았습니다. JOIN 쿼리를 짜기 힘들 정도로 통합된 경우에는 재귀 쿼리를 사용하는 경우가 많았는데, 재귀의 정지조건이 올바르지 못한 경우에는 성능이 급격하게 느려지는 현상이 종종 발생하였습니다. 이렇게 급격하게 느려질 때는 DB 의 리소스를 많이 잡아먹어서, 무거운 쿼리가 존재하는 서비스와 상대적으로 가벼운 서비스를 아예 별도 DB 와 서버로 분리해야 하나를 고민했었습니다. 하지만 스키마를 재설계하지 않으면 기술 부채가 되어 계속 문제가 생길 것 같아서, 스키마를 정규화 한 후 쿼리 필요에 따라 다시 통합하였습니다. 이 과정을 통해 좋아졌던 첫번째 부분은 재귀 쿼리를 사용하던 부분을 JOIN 쿼리로 해결할 수 있게 되었다는 점이었고, 두번째 부분은 지나치게 통합되어 nullable 을 허용해야만 하던 필드들에게 NOT NULL 조건을 붙일 수 있게 되었다는 점이었습니다.

5) 재설계한 스키마로 기존 데이터 마이그레이션

하지만 이렇게 새로 설계한 스키마로 기존 데이터를 마이그레이션 하려니 형태가 달라진 만큼 굉장히 복잡한 마이그레이션 로직이 필요하게 되었습니다. 마이그레이션을 쿼리로 짰다가는 검증이 무척 힘들 것 같아서, JPA 로 마이그레이션 로직을 작성하고 테스트 코드로 검증하였습니다. 단, JPA 로 대규모 INSERT 를 할 경우, 해당 엔티티가 isNew 인지 확인하여 persist 혹은 merge 를 하는 비효율이 발생하였고 rewriteBatchedStatements 옵션만으로는 해결이 되지 않아서, Persistable 인터페이스를 상속해 isNew 가 항상 true 이도록 하였습니다. 이렇게 하니 1 시간 넘게 걸리던 마이그레이션 배치 작업이 10 분 정도로 줄어들게 되었고, 고객에게 30 분 정도의 점검시간을 가지겠다고 공지한 후 마이그레이션을 진행할 수 있었습니다.

6) 모니터링 및 로깅 시스템 구축

DataDog 과 같은 서비스 모니터링 솔루션은 경영진이 돈을 아끼고 싶어했기에 도입하지 못했습니다. 헬스 체크와 메트릭 이상은 CloudWatch 를 통해 알람이 울리도록 할 수 있었지만, 디버깅을 위해서라도 API 비정상 호출 시의 Request 와 Response 는 로그를 남길 필요가 있다고 생각했습니다. 처음에는 API 비정상 호출 로깅을 AOP 를 통해 구현했습니다. AOP 를 처음 선택한 이유는 Audit 기능을 사용하기 편했고, Request 및 Response 를 ContentCachingWrapper 로 감싸는 번거로움이 없었기 때문입니다. 하지만 AOP 를 통한 로깅은 Spring Security 의 필터를 통해 비정상 종료되는 로그를 기록하지 못했고, 그렇기에 AOP 대신 로깅을 하는 필터를 직접 구현해 필터 체인에 끼워 넣었습니다. API 요청 응답에 관한 로깅은 데이터 양이 많았고, 실서비스에 영향이 가지 않도록 별개의 DB 서버에 저장하도록 했습니다. 1 차로 DB 에 저장된 로그는 2 주가 지나면 배치작업을 통해 S3 로 옮겨져서 용량으로 인한 비용이 발생하지 않도록 하였습니다.

7) 독단적이지 않은 의사결정

저는 현재 회사에서 근무하며 자랑스러운 점이 두가지 있습니다. 하나는 신규 기능 개발을 하느라 바쁜 와중에도 리팩터링에 대한 욕심을 포기하지 않았다는 점입니다. 설령 잠을 4 시간 이하로 줄이는 한이 있더라도, 외주 수준의 개발 품질에 남아있고 싶지 않았습니다. 두번째는 이 모든 리팩터링을 저 혼자의 독단으로 진행한 게 아니라, 개발을 잘 모르는 경영진을 설득하며 진행했다는 점입니다. 개발 용어로 경영진과 이야기를 한 게 아니라, DAU 와 같은 투자에 있어서 중요한 KPI 수치를 통해 버그를 잡아야할 필요성과 서버가 죽는 불안정성을 해결할 필요를 설득했습니다. 서버가 몇 시간만 죽어도 그날의 DAU 수치가 굉장히 나쁘게 나오기 때문에, 이중화와 인프라 구축에 들어가는 리소스의 필요성을 경영진에게 설득할 수 있었습니다. 버그는 VOC 를 통해 고객 평가가 나쁘다는 점을 경영진에게 알렸고, 버그가 재발하지 않기 위해서는 테스트 코드에 시간을 써야만 한다고 경영진을 설득했습니다. 수면 시간이 부족했지만, 신규 기능 개발과 리팩터링을 병행하며 3 년간 서비스를 운영했습니다. 그리고 저의 지식과 경험이 부족한 일이 생기면, 혼자서 해결하는 게 아니라 NEXTSTEP 과 같은 커뮤니티에 참여하여 적극적인 스터디를 통해 실무의 문제를 해결하였습니다.

외부 링크

- 개인 블로그 글
  - 동료와 성장에 관한 생각
    - https://velog.io/@pyro/comrade
  - 리뷰와 피드백에 관한 생각
    - https://velog.io/@pyro/review-guide

- 코드 리뷰 예시
  - 테스트와 예외처리
    - https://github.com/woowacourse/kotlin-lotto/pull/93#issuecomment-1970238649
  - 람다와 함수형 프로그래밍
    - https://github.com/woowacourse/kotlin-omok/pull/55#discussion_r1535278325