송재민, 백엔드 개발자

안녕하세요! 본질을 꿰뚫는 개발자 송재민입니다.

[특정 기술이 쓰이는 이유를 정확히 파악합니다]

남에게 설명하지 못하면 개념을 알고 있는 것이 아니라고 생각합니다.

예를 들어 교내 웹 프로그래밍 동아리 회장으로서 멘토링 및 28회의 세미나를 진행하며 Spring의 본질을 파악하였습니다.

앞으로도 모든 원리와 근거를 파악하고 활용하는 개발자가 되려고 합니다!

Contact

Github: https://github.com/jeli01
Email: song98.dev@gmail.com
Blog: https://aole.tistory.com/
Phone: 010-8742-7322

Skills

JAVA / SPRING / JPA / MYSQL / Redis / Kafka / Git

About Me

1. 끊임 없이 성장합니다.

2025년 02월 05일 부터 "하루도 빠짐없이" til을 작성
그날 배운 내용이나 깨달은 점을 매일 기록하고 회고
하루를 어떻게 보냈는지 기록하고 되돌아보며 미래를 계획

2. 커뮤니케이션 능력이 뛰어납니다.

웹 프로그래밍 동아리 회장으로서 조직 운영 및 리더십 경험
학술 스터디를 3개 이상 운영한 경험
5개 이상의 동아리 활동을 통해 다양한 성향의 사람들과의 협업 능력 개발

Project 1 - 스펙랭킹

소개

서비스 설명: 사용자의 스펙 점수를 제공하는 서비스
개발 기간 : 2025.05~2025.06 (2개월)
개발 인원 : 7명 (백엔드 2명, AI 2명, 클라우드 3명)
담당 역할 : 백엔드

Github

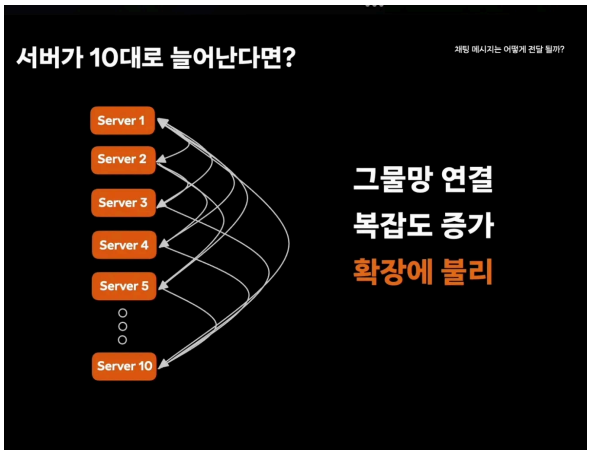
Main Server: https://github.com/100-hours-a-week/19-Respec-BE/tree/main
Consumer Server: https://github.com/100-hours-a-week/19-Respec-BE-Chatconsumer

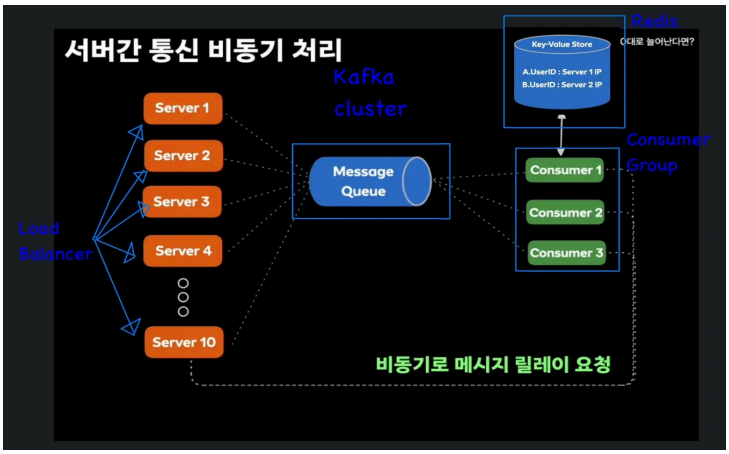
주 기술 스택

Websocket, Spring, JPA, MySQL, Redis, Kafka, Git

기여한 부분

1. 분산 환경에서 동작하는 **Chat Server** 구축





[도입 배경]

서버가 분산 되었을 때 1:1 채팅 시, 상대방이 접속해 있는 ec2에 메세지를 보내야 하는 상황이 발생했습니다. 서버가 늘어날 수록 아키텍처가 복잡해지는 문제가 발생했습니다. 이를 외부 브로커를 이용해 해결했습니다. 또한 저희는 **MSA** 아키텍처로의 확장 가능성도 열어둔 상태였습니다. 대용량 데이터 스트림 처리에 있어 성능이나 확장성 측면을 고려해 **Kafka**를 선택했습니다.

[트러블 슈팅]

1) 사용자의 비정상적인 종료 문제

사용자가 컴퓨터 강제종료 등. 비정상적인 종료를 할 경우 두가지 문제점이 발생할 수 있었습니다. 첫번째는 스프링 서버가 사용자의 종료를 감지하지 못해 **websocket session**을 정리하지 못하는 문제입니다. 이를 **30**초 마다 **ping-pong** 로직을 구현함으로써 해결하였습니다. 두번째는 redis 메모리 누수입니다. redis에는 채팅에 참여한 사용자의 서버의 ip 주소가 저장됩니다. 비정상적인 종료시 **redis**에 해당 유저의 값이 남아 있게 됩니다**.** redis에 해당 값이 있으면 '채팅' 없으면 '알림'을 보내는데 값이 존재하므로 알림이 가지 않고 '채팅' 요청을 보내게 됩니다. 따라서 릴레이 요청을 받는 서버에서 클라이언트로 **send**를 보냈을 때 **Send Error**가 발생했을 경우 **Consumer**의 잘못된 요청으로 판단하고 **redis**의 값을 삭제**,** 알림을 생성함으로써 이를 해결하였습니다.

2) kafka exactly-once 문제

카프카에서 "정확히 한번" 메세리를 처리하기 위해 2가지를 고려하였습니다. 첫번째는 프로듀서에서의 exactly-once 입니다. 이는 메세지 마다 고유의 **PID**와 **sequence number**를 함께 보내는 **enable.idempotence** 옵션으로 보장하였습니다. 두번째는 컨슈머에서의 exactly-once 입니다. 카프카에서 리밸런싱이 일어나는 등 이유들로 중복 데이터가 발생할 수 있습니다. 이를 **producer**가 보내는 **uuid**로 판단하고 멱등을 보장하여 해결하였습니다. 대안으로 primary key를 uuid로 전환하고 upsert 로직을 통해 멱등성을 보장할 수 있었습니다. 다만 이 방법은 uuid의 성능상 문제, 메모리 문제로 위의 방법을 채택하였습니다.

3) 순수 웹 소켓으로 구현

Stomp를 사용하지 않고 순수 **Websocket**으로 분산 서버에서의 채팅 서버를 구성하였습니다. 그 이유는 세가지입니다. 첫째 Kafka는 Stomp를 지원하지 않아 Stomp를 사용한다면 스프링의 인메모리 브로커를 거쳐 외부 브로커로 가는 **2**중 브로커 아키텍처가 구성되기 때문입니다. 둘째 2중 브로커 아키텍처를 구성한다고 가정했을때, 메인 서버는 producer와 consume 역할을 두개 하게 됩니다. 자신의 사용자에게 오는 채팅인지 알 수 있는 방법이 없으므로 모든 **chat**을 **consume**해야 하는 문제가 발생합니다. 셋째 모든 chat을 consume하지 않고 chatroom 마다 토픽을 생성하는 방식을 활용하면 자신의 chatroom에 관한 chat만 받아볼 수 있습니다. 다만 그렇게 될 시 채팅방 수 만큼 **topic**이 생성되어야 하므로 불필요하게 많은 topic 수가 예상되었습니다. 따라서 **producer, consumer**를 분리하는 위의 아키텍처를 채택하게 되었습니다.

【성과】

1) 분산 환경에서의 실시간 통신 안정성 보장
2) MSA 확장성을 고려한 확장 가능한 채팅 시스템 설계
3) 비정상 종료로 인한 채팅 메시지 누락 문제 해결
4) Kafka exactly-once 메시지 신뢰성 확보
5) 확장 가능한 아키텍처 구축으로 향후 트래픽 증가에 대비

2. RankingBoard 성능 개선

【도입 배경】

SpecRanking 서비스는 사용자의 스펙 랭킹을 보여주는 서비스 입니다. 사람들은 다른 사용자들의 랭킹을 조회합니다. 이 부분에 대한 리소스가 가장 많이 소요될 것으로 예상했습니다. 따라서 이 부분을 개선합니다.

【트러블 슈팅】

1) 커서 기반 페이지 네이션

만약 모든 스펙을 가져와 랭킹 보드에 띄운다면 리소스가 많이 들 것으로 예상하였습니다. 이에 페이지네이션을 고려하였습니다. 무한 스크롤이 요구사항이었기 때문에 커서 기반 페이지네이션과 오프셋 기반 페이지네이션을 선택할 수 있었습니다. 오프셋 기반 페이지네이션의 경우 중간에 데이터가 들어올 경우 중복으로 사용자에게 데이터가 보여지는 경우가 있을 수 있습니다. 따라서 커서 기반 페이지네이션으로 구현 하였습니다.

2) 레디스 캐싱

TOP 10 랭킹 보드의 경우, 홈 화면 및 랭킹 화면에서 보여지게 됩니다. 비로그인 사용자, 로그인 사용자도 해당 내용을 볼 수 있기 때문에 레디스 캐싱하였습니다. 다만 이는 캐시 스탬피드가 발생할 수 있었습니다. **ttl**이 끝나는 순간 모든 사용자들이 **db**에 접근하여 **db** 부하가 생기는 문제입니다. 이를 **PER** 알고리즘을 활용하여 해결하였습니다. 또한 스펙 상세 조회의

경우에도 많은 사람들이 보는 스펙은 고 스펙으로 정해져 있기 때문에 ttl을 활용하여 캐싱하였습니다. 여기에는 캐시 페네트레이션이 발생할 수 있었습니다. 스펙 api는 id로 조회를 할 수 있습니다. 만약 사라진 스펙, 없는 스펙을 id로 조회를 하게 된다면 캐싱이 되지 않아 **db** 부하가 생기는 **"**공격**"**과 같은 문제가 생길 수 있었습니다. 이를 **null** 캐싱을 활용하여 해결하였습니다.

3) 쿼리 성능 향상

캐싱으로 인한 성능 향상 외에도 근본적인 쿼리 성능 향상을 실시했습니다. 분석 결과 2가지 문제점을 발견했습니다. 첫째, 10개의 스펙을 하나하나 쿼리하는 문제입니다. RankingBoard는 커서 기반 페이지 네이션으로 10개씩 스펙에 관련된 필드를 가져오게 됩니다. 이를 **group by** 절을 활용해 한번에 가져오도록 개선하여 **22.6%** 성능 개선 하였 습니다. 둘째, User 테이블을 기준으로 Spec 테이블을 조인하여 스펙 보유 유저 수를 계산하는 비효율적인 쿼리 구조 문제입니다. 이를 Spec 테이블에서 DISTINCT user_id로 쿼리하도록 변경하여 불필요한 조인 연산을 제거해 **40%** 성능 개선하였습니다.

4) 프론트의 bookmark 상태 관리

랭킹 board에서 해당 랭킹에 대해 bookmark를 했는지에 대한 여부가 프론트엔드에서 띄워지게 됩니다. 따라서 해당 부분은 캐싱을 진행할 수 없어 **redis**가 아닌 **db**에서 가져오게 됩니다**.** 이 부분을 로그인 할 때 프론트엔드에서 사용자가 즐겨찾기 한 부분을 api로 모두 가져오고 상태관리를 하는 방식으로 성능 개선 하였습니다. 프론트엔드 단에서 무엇이 즐겨찾기 되었는지 관리하고 있기 때문에 백엔드에서 즐겨찾기에 대한 부분을 json으로 내려줄 필요가 없어 **db**에 접근하는 일이 없어졌습니다**.** 즉 캐싱 되지 않는 TOP10 랭킹의 데이터는 존재하지 않게 되었습니다.

[성과**]**

1) 커서 기반 페이지네이션으로 데이터 중복 노출 문제 완전 해결
2) Redis 캐싱으로 TOP10 랭킹 조회 성능 97.7% 향상
3) PER 알고리즘 적용으로 캐시 스탬피드 문제 해결, DB 부하 감소
4) group by 절을 활용한 쿼리 최적화로 22.6% 성능 개선
5) 조인 연산 최적화로 추가 40% 성능 개선
6) 프론트엔드 상태관리 최적화로 불필요한 DB 접근 완전 제거

3. 테스트 코드의 **"**신호**"** 감지 후 리팩토링

[도입 배경**]**

테스트 코드를 작성할 때, 짜기 어려운 경우가 생깁니다. 이는 본 코드를 리팩토링 하라는 테스트 코드의 "신호"입니다. 이를 받아들이고 리팩토링을 진행합니다.

[트러블 슈팅**]**

ChatRelayService는 알림을 저장하는 기능, Relay 요청을 보내는 기능을 가지고 있었습니다. 해당 기능에 대해 구체적인 비즈니스 로직을 의존하고 있었습니다. 이에 테스트 하기에 어려움을 겪었습니다. 이를 테스트 코드가 **"**리팩토링 하라는 신호**"**를 저에게 보내는 것으로 인식 하였습니다. 따라서 이를 **ChatDeliveryService, ChatRelayService, NotificationService**로 분리하였습니다. 이로써 테스트 코드 작성에 용이함을 가져왔습니다. 또한 클래스를 분리함으로써 **SRP**를 지켜낼 수 있었습니다. 또한 테스트 코드를 작성하며 에러 처리에 관해 부족함이 있는 코드임을 깨달았습니다**.** Kafka 설정으로 ErrorHandler를 적용하였지만 webClient에서 나오는 에러는 처리하지 못하기 때문입니다. 이것을 테스트 코드가 알려주는 **"**추가 개발 신호"로 캐치하고 webClient에서 Error가 발생한다면 Dlq에 해당 내용을 프로듀스 하도록 개발하고 테스트 하였습니다.

[성과**]**

1) SRP 적용으로 클래스 응집도 향상 및 결합도 감소

2) 1개 클래스를 3개로 분리하여 각 클래스의 책임 명확화

Project 2 - DevTalk (개인 프로젝트)

소개

서비스 설명: IT 개발자들에게 소통 공간을 제공하는 서비스
개발 기간 : 2025.03~2025.03 (1개월)
개발 인원 : 1명 (풀스택 1명)
담당 역할 : 풀스택

Github

https://github.com/100-hours-a-week/2-jelly-song-community

주 기술 스택

JavaScript, Spring, JPA, MySQL, AWS(EC2, RDS, ELB, ECR, CodeDeploy), Git

기여한 부분

1. 퍼블리싱 라이브러리 개발 후 활용 (블로그 링크 : https://aole.tistory.com/52)

[도입 배경]

카카오테크 부트캠프 개인 프로젝트에서 순수 HTML, CSS, JavaScript로 커뮤니티 사이트를 개발해야 했습니다. 템플릿 엔진 사용이 제한된 상황에서 코드의 가독성과 재사용성이 떨어지는 문제가 발생했습니다. 로버트 마틴의 클린코드 원칙인 **"**신문 읽듯이 코드를 작성해야 한다**"**는 개념을 HTML에도 적용하고 싶었습니다. Css 라이브러리인 **BootStrap**과 **Spring**의 **Thymeleaf**와 같은 기능을 순수 **JavaScript**로 구현하여 **HTML**의 추상화 수준을 높이고자 했습니다.

[트러블 슈팅]

첫째, HTML 컴포넌트 분리 전략을 수립했습니다. 기존 하나의 HTML 파일에 모든 코드가 섞여 있던 구조를 header, main-container, footer 등으로 분리하여 각각 독립적인 HTML 파일로 관리했습니다. 둘째, includeHTML 함수를 개발했습니다. include-html 속성을 가진 태그를 탐지하고 fetch를 통해 해당 경로의 **HTML**을 동적으로 로드하는 라이브러리를 구현했습니다. 이를 통해 간단하게 컴포넌트를 재사용할 수 있게 되었습니다. 셋째, 비동기 로딩 문제를 해결했습니다. **HTML**이 로드된 후에 **JavaScript**가 실행되어야 하는 의존성 문제를 콜백 패턴으로 해결했습니다. includeHTML 완료 후 콜백에서 동적으로 script를 추가하여 DOM 조작 코드가 정상적으로 실행되도록 보장했습니다.

[성과]

1) 단일 HTML 파일을 컴포넌트 단위로 분리하여 코드 가독성 향상, 중복 감소
2) "include-html" 속성 한 줄로 여러 컴포넌트 삽입 가능
3) 클린코드 원칙을 HTML에 적용한 새로운 접근법 창안

2. 보안성을 고려한 **Auth** 구축

[도입 배경]

일반적인 **JWT** 인증 방식에서는 refresh 토큰이 탈취될 경우 장기간 악용될 수 있는 보안 취약점이 존재했습니다. 또한 토큰을 어디에 저장할지에 대한 보안 고려사항도 있었습니다. 로컬 스토리지는 XSS 공격에 쿠키는 CSRF 공격에 취약한 특성을 가지고 있어 각 토큰의 특성에 맞는 최적의 저장 방식과 보안 전략이 필요했습니다.

【트러블 슈팅】

1) Refresh Token Rotation 구현

기존 **refresh** 토큰이 한 번 탈취 된다면 만료까지 계속 사용되는 문제를 해결하기 위해 **RTR** 방식을 도입했습니다. refresh 토큰으로 새로운 access 토큰을 요청할 때마다 새로운 refresh 토큰도 함께 발급하여 이전 refresh 토큰을 무효화하는 방식으로 구현했습니다. 이를 통해 토큰이 탈취되더라도 피해 범위를 최소화할 수 있었습니다.

2) 토큰별 차별화된 저장 전략

Access 토큰과 refresh 토큰의 특성을 분석하여 각각 다른 저장 방식을 적용했습니다. **Access** 토큰은 Header로 전송하고 프론트엔드에서 로컬 스토리지에 저장하도록 했습니다. 생명주기가 짧아 탈취되어도 피해가 적고**,** 다양한 **API** 요청에 사용되므로 **CSRF** 공격을 방어하기 위함입니다. **Refresh** 토큰은 쿠키로 전송하고 **HttpOnly** 설정을 적용했습니다. **XSS** 공격을 방어할 수 있고, 탈취되어도 공격자가 할 수 있는 행위가 토큰 재발급뿐이므로 실질적 피해가 제한적 입니다.

3) 만료된 토큰 정리 로직

서버 메모리 및 데이터베이스 리소스 관리를 위해 스케줄링 기반의 토큰 정리 로직을 구현했습니다. 주기적으로 만료된 refresh 토큰을 탈취하고 삭제하여 불필요한 데이터 축적을 방지했습니다. 이를 통해 시스템 성능을 유지하고 보안성을 강화할 수 있었습니다.

【성과】

1) RTR 도입으로 토큰 탈취 시 피해 범위 최소화
2) 토큰별 차별화 저장 전략으로 XSS/CSRF 공격 원천 차단
3) 만료 토큰 정리 스케줄링으로 메모리 사용량 최적화

3. 반정규화

【도입 배경】

게시물 목록에서 모든 게시물 들이 좋아요수를 조인 쿼리로 가져와 **db** 부하가 발생

【트러블 슈팅】

1) 단계별 성능 최적화 접근법 적용

데이터베이스 성능 최적화의 일반적인 프로세스를 따라 단계적으로 접근했습니다. 먼저 제 3정규화까지 진행된 설계에서 데이터 무결성을 최대한 보장하고, 성능 문제가 발생한 시점에서 인덱스 최적화를 시도했습니다. 하지만 게시물 목록 조회라는 빈번한 읽기 작업에서는 인덱스만으로는 한계가 있었습니다.

2) 선택적 역정규화 전략 수립

모든 테이블을 역정규화하는 것이 아닌, 성능 병목이 발생하는 특정 부분에 대해서만 선택적으로 역정규화를 적용했습니다. Board 테이블에 like_count 필드를 추가하여 실시간 **JOIN** 쿼리를 제거했습니다. 이를 통해 게시물 목록 조회 시 단순한 **SELECT** 쿼리로 변경되어 성능이 대폭 개선되었습니다.

3) 데이터 무결성 보장 매커니즘 구현

역정규화로 인한 데이터 불일치 문제를 방지하기 위해 좋아요 생성/삭제 시 Board 테이블의 **like_count**를 동기화하는 로직을 구현했습니다. 트랜잭션을 활용하여 **Like** 테이블 변경과 **Board** 테이블 업데이트가 원자적으로 처리되도록 보장했습니다. **ddd**를 이용한 원자적 처리도 고려하였지만, 연관관계 주인을 변경하거나 양방향 매핑을 도입할 경우 유지보수성이 저하될 위험이 있어 트랜잭션 기반 접근 방식을 채택했습니다.

【성과】

1) JOIN 쿼리 제거로 게시물 목록 조회 성능 향상
2) 선택적 역정규화 전략으로 성능과 데이터 무결성 동시 확보
3) 원자적 동기화 로직을 구성하여 데이터 일관성 보장

4. 고가용성 클라우드 인프라 설계 및 **CI/CD** 자동화

【도입 배경】

기존 프로젝트에서는 단일 서버 환경에서 수동 배포를 진행하고 있어 서버 장애 시 전체 서비스 중단, 트래픽 급증 시 확장 대응 불가, 수동 작업으로 인한 인적 오류 및 시간 소모가 큰 문제였습니다. 또한 직접적인 서버 접근으로 인한 보안 위험과 실시간 모니터링 체계 부재로 인한 장애 감지 지연이 발생할 수 있었습니다. 이러한 문제를 해결하기 위해 멀티 AZ, Auto Scaling, CloudWatch, Bastion Host, RDS standby를 포함한 고가용성 **AWS** 인프라와 완전 자동화된 **CI/CD** 파이프라인을 구축했습니다.

【트러블 슈팅】

1) 멀티 AZ 고가용성 구조 설계

단일 장애점 제거를 위해 두 개의 가용 영역에 인프라를 분산 배치했습니다. Public Subnet에는 ALB와 Bastion Host를, Private Subnet에는 애플리케이션 서버를 격리 배치하여 보안성을 강화했습니다. RDS Multi-AZ 구성으로 Primary DB 장애 시 Standby로 자동 페일오버되어 데이터 손실 없이 서비스 지속이 가능하도록 구성하였습니다.

2) Auto Scaling 및 로드 밸런싱 구현

트래픽 변화에 대응하기 위해 Auto Scaling Group을 구성했습니다. Application Load Balancer로 트래픽을 분산하고, Target Group의 헬스 체크를 통해 장애 인스턴스를 자동 격리하여 트래픽 급증 시 자동 확장과 서비스 안정성을 보장했습니다.

3) CloudWatch 기반 모니터링 체계 구축

Spring Boot 로그를 CloudWatch Logs로 수집한뒤 EventBridge Rule을 통해 S3에 보관하고, ERROR 발생 시 Discord 실시간 알림을 보내도록 구현했습니다. 메트릭 기반 모니터링은 CPU/Memory 기반 CloudWatch 알람으로 임계치 초과 시 Discord 알림과 Auto Scaling 자동 확장을 동시에 트리거하여 실시간 장애 대응과 서비스 안정성을 보장하였습니다.

4) Docker 기반 CI/CD 파이프라인 구축

Github Actions, Docker, ECR, CodeDeploy를 활용한 자동화 파이프라인을 구성했습니다. Docker 컨테이너화로 환경 일관성을 보장하고, ECR에서 이미지를 중앙 관리하며, CodeDeploy를 통해 무중단 배포를 실현했습니다.

【성과】

1) 멀티 AZ 구성으로 고가용성 서비스 환경 구축 및 데이터 손실 위험 완전 제거

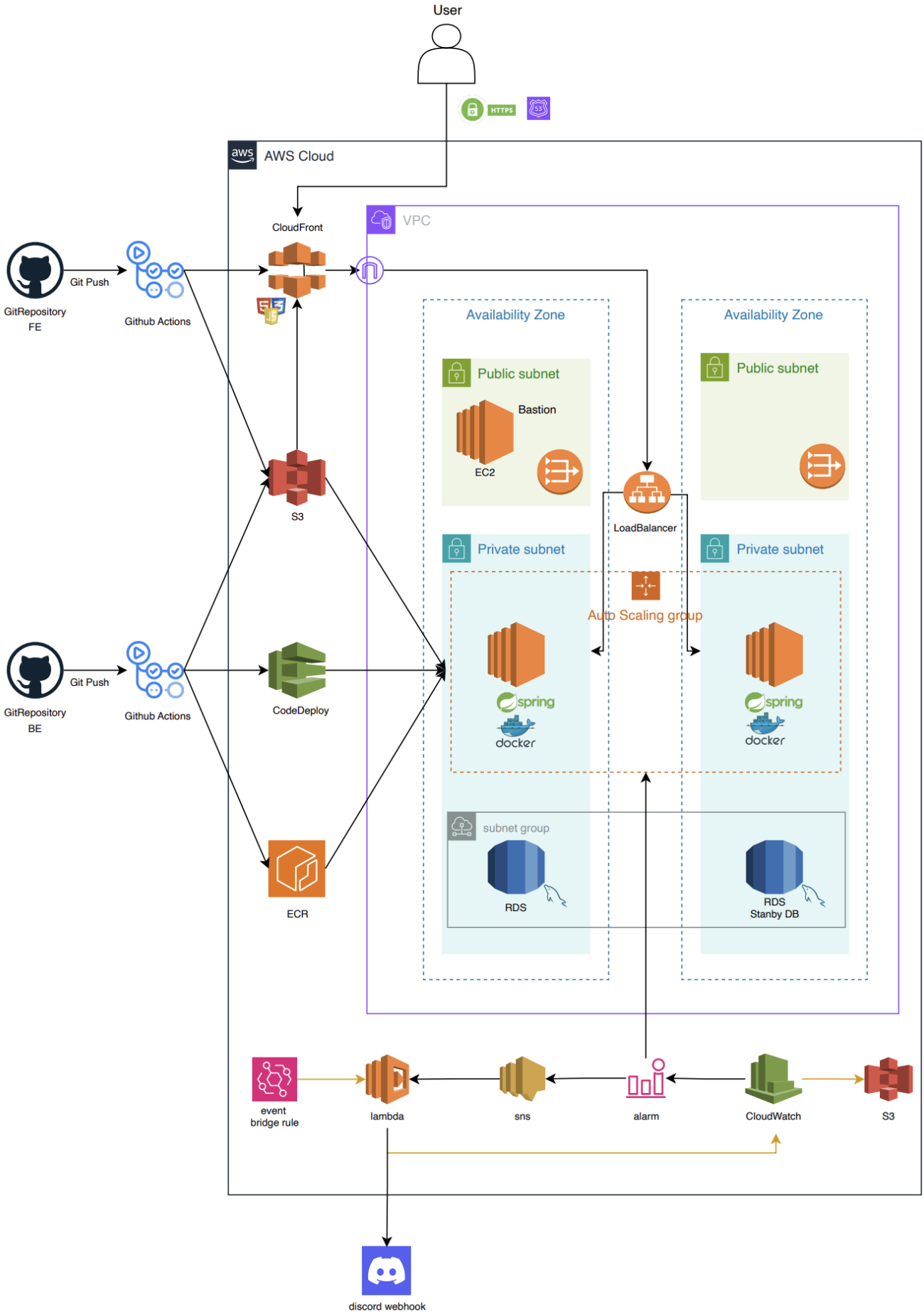
2) Auto Scaling 적용으로 트래픽 변화에 따른 자동 확장 및 운영 비용 최적화

3) Bastion Host와 Private Subnet 격리로 보안 강화 및 무중단 배포를 통한 CI/CD 자동화로 배포 효율성 대폭 향상

4) CloudWatch 기반 실시간 에러 감지 및 자동 확장으로 장애 대응 시간 단축과 서비스 안정성 향상

5) 확장 가능한 고가용성 아키텍처 구축으로 안정적인 서비스 운영 및 향후 성장 대비 인프라 완성

【아키텍처】


User
HTTPS
AWS Cloud
CloudFront
VPC
Git Push
GitRepository
FE
Github Actions
Availability Zone
Availability Zone
Public subnet
Bastion
EC2
Public subnet
S3
LoadBalancer
Private subnet
Private subnet
Auto Scaling group
Git Push
GitRepository
BE
Github Actions
CodeDeploy
spring
docker
spring
docker
subnet group
RDS
RDS
Stanby DB
ECR
event
bridge rule
lambda
sns
alarm
CloudWatch
S3
discord webhook

Study

1. 스프링 파헤치기

스프링 역사를 탐구한다.

1. **spring di container**를 직접 구현한다. 이를 수동 빈 등록, 컴포넌트 스캔으로 점진적 마이그레이션 한다.

2. db vender에 종속적인 connection에서 부터 QueryDSL까지 변화한 이유를 모두 알아 낸다. 또한 트랜잭션 매니저의 장점을 이해하기 위해 직접 **connection**을 파라미터로 전달하며 구현해 본다. 후에 트랜잭션 매니저를 이용해 마이그레이션 한다.

3. java servlet에서 부터 어노테이션 스프링 mvc까지 변화한 이유를 모두 알아 낸다. 이 역사를 기반으로 점진적 마이그레이션을 진행한다.

4. 각각의 기술들의 동작 원리를 이해한다. 또한 위의 역사를 가지게된 이유를 이해한다. 그 내용을 부원들에게 세미나 발표 진행한다.

2. 클린코드 (로버트 마틴)

클린 코드 원칙을 학습하고 자바 **SerialDate** 클래스를 리팩토링 진행한다.

1. 네이밍과 함수 설계

의도를 밝히는 변수명, 서술적인 함수명을 사용한다. 함수는 작게 만들고 한 가지 일만 하며 하나의 추상화 수준을 유지한다. 함수 인수는 최소화하고 플래그 인수를 피한다. 보이 스카우트 규칙을 적용하여 체크인할 때마다 코드를 조금씩 개선한다.

2. 주석과 코드 표현력

'주석은 나쁜 코드를 보완하지 못한다'는 원칙을 이해한다. 코드로 의도를 표현하는 방법을 학습하여 불필요한 주석을 제거하고 자기 설명적인 코드를 작성한다.

3. 예외 처리와 안전성

오류 코드 대신 예외를 사용하여 비즈니스 로직과 오류 처리를 분리한다. Try/Catch 블록을 분리하고 예외에 의미 있는 정보를 제공한다. **null** 반환/ 전달을 지양하고 빈 컬렉션 반환 등을 통해 안전한 코드를 작성한다. **(**이는 스펙랭킹 프로젝트에서 **AI** 서버로의 **json** 전달에서 유용성을 경험한다**.)**

4. 클래스와 구조 설계

단일 책임 원칙을 적용하여 작은 클래스를 만든다. 인스턴스 변수를 **3**개 이하로 적게 유지하여 응집도을 높이고, 추상화된 것에 의존하여 변경으로부터 격리한다. 중복을 제거하고 개념은 빈 행으로 분리하며 밀접한 개념은 세로로 가까이 배치한다.

3. 우아한 테크코스 프리코스 (github: https://github.com/jeli01/woowa-course-package)

프리코스 **4**주차를 통해 객체 지향 설계와 클린 코드를 경험한다.

1. 객체 지향 설계 원칙

setter, getter 사용을 줄이고 '객체에게 일 시키기' 개념을 적용한다. 디미터 법칙을 학습하여 객체 간 협력 방식을 개선한다. 원시값 포장을 통한 자동 검증 시스템을 구축한다. Collections getter 반환 시 immutable 변환을 적용하는등, 객체의 상태를 보호하면서도 필요한 정보는 제공하는 방법을 구현한다. 원시값 포장과 일급 컬렉션 개념을 도입하여 객체에 제약조건을 부여한다.

2. 테스트 범위 최적화

모든 메서드 테스트의 비효율성을 깨닫고 도메인 핵심 로직 중심으로 전환한다. CRUD나 화면 출력, 구체적인 도메인 로직은 테스트 하지 않는다. 클라이언트가 원하는 **api** 주위로 테스트 진행한다. 일반적인 경우와 경계값 위주로 테스트 진행한다.

3. 빠른 개발 후 리팩토링 vs 사전 설계 장단점 경험

협업과 프로젝트 규모에 따른 적정 고민 시간의 균형점을 찾는다. 이는 스펙랭킹 프로젝트에서 **"**테크 스펙**"** 작성 경험으로 이어진다.

4. TDD

테스트 주도 개발의 **Red-Green-Refactor** 사이클을 체득한다.

1. TDD 사이클 체화

실패하는 테스트 작성 -> 최소한의 코드로 통과 -> 리팩토링 과정을 반복하며 점진적으로 기능을 완성한다. 암호 검사기 구현을 통해 **TDD**의 핵심 흐름을 경험한다.

2. 테스트 작성 전략

쉬운 경우에서 어려운 경우로, 예외 상황에서 정상 상황으로 진행하는 전략을 학습한다. 초반 복잡한 테스트를 피하고 구현하기 쉬운 테스트부터 시작하여 빠른 피드백 루프를 구축한다.

3. 테스트 가능한 설계

하드코딩된 값, 의존 객체 직접 생성, 정적 메서드 사용 등 테스트 어려운 코드를 식별한다. 전략 패턴을 이용하여 Random, LocalDate.now() 등 실행 시점에 따라 달라지는 결과를 테스트 가능하게 설계한다.

4. 대역 활용

Stub, Mock, Spy 등 대역의 종류와 쓰임새를 이해한다. 외부 의존성 제거를 통해 개발 속도를 향상시키고 의존 대상 구현 없이도 테스트 대상을 완성할 수 있음을 경험한다. 이는 스펙랭킹 프로젝트에서 **"Ai, S3, Auth mock** 서버 구축**"** 경험으로 이어진다.

5. 테스트 범위와 유지보수

단위 테스트**,** 통합 테스트**,** 기능 테스트의 차이점을 이해한다. 과도한 구현 검증을 피하고 변수나 필드를 사용한 기댓값 표현을 지양하여 테스트 코드 자체의 유지보수성을 확보한다.

ETC

사소한 부분도 기술을 고민하고 도입합니다.

DTO Lombok vs Record 기술 선택 과정

1) @Data 선택

DTO에 필요한 필수 기능이 모두 포함되어 있음

디버깅 시 ToString을 통한 편리한 로깅 가능

비즈니스 로직이 없는 DTO에서는 Setter 사용도 문제 없다고 판단

2) 생성자 + Getter 방식으로 전환

불변성 유지를 위해 Setter 제거 결정

Setter가 없음으로써 안정감을 느낄 수 있다는 확신

Record 대신 Lombok 선택 이유: getXXX 메서드가 Java 개발자에게 더 친숙

3) Record로 마이그레이션

JSON 직렬화/역직렬화 시 @JsonValue, @JsonCreator 필요성 발견

생성자 방식은 이를 자동 지원하지 않음을 확인

Record의 장점 재평가: DTO를 위해 설계된 언어 차원의 지원, JSON 처리 자동 지원, ToString 기본 제공, 최신 개발 트렌드에서 getXXX 없는 접근도 많이 사용됨.

결론: 보일러플레이트 제거뿐만 아니라 **JSON** 처리**,** 언어 차원의 지원**,** 개발 트렌드까지 종합적으로 고려하여 **Record** 최종 선택

Community

세종대학교 프로그래밍 학술동아리 아롬 **14**기 **/** 부원 **(2024.09 ~ 2024.12)**

- 클린코드 스터디 운영
  자바 **SerialDate** 클래스를 직접 리팩토링하는 경험.

세종대학교 웹 프로그래밍 학술동아리 **OpenYearRound 11**기 **/** 회장 **(2023.03 ~ 2023.12)**

- 동아리 운영, 스터디 자료 및 과제 기획, 코드 리뷰
- 웹 프로그래밍 지식 강의, 프로젝트 멘토링

세종대학교 웹 프로그래밍 학술동아리 **OpenYearRound 10**기 **/** 부원 **(2022.03~2022.12)**

- 웹 개발 스터디 및 학술제 참여

Award

- 세종대학교 2023 제16회 창의설계 경진대회 대상 (23.12.01)
- 제6회 SW코딩경시대회 세종코딩챌린지위크 4등 (19.12.04)

Certification

- 정보처리기사
- SQLD

Education

세종대학교 소프트웨어학과 졸업 **(2019.03 ~ 2025.02)**
- 전공 학점: 4.12/4.5
- 전체 학점: 4.00/4.5

카카오테크 부트캠프 판교 **2**기 풀스택 과정 **(2025.01 ~ 2025.06)**

삼성청년**SW·AI**아카데미 **14**기 **(2025.07 ~)**