Portfolio

버린 이유까지 짚어,
고른 이유를 분명히 합니다

김민영 백엔드 개발자

이력 사항



김민영

백엔드 개발자

1995.12.01
010-4104-6850
miny20023@gmail.com

경력

- 2022.06 ~ 2025.09 주식회사 헤딧 플랫폼 개발부

2025.01 ~ 2025.09	**인생시계**	백엔드
2023.11 ~ 2025.08	**톡메모**	백엔드
2024.08 ~ 2025.01	**엑서플랫폼**	풀스택
2024.04 ~ 2024.09	**마음알림**	백엔드
2023.02 ~ 2023.12	**비콕**	풀스택
2023.05 ~ 2023.09	**맘잡고**	프론트엔드
2022.07 ~ 2022.12	**비런치**	프론트엔드

교육

- 2026.05 ~ 2026.07 Spring AI와 MSA를 활용한 백엔드 개발
  그렙(프로그래머스)
- 2021.03 ~ 2021.08 빅데이터 분석 전문가 양성 과정 글로벌 IT 인재 개발원
- 2019.01 ~ 2019.02 리눅스 기초 코리아 IT 아카데미
- 2019.01 ~ 2019.02 JAVA 기초 코리아 IT 아카데미
- 2014.03 ~ 2021.02 선문대학교 컴퓨터공학과

기술

언어	Java, Kotlin, JavaScript, TypeScript, SQL
프레임워크/라이브러리	Spring Boot, Spring Security, Spring WebFlux, Spring Cloud Gateway, Kotlin Coroutines, JPA/Hibernate, QueryDSL, React, Redux, Zustand
DB/캐시	PostgreSQL, Redis
통신/메시징	Socket.IO, SSE, Kafka
API/SDK	StoreKit, Google Play Developer API
인프라/도구	AWS, Docker, Kubernetes, nginx, k6

수상

- 2020.12 **교내 G-Pair 경진대회** **대상 (총장상)**
  선문대학교
- 2014.06 **과내 기초프로젝트(PBL) 경진대회** **은상 (컴퓨터공학과장상)**
  선문대학교

프로젝트 한눈에

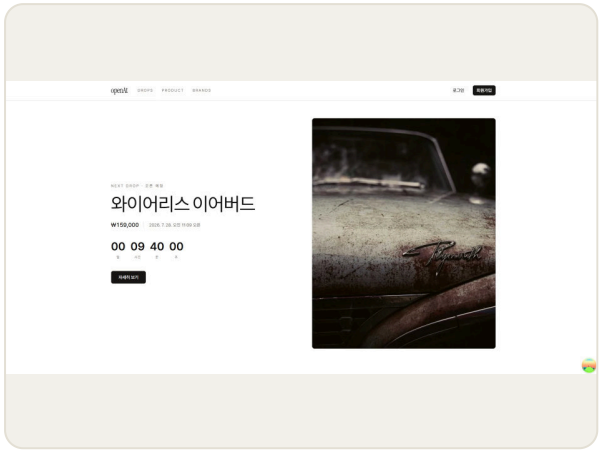
프로젝트는 종료 시점순으로 정리했습니다. 상세 내용은 최신 프로젝트부터 이어집니다.

판단 범위의 확장	프로젝트	수행 기간 / 유형	담당 범위	핵심 과제와 주요 내용
1단계 **처리 위치**	**비콕** p.23	2023.02 ~ 2023.12 자사 서비스	서버 인수 후 풀스택 단독	**화면과 서버의 책임 분담** 서버 인수 운영, 서버 사이드 썸네일 수집
2단계 **자원 운용**	**엑서플랫폼** p.16	2024.08 ~ 2025.01 외주 B2B	어드민 웹 풀스택 및 콘텐츠 서버 단독	**고객사 비용과 서버 부담을 고려한 업로드 구조** presigned URL 기반 직접 업로드 구조 결정, 다국어 스키마
3단계 **운영 연속성**	**톡메모** p.10	2023.11 ~ 2025.08 자사 서비스	백엔드 단독	**운영 중 데이터 구조 전환** 오프라인 우선 동기화(Local-first), 서버 버전 병행 운영과 데이터 이관
4단계 **부하 대응**	**openAt** p.04	2026.06 ~ 2026.07 교육 프로젝트	게이트웨이, 대기열, 회원	**한정된 자원에서의 트래픽 제어** API 게이트웨이 인증, 진입통제 대기열, SSE 실시간 전송

Project 1. openAt

선착순 한정 판매 시스템

회원, API 게이트웨이, 대기열 모듈 담당



기간	2026.06 ~ 2026.07
팀	백엔드 개발자 5
링크	**GitHub** https://github.com/prgrms-be-adv-devcourse/beadv6_6_Let-sGPT_BE

목표

재고 연동 입장 정원 제어로 선착순 입장 정합성 유지

판매 개시 시점 요청 유입 제어로 추가 자원 없이 하위 서비스 과부하 방지

대기 인원 증가에 따른 반복 조회 요청 감소

사용 기술

Java, Kotlin, Spring Cloud Gateway

Spring MVC, Spring WebFlux, Coroutines, SSE

Redis, Kafka, JPA/Hibernate, PostgreSQL, Kubernetes

담당 역할

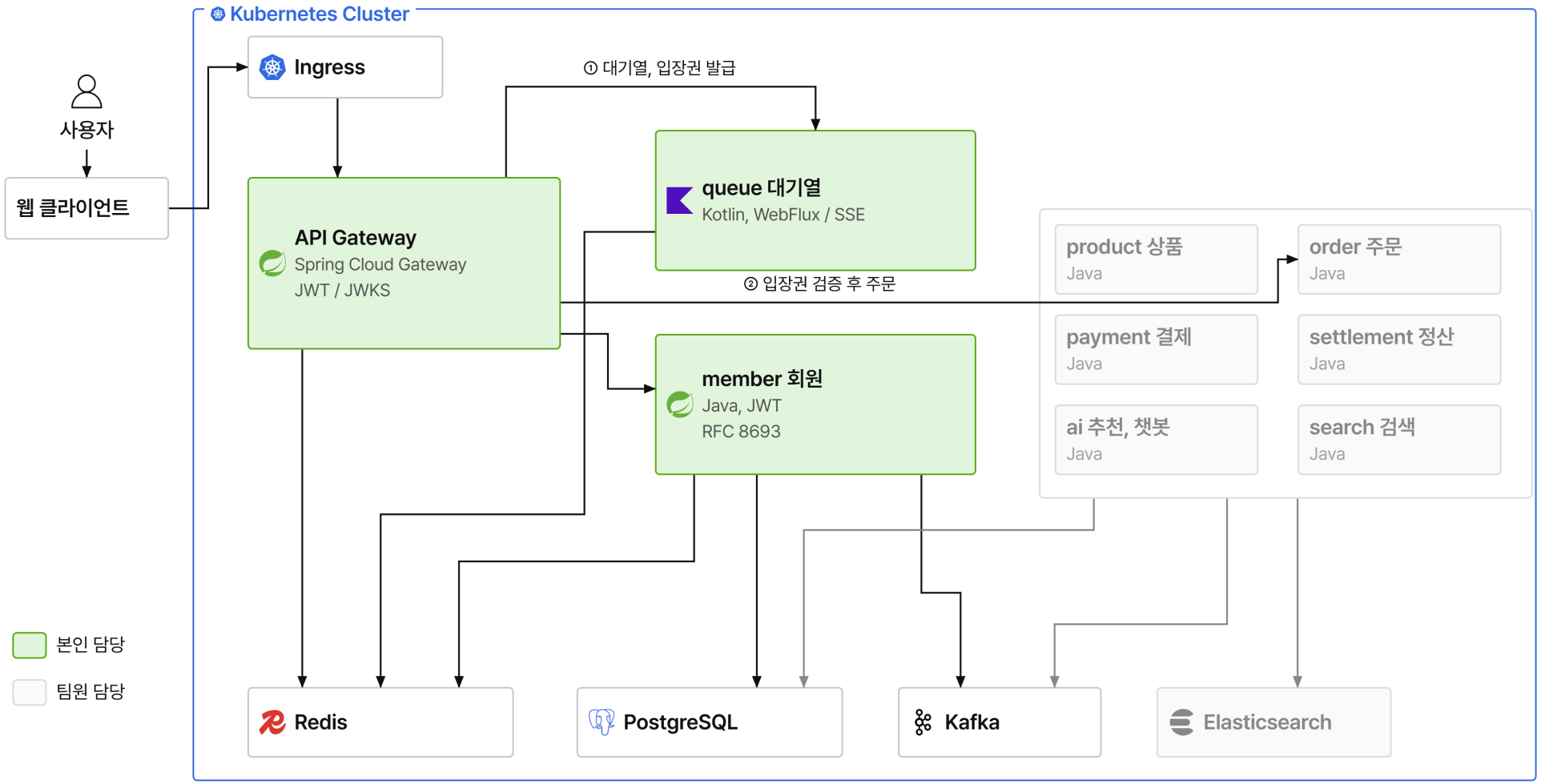
API 게이트웨이 중앙 집중 인증과 하위 모듈 인증 정보 전달

역할 기반 권한 제어와 토큰 교환 기반 판매자별 접근 범위 제한

유량제어와 진입통제 대기열 기반 하위 모듈 요청 유입 제어

대기 상태 전달을 폴링에서 SSE로, 요청 처리를 WebFlux로 전환

Project 1. 시스템 아키텍처



Project 1. 구현 기능 및 개발 과정

인증과 권한

- Spring Cloud Gateway 기반 중앙 집중식 인증
- 게이트웨이 기반 인증 헤더 스푸핑 방지
- RFC 8693 토큰 교환 기반 판매자별 접근 제어
- 키 교체를 고려한 RS256 서명과 JWKS 기반 공개키 제공

대기열 입장 제어

- Redis 기반 진입통제 대기열 구현
- Redis Lua 스크립트 원자 실행으로 동시 입장 정합성 확보
- 재고 연동 입장 정원 제어
- Sorted Set 기반 선착순 순번 관리

실시간 데이터 전송

- SSE 기반 대기 상태 실시간 전송
- WebFlux와 코루틴 기반 비동기 처리
- 동시 처리 상한 필터와 초과 요청 429 응답



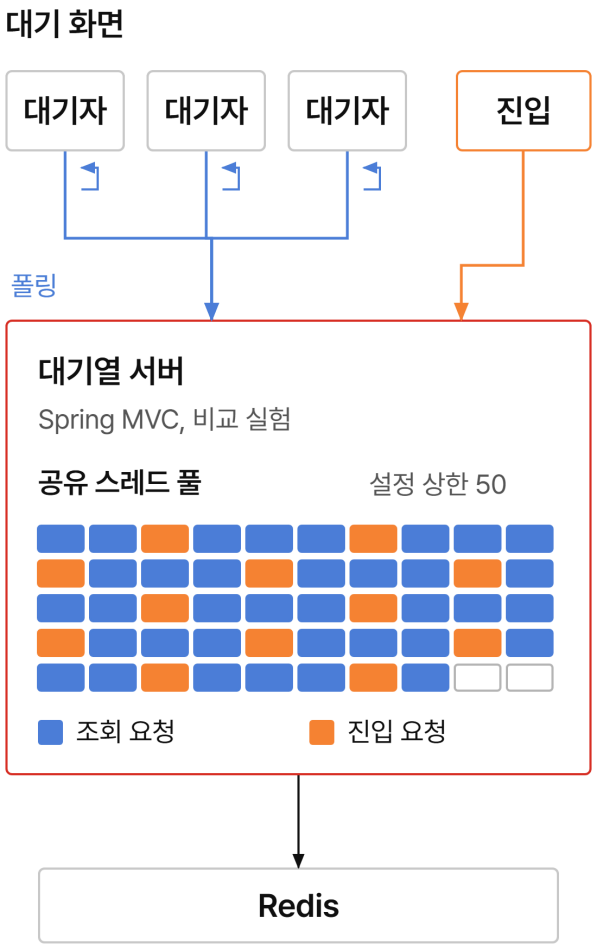
Trouble Shooting 1.

문제 상황

- 판매 개시 시점의 동시 접속을 가정한 대기열
- 앞 순번의 구매 결정 전까지 다음 입장을 보류하는 엄격한 FIFO
- 2초 주기 폴링으로 입장 가능 여부를 확인하는 구조
- 레플리카 증설 없이 아키텍처로 요청 부하를 줄여야 하는 환경

원인 분석

- 대기 인원이 늘수록 증가하는 2초 주기 조회 요청
- MVC 구조에서 조회와 입장 요청의 동일 스레드 풀 공유
- 300ms 인위 지연으로 스레드 풀 포화 재현 시 조회와 입장 요청 모두 응답 지연



Trouble Shooting 1.

폴링을 SSE로 전환해 대기열 요청 수 89% 감소, 구매 차례 알림 시간 절반 단축

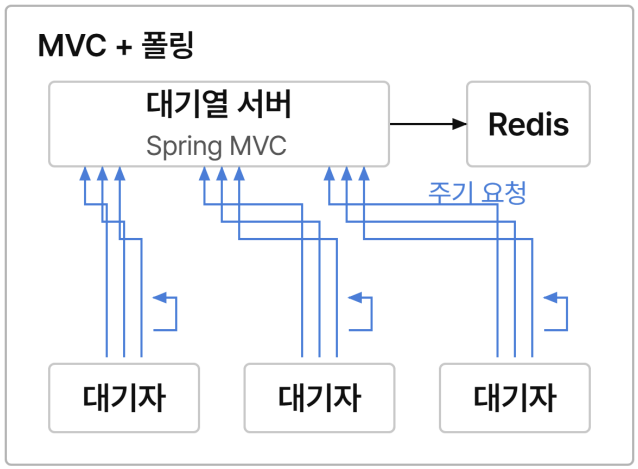
해결 과정

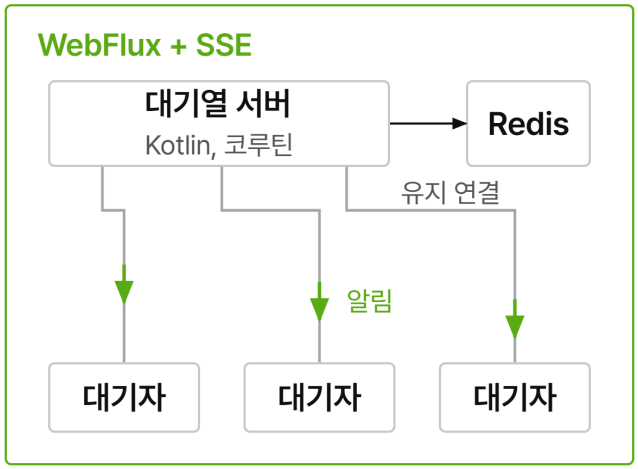
- 폴링 주기 연장, 적응형 폴링은 대기자 인지 지연으로, WebSocket은 양방향 불필요로 기각
- SSE 기반 입장 가능 여부 푸시로 2초 주기 반복 조회 제거
- WebFlux, 코루틴, Reactive Redis 적용으로 I/O 대기 중 스레드 점유 해소
- 통신 방식과 실행 모델의 효과 분리 측정

처리 결과

- 반복 조회를 SSE 푸시로 대체해 대기열 요청 수 **89% 감소**
- 빈자리 발생부터 다음 대기자 알림까지 걸리는 시간 **52% 단축**
- 폴링 고정, MVC 대비 WebFlux 대기열 진입 응답 시간 **37% 단축**

k6 부하 시험. 37%는 중앙값, 양쪽 폴링 고정, 인위 지연 300ms, 동시 부하 400명





Project Review 1.

openAt은 트래픽이 한꺼번에 몰리는 상황을 처음 다뤄 본 프로젝트입니다. 서비스가 커질수록 트래픽에 어떻게 대응하느냐가 곧 고객이 겪는 경험으로 이어진다고 생각해, 예정된 시간에 판매가 열리면 요청이 몰리는 선착순 한정 판매 플랫폼에서 그 요청을 가장 먼저 받는 대기열과 게이트웨이를 맡았습니다. 교육 과정에서 팀당 할당된 AWS 인스턴스는 추가 요청이 불가했고 모듈마다 메모리 할당량이 정해져 있어 자원을 늘리지 않고 아키텍처만으로 최대한 버티는 것이 목표였습니다.

대기열이 없으면 사용자는 자기 요청이 제대로 처리되고 있는지 알 수 없고, 몰리는 요청 속에서 고생 끝에 주문 절차를 밟았는데 매진이라는 결과를 받으면 사용자가 받는 상대적 박탈감이 서비스 이탈로 이어질 수 있다고 판단했습니다. 그래서 대기열을 도입하기로 했고, 게이트웨이에서는 유량제어로 전체 유입 속도를 조절하고 대기열에서는 재고만큼만 들여보내는 진입통제를 두어 주문 단계로 가는 인원을 재고에 맞췄습니다.

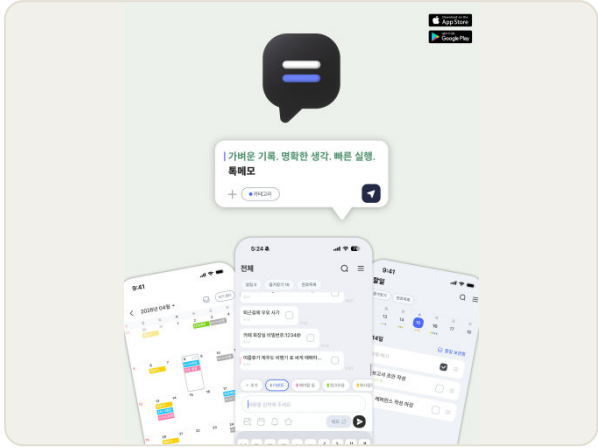
인기 상품의 예정된 오픈 시간에 순간 트래픽이 몰리는 상황을 가정하면 MVC의 최대 스레드 수를 50개로 두면 스레드 풀이 금방 차서 요청이 밀리고 결국 타임아웃까지 날 것이라고 가설을 세웠습니다. 그래서 대기 상태 전달은 2초 주기 폴링 대신 상태가 바뀔 때만 보내는 SSE로, 요청 처리는 I/O 대기 중 스레드를 점유하지 않는 WebFlux로 바꾼 뒤 k6로 검증했습니다. 스레드 풀이 가득 차는 부하에서 MVC는 예상대로 조회와 입장 요청이 함께 스레드를 기다리며 느려졌고 WebFlux에서는 그 대기가 없었습니다. 하지만 별도 한계 부하 시험에서 동시 접속을 계단식으로 올리자 MVC와 폴링 조합은 느려지기만 하고 버틴 반면, WebFlux와 SSE 조합은 같은 힙 220MB에서 동시 접속 5,000~6,000명 구간에 이르자 세 번 모두 OOM으로 멈췄습니다.

원인을 따라가 보니 MVC는 스레드 풀이 동시 처리 수를 제한해 서버를 보호하고 있었고, 바꾼 구조에는 SSE 연결 유지 비용이 더해졌는데도 동시 처리 상한이 없었습니다. 초과 요청을 대기시키면 메모리를 또 쓰게 되므로 상한을 넘는 요청은 바로 429로 돌려보냈고, 평시 주문 응답이 150ms에서 335ms로 늘었지만 요청이 몰릴 때 멈추지 않는 쪽을 택했습니다. 트래픽을 다룰 때는 처리 방식과 함께 메모리와 동시 처리 상한, 한도를 넘었을 때의 응답까지 설계해야 한다는 것을 알게 됐고, 지금이라면 상한에 도달한 추가 접속을 주기를 늘린 폴링으로 돌리는 경로까지 설계하겠습니다.

Project 2. 톡메모

SNS 대화방처럼 익숙한 방식으로 메모를 남기는 채팅형 메모 앱

서버 v1부터 v3까지 백엔드 단독 개발과 운영



기간 2023.11 ~ 2025.08

팀 기획자 1
앱 개발자 1
서버 개발자 1 (담당)

링크 **App Store**
https://apps.apple.com/kr/app/id6474009588

Google Play
https://play.google.com/store/apps/details?id=com.headit.easycalendar

목표

오프라인 상태에서 메모 읽기와 쓰기 유지

시간대 차이를 고려한 메모 수정과 삭제의 동기화 충돌 해소

데이터 이관 중 v1 앱 사용자 이용 유지

사용 기술

Kotlin, Spring Boot 3, JPA, PostgreSQL

Socket.IO, Caffeine

AWS EC2, RDS, Jib, Docker

담당 역할

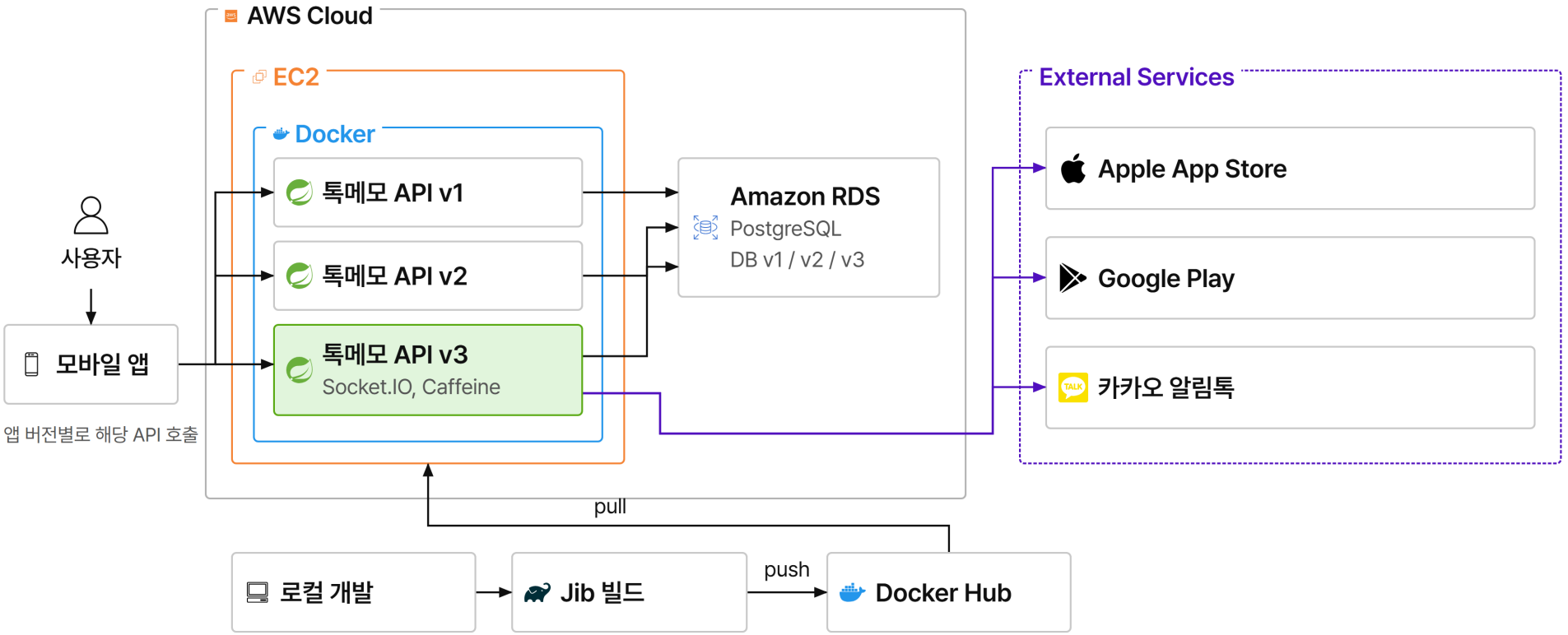
서버 v1부터 v3까지 병행 운영과 버전 간 데이터 이관

메모 원본은 앱에, 사본은 서버에 두는 오프라인 우선 동기화

Socket.IO 기반 다기기 실시간 동기화와 접속 세션 관리

인앱결제 구독 상태 검증

Project 2. 시스템 아키텍처



Project 2. 구현 기능 및 개발 과정

메모 동기화와 충돌 처리

- 메모 원본은 앱에, 사본은 서버에 두는 오프라인 우선 동기화
- 수정 시각과 삭제 시각 비교로 충돌 해소
- 시간대를 고려한 jsonb 기반 현지 시각과 오프셋 저장

실시간 동기화

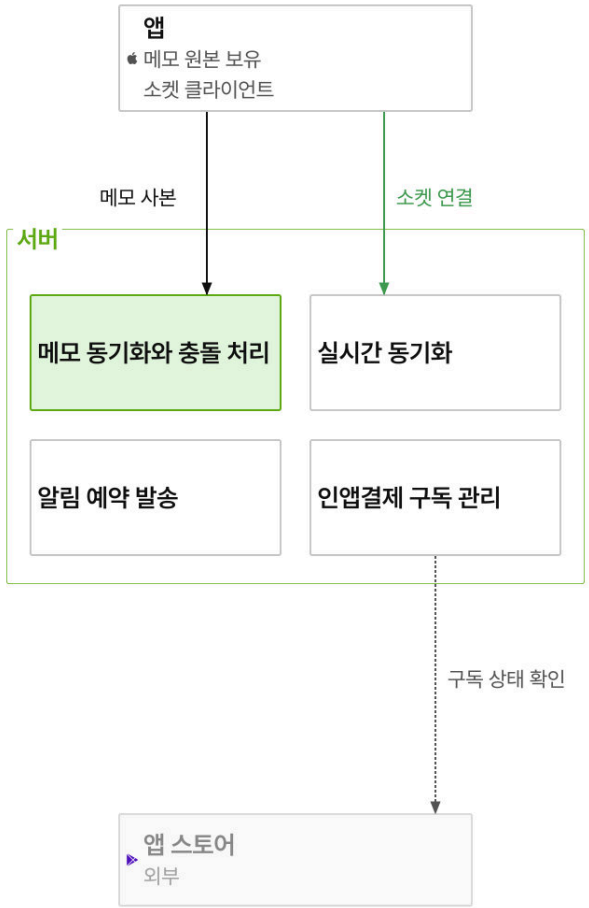
- netty-socketio 기반 다기기 실시간 동기화
- 일회성 토큰 기반 소켓 핸드셰이크 인증
- 사용자별 접속 세션 관리

알림 예약 발송

- 반복 종류별 예약 방식과 10분 전 사전 알림
- Quartz 대신 스케줄러와 예약 관리자를 직접 구성
- 카카오 알림톡 API 연동 발송

인앱결제 구독 관리

- StoreKit, Google Play Developer API 기반 영수증 검증
- 스토어 서버 알림 웹훅 기반 구독 데이터 자동 갱신
- App Store Server API 재조회와 JWS 서명 검증
- 유니크 제약과 ON CONFLICT로 결제 영수증 중복 저장 방지



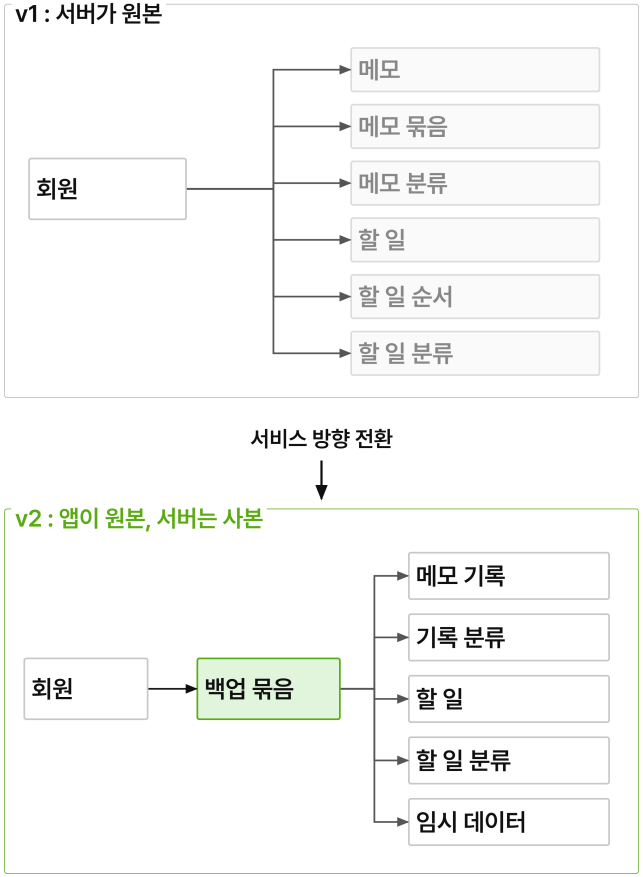
Trouble Shooting 2.

문제 상황

- 서비스 방향 전환에 따른 데이터 구조 전면 변경
- 강제 업데이트가 없는 v1 앱 사용자의 이용 중단 우려

대안 분석

- 단일 서버 API 버전 분기, 당시 컨트롤러 상속 구조에서 공통 API 재사용 방안을 찾지 못해 기각
- 기존 테이블 확장, 변경 폭이 커 신규 설계가 효율적이라 기각
- 앱 버전별 요청 라우팅, 요청마다 버전 전달과 분기가 필요해 기각
- 구버전 기능 중단, 기존 사용자 이용 차단으로 기각
- 별도 인스턴스 증설, 서비스 규모 대비 추가 비용으로 기각



v1 앱은 강제 업데이트가 없어 v1 서버 유지

Trouble Shooting 2.

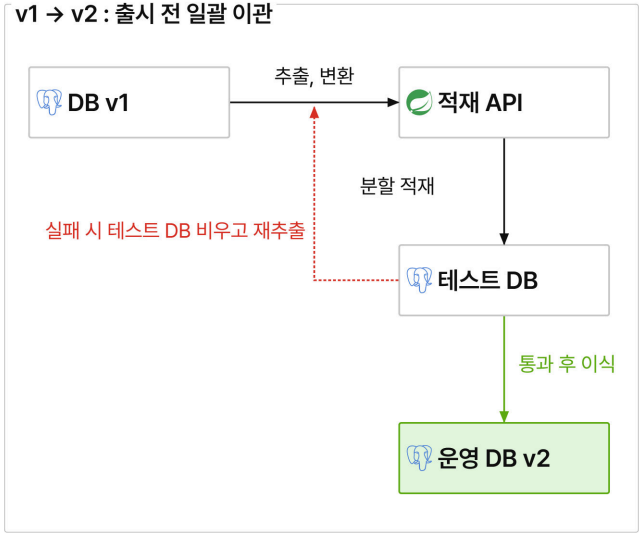
v1 앱 이용을 유지한 채 서버와 데이터 구조를 두 차례 전환

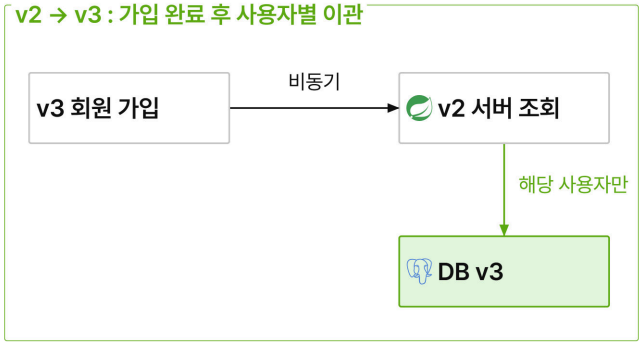
해결 과정

- 버전별 서버와 DB 분리, 단일 EC2 내 컨테이너 병행 운영
- DB 직접 적재 대신 새 서버 검증을 거치는 적재 API 방식 선택
- 요청 본문 크기 한계에 맞춘 분할 이관
- 테스트 DB에서 적재와 검증 반복 후 운영 DB 이식
- v3는 가입 완료 후 해당 사용자의 v2 데이터 비동기 이관

처리 결과

- v1 앱 사용자는 업데이트 없이 계속 이용
- v1에서 v2 이관 전후 데이터 건수 일치 확인 후 출시
- 이관에 실패해도 v3 가입은 정상 처리





Project Review 2.

톡메모는 운영 중인 서비스를 고도화하면서 데이터 구조를 두 차례 새로 설계하고 기존 데이터를 이관한 프로젝트입니다. 먼저 출시해 반응을 보고 지속 여부를 판단하는 서비스였는데, v1 이후 고도화가 결정되면서 방향이 크게 바뀌었고 데이터 구조도 많이 달라져 기존 테이블에 컬럼을 추가하고 수정하는 것보다 새로 만드는 것이 효율적이라고 판단했습니다. 그런데 v1 앱에는 새 버전이 나와도 업데이트 전 이용을 막는 기능이 없어 사용자가 구버전을 계속 쓸 수 있었습니다. 그 사용자들의 이용이 갑자기 끊기지 않도록 v1 서버를 함께 운영하며 데이터를 옮겼습니다.

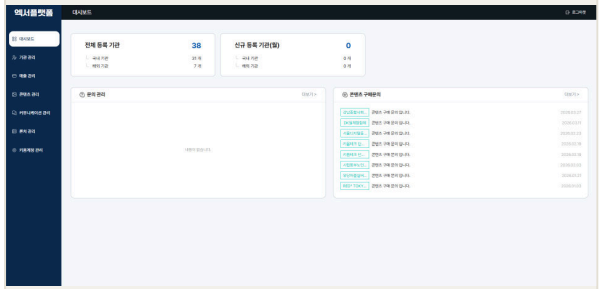
v1에서 v2로 옮길 때는 DB에 직접 넣기보다 제게 익숙한 API 적재 방식이 더 빠르고 정확하다고 판단했고, 새 서버의 검증을 통과한 데이터만 들어간다는 이점도 있었습니다. 테스트 DB에 넣어 보고 실패하면 테스트 DB를 비우고 처음부터 다시 하기를 반복해, 이관 전후 건수가 일치하는 것을 확인한 DB를 운영 시작 전에 옮겼습니다. v2에서 v3로는 가입한 사용자의 데이터를 개별 이관했고, v2 서버를 내릴 시점은 이관 요청이 뜸해지는지를 보고 판단하려 했습니다. 지금이라면 Flyway 같은 마이그레이션 도구로 스키마 변경 이력을 남기고, 사용자별 이관 여부와 시각도 기록해 구버전 서버의 종료 판단 근거로 삼겠습니다.

v2부터는 메모 원본을 앱에 두고 서버는 사본을 갖는 오프라인 우선 구조였기 때문에, 여러 기기에서 수정과 삭제가 엇갈리면 어느 쪽을 남길지 정해야 했습니다. 기술로 정답이 나오는 문제가 아니라 서비스 정책의 문제라고 판단해 경우의 수와 처리 기준을 정리해 기획자와 합의했고, 삭제 시각이 수정 시각보다 늦을 때만 지우는 방식으로 구현했습니다. 그때는 이름을 몰랐지만 지금 돌아보면 local-first라고 부르는 구조였고, 충돌 처리도 CRDT의 한 방식인 Last-Write-Wins와 같은 원리로 시각을 비교해 최신 변경을 우선한 것이었습니다.

Project 3. 엑서플랫폼

복지기관 대상 VR 콘텐츠 통합 운영 플랫폼

VR 기기 런처와 연동되는 서버, 어드민 웹 클라이언트 풀스택 개발



기간	2024.08 ~ 2025.01
팀	기획자 1 유니티 개발자 2 **웹 풀스택 개발자 1 (담당)**
링크	비공개 운영 중

목표

고객사 인프라 비용을 고려한 단일 서버 자원 보호

스키마 변경 없는 지원 언어 확장

업로드 성공 상태의 활성 버전만 기관 런처에 노출

사용 기술

Kotlin, Spring Boot 3.3, JDK 17, JWT

PostgreSQL, JPA, QueryDSL

AWS EC2, S3, RDS, ALB

TypeScript, React 18, Zustand, MUI

담당 역할

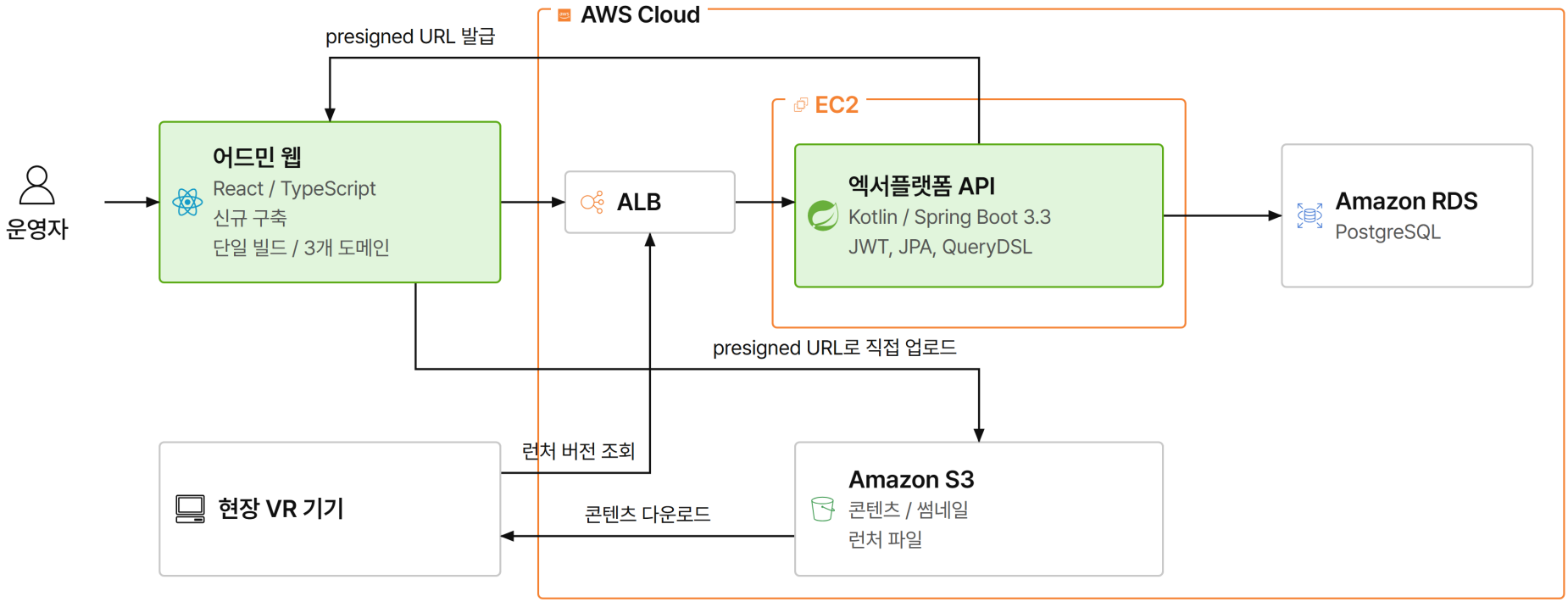
JWT 인증과 기관별, 사업자별 회원 권한 구조 설계

콘텐츠 배포와 기기 관리, S3 직접 업로드 연동

언어별 행 분리 스키마 기반 다국어 지원

어드민 웹 프론트엔드 신규 구축과 단일 빌드 멀티 도메인 운영

Project 3. 시스템 아키텍처



Project 3. 데이터 구조



Project 3. 구현 기능 및 개발 과정

인증과 권한

- JWT 기반 토큰 인증
- 기관별, 사업자별 회원 권한 구조 설계

콘텐츠 업로드와 관리

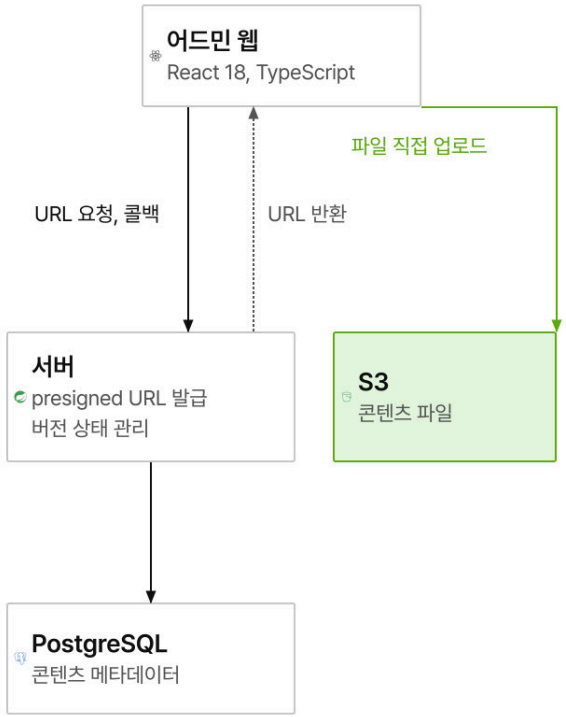
- presigned URL 직접 업로드로 파일 전송 경로에서 API 서버 제외
- 어드민 웹의 업로드 성공, 실패 콜백에 따른 버전 상태 관리
- 업로드 성공 상태의 활성 버전만 기관 런처에 노출
- S3 비동기 클라이언트 기반 병렬 복사, 삭제로 폴더 이동 구현

다국어 지원

- 언어별 행 분리 다국어 스키마 설계
- 스키마 변경 없는 지원 언어 확장 구조

멀티 도메인 운영

- 기능이 같은 세 서비스의 빌드와 이미지를 공통화해 3개 도메인 운영



Trouble Shooting 3.

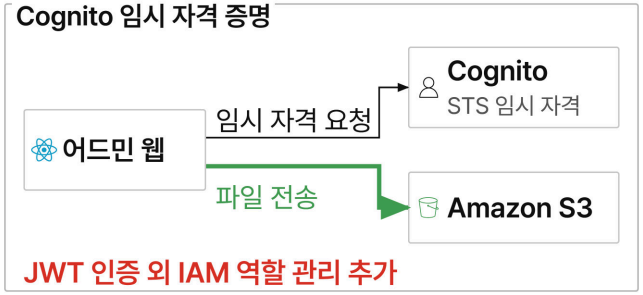
문제 상황

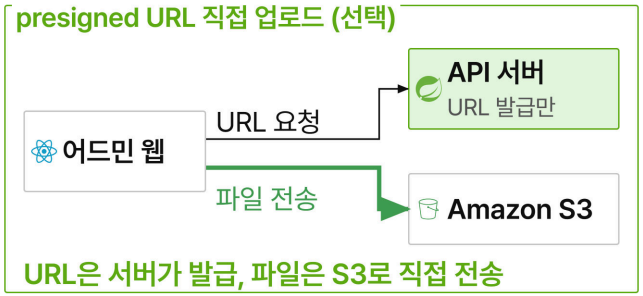
- 콘텐츠 버전 1건당 VR 빌드 구성 파일 100~200개
- EC2 1대 단일 서버 구성, 인프라 비용은 고객사 부담
- 업로드 중이거나 실패한 버전의 기관 런처 노출 차단 필요

대안 분석

- 서버 파일 시스템 저장, 디스크 증설 비용과 운영 부담으로 기각
- 서버 경유 업로드, 단일 서버의 네트워크 자원 점유로 기각
- Cognito 임시 자격 증명, JWT 인증 외 IAM 역할 관리 추가로 기각







Trouble Shooting 3.

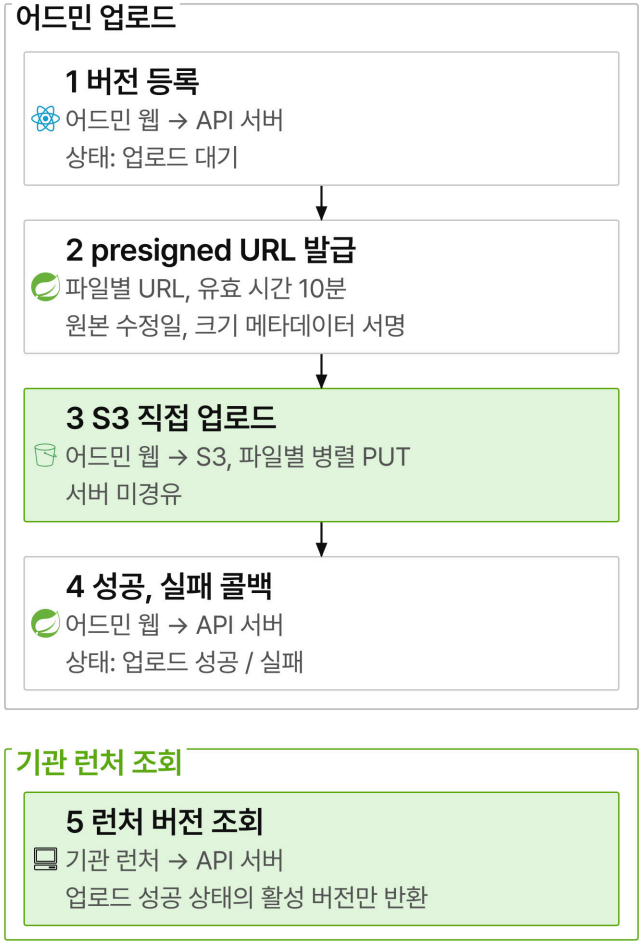
파일 전송 경로에서 API 서버 제외, 업로드 성공 상태의 활성 버전만 런처에 노출

해결 과정

- S3 저장과 presigned URL 직접 업로드 선택
- 업로드 대기 상태로 버전 등록 후 파일별 URL 발급
- 원본 수정일과 크기를 사용자 정의 메타데이터로 서명 요청에 포함
- 어드민 웹의 파일별 병렬 PUT 업로드
- 업로드 성공, 실패 콜백으로 버전 상태 전환

처리 결과

- 서버는 URL 발급과 버전 상태 관리, 파일은 S3로 직접 전송
- 업로드 성공 상태의 활성 버전만 반환하는 런처 버전 조회
- 원본 수정일과 크기를 S3 객체 메타데이터로 기록



Project Review 3.

엑서플랫폼은 AWS로 외주 플랫폼을 구축하며 고객사가 부담할 클라우드 비용까지 설계 기준에 넣어 본 프로젝트입니다. VR 콘텐츠는 버전 하나에 파일이 100~200개라 저장소부터 서버 파일 시스템과 S3를 비교했고, 가격과 운영 부담을 생각해 S3를 골랐습니다. S3에 올리면 원본 파일의 수정일 같은 정보가 그대로 남지 않았지만 메타데이터로 따로 실어 보내면 되는 정도라 감수할 수 있다고 판단했습니다. 업로드는 서버가 받아 넘기는 방식과 임시 자격 증명을 주는 방식, presigned URL을 비교했습니다. 서버가 받아 넘기면 파일 전송이 단일 서버의 네트워크 자원을 점유하고, 임시 자격 증명은 JWT 인증 외에 IAM 역할까지 관리해야 해서, 파일은 S3로 직접 보내면서도 파일마다 경로와 유효 시간을 서버가 정할 수 있는 presigned URL을 선택했습니다.

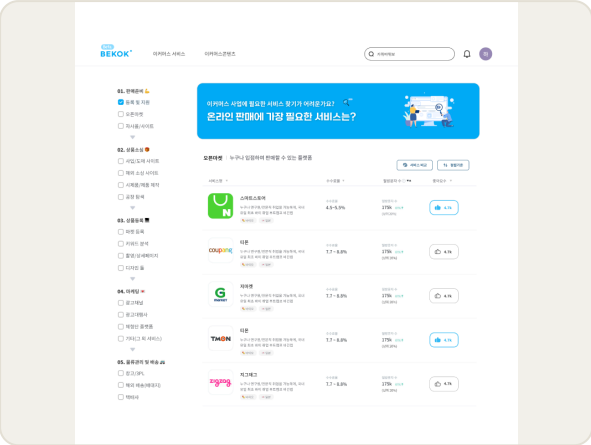
1차에는 다국어 기획이 없어 한 테이블에 언어별 칸을 두는 방식으로 진행했는데, 2차 기획에 언어별 대응이 들어오면서 언어를 따로 빼 두어야 나중에 언어가 늘어도 대응하기 쉽다고 판단해 언어별 행으로 스키마를 나눴습니다. 이 구조로 유니티 클라이언트가 받는 서버 데이터를 언어별로 제공했고, 일본에서도 서비스가 사용됐다고 전해 들었습니다.

presigned URL로 파일을 직접 올리는 구조에서는 서버가 업로드 결과를 따로 전달받아야 해서, 성공과 실패를 알리는 콜백으로 버전 상태를 관리했고 런처 버전 조회에는 업로드 성공 상태의 활성 버전만 반환하도록 했습니다. 이 과정에서 파일 전송을 서버에서 떼어 내는 것과 배포할 버전의 업로드 완료를 확인하는 것은 따로 설계해야 한다는 것을 알게 됐습니다. 지금이라면 S3 Event Notifications로 서버가 파일별 업로드 완료를 직접 확인하고, 해당 버전의 파일이 모두 올라왔을 때 배포 가능한 상태로 바꾸겠습니다. 이벤트 중복 처리와 파일별 완료 상태 관리가 늘어나더라도, 배포 여부를 클라이언트의 콜백에만 맡기지 않는 쪽을 택하겠습니다.

Project 4. 비콕

이커머스 셀러 사업 준비 플랫폼

기존 서버 인수 운영과 프론트엔드 신규 구축



기간 2023.02 ~ 2023.12

팀 기획자 1
백엔드 개발자 1
풀스택 개발자 1 (담당)

링크 **웹**
https://bekok.headit.me (운영 중지)

목표

일정의 화면 표시 순서와 서버에 저장된 순서 간 정합성 유지

CORS로 막힌 외부 이미지의 썸네일 제공과 원본 소실 대비

CSR 페이지의 검색 결과 노출 확보

사용 기술

Kotlin, Spring Boot 2.7, JDK 11

PostgreSQL, JPA, QueryDSL, Jsoup

JavaScript, React 18, Redux, AWS S3

담당 역할

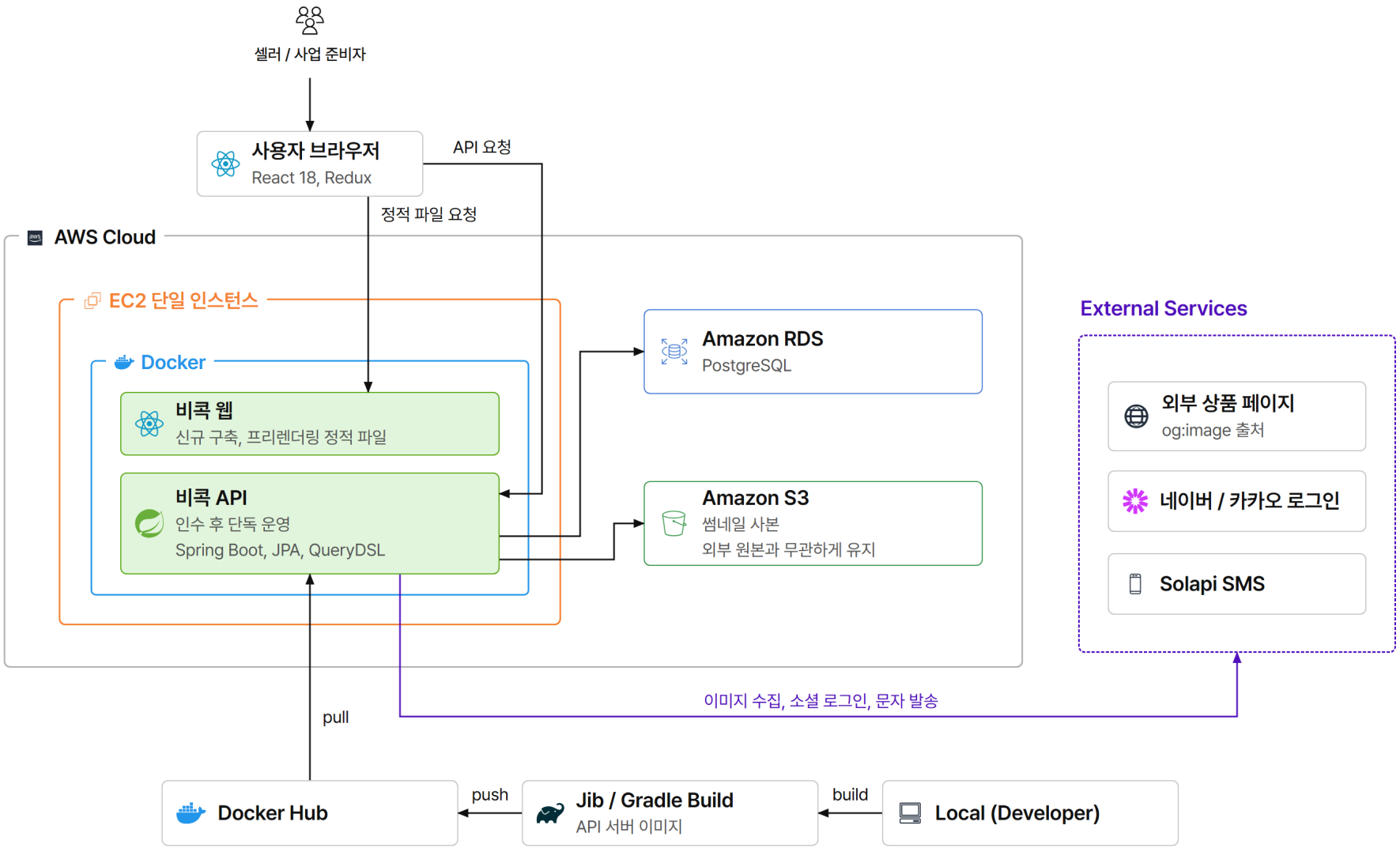
기존 백엔드 코드 분석과 인수 후 단독 운영

프론트엔드 신규 구축과 클라이언트, 서버 간 책임 분담 설계

Jsoup 기반 외부 링크 대표 이미지 수집과 S3 사본 저장

react-snap 기반 프리렌더링 적용

Project 4. 시스템 아키텍처



Project 4. 구현 기능 및 개발 과정

소셜 로그인

- 카카오, 네이버 소셜 로그인 연동
- 화면이 받은 소셜 토큰을 서버로 전달, 서버가 제공자 API로 검증

외부 링크 썸네일 수집

- Jsoup 기반 og:image URL 추출과 서버 사이드 이미지 수집
- 원본 소실 대비 S3 사본 저장

사업 준비 일정 관리

- 서버 사이드 일정 순서 계산과 저장
- 화면과 서버에 동일한 일정 순서 규약 적용

검색 노출 대응

- react-snap 기반 빌드 타임 프리렌더링
- 프리렌더링 적용 페이지의 구글 검색 결과 노출



Trouble Shooting 4.

문제 상황

- 게시글 외부 링크의 대표 이미지 썸네일 노출 필요
- 화면에서 외부 이미지를 받아 S3에 저장하던 구조
- 일부 사이트의 이미지 수집 실패로 썸네일 미표시

원인 분석

- 일부 사이트의 CORS 제한으로 브라우저에서 이미지 수집 불가
- 외부 사이트의 CORS 정책은 화면 코드로 바꿀 수 없는 제약



검토한 대안: 화면 수집 유지

외부 사이트의 CORS 정책은
화면 코드로 바꿀 수 없어 기각

Trouble Shooting 4.

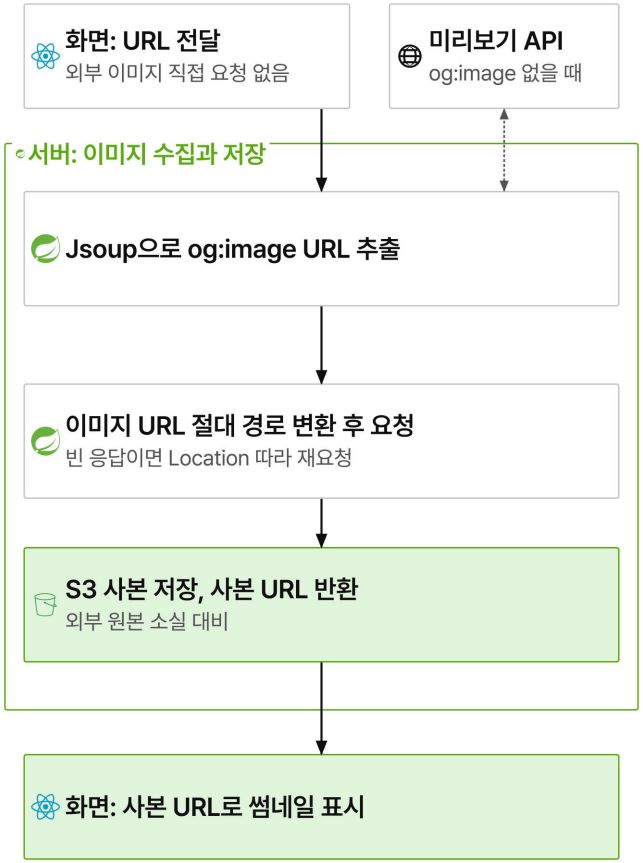
CORS로 수집하지 못하던 이미지의 썸네일 제공, 수집과 저장을 서버로 이전

해결 과정

- 브라우저 수집 유지안은 CORS 제약으로 기각, 수집을 서버로 이전
- 화면은 URL만 전달, 서버가 og:image URL 추출 후 이미지 수집
- og:image 값이 비어 있으면 외부 미리보기 API로 재조회
- 상대 경로를 절대 URL로 변환, 리다이렉트 대상에 재요청
- 외부 원본 소실에 대비한 S3 사본 저장과 사본 URL 반환

처리 결과

- CORS 제한으로 수집하지 못하던 이미지의 썸네일 노출
- 화면은 URL 전달, 서버는 수집과 저장을 맡는 책임 분리
- 외부 원본과 무관한 S3 사본 URL로 썸네일 제공



Project Review 4.

비콕은 처음으로 서버를 맡아 화면과 서버가 각각 무엇을 맡아야 하는지 판단해 본 프로젝트입니다. 전임자가 계약 만료로 떠나면서 문서나 구두 인계 없이 서버를 넘겨받았고, 엔드포인트에서 서비스 코드, DB 코드 순으로 따라 들어가며 구조를 파악했습니다. 같은 시기에 React로 프론트엔드를 새로 만들고 있었기 때문에 기능 하나를 만들 때마다 이 일을 화면에서 할지 서버에서 할지를 제가 정해야 했습니다.

게시글에 붙인 외부 링크의 썸네일은 처음에 화면에서 외부 이미지를 받아 S3에 올리도록 만들었는데, 일부 사이트의 CORS 정책 때문에 브라우저에서 이미지를 가져올 수 없어 화면 코드를 고쳐서는 해결되지 않았습니다. 그래서 화면은 URL만 넘기고 이미지 수집과 저장은 서버가 맡도록 옮겼고, 외부 원본이 사라져도 썸네일이 깨지지 않도록 S3에 사본을 남겼습니다. 이 일로 외부 이미지 수집처럼 브라우저의 CORS 제약을 받는 일은 화면보다 서버가 맡는 것이 맞다는 기준이 생겼습니다. 일정 순서는 서버가 계산하고 화면에도 같은 규약을 적용해, 화면에 보이는 순서와 저장된 순서가 어긋나지 않게 했습니다.

검색 노출은 Next.js 같은 SSR 프레임워크로 바꾸는 방법과 기존 React에 react-snap으로 프리렌더링을 적용하는 방법을 비교했는데, 서버를 막 넘겨받아 익히는 중이라 새 프레임워크까지 학습할 시간은 없다고 판단해 react-snap을 선택했고 구글 검색 결과에 노출되는 것까지 확인했습니다.

지금이라면 서버가 외부 URL을 대신 요청하게 된 만큼 내부망 주소 차단과 리다이렉트 대상 재검증, 응답 크기와 시간 제한을 함께 두고, 썸네일 수집은 게시글 저장과 떼어 별도 작업으로 처리해 외부 사이트가 느리거나 실패해도 게시글 저장에는 영향이 없게 하겠습니다. 수집 책임을 서버로 옮기면 그 요청이 만드는 위험과 지연도 서버가 함께 맡게 되기 때문입니다.

Thank you

검토해 주셔서 감사합니다