손 호 영

Backend Engineer

010-8586-4134 · tls1234568@naver.com · github.com/pyrimidine02 · portfolio.noraneko.cc

---

안녕하세요. 저는 주어진 제약 안에서 구조를 짜고, 미래에 생길 문제를 미리 가정하면서 시스템을 설계하는 일을 좋아하는 개발자 손호영입니다. 알고리즘 문제풀이를 하면서 "어떤 입력이 들어와도 깨지지 않는 로직"을 고민해 온 경험이 자연스럽게 서비스 아키텍처와 데이터 모델링으로 확장되었고, 지금은 일상의 불편함을 "어떻게 하면 백엔드 구조로 풀 수 있을까"라는 관점에서 바라보고 있습니다.

RailNetwork

서울 지하철 실시간 운행 정보를 다루는 'RailNetwork' 프로젝트는 이런 문제 해결 방식이 처음으로 실서비스 수준의 코드에 적용된 경험이었습니다. 서울교통공사 공식 앱을 Proxyman으로 리버스 엔지니어링해 공개되지 않은 내부 API 엔드포인트를 찾고, 문서가 없는 상태에서 응답 스키마와 제약을 스스로 정의해야 했습니다. 열차 상태가 1, 2, 3, 4 숫자 코드로만 내려오는 상황에서, 이를 "역 도착/출발/지연/운행 종료"와 같은 도메인 상태로 재해석하고, 시간표와 결합해 지연 시간을 계산하는 로직과 train update 알고리즘을 직접 설계했습니다. 이 과정에서 "API가 어떤 식으로 바뀌더라도 내 도메인 모델은 최대한 안정적으로 유지되도록" 상태 전이를 정교하게 설계했고, 이 경험이 이후 아키텍처·DB·캐싱을 고민할 때 기준점이 되었습니다.

Girls Band Tabi

이후 제가 기획부터 설계, 구현까지 책임지고 있는 'Girls Band Tabi'에서는 처음부터 "미래에 생길 문제들을 설계 단계에서 최대한 선제적으로 막는다"는 기준으로 아키텍처를 잡았습니다. 여러 애니·밴드 프로젝트를 한 백엔드에서 통합 관리해야 하고, 시간이 지날수록 프로젝트·유닛·장소·라이브·뉴스·커뮤니티 데이터가 계속 쌓일 수밖에 없는 구조입니다. 만약 이를 단순히 기능 단위 컨트롤러와 테이블 몇 개로만 시작하면, 나중에 "특정 프로젝트의 성지 중에서, 후속 유닛의 라이브까지 포함해 타임라인으로 조회" 같은 요구가 등장했을 때 JOIN과 조건이 얽히면서 급격히 복잡해질 수 있습니다. 이를 피하기 위해 Kotlin + Spring Boot 3.x 기반의 MVC 아키텍처 위에, 프로젝트/유닛/장소/이벤트/커뮤니티를 각각 독립적인 도메인으로 나누고 서비스·리포지토리 계층도 도메인 경계를 기준으로 분리했습니다. 아직 모놀리식 배포이지만, 도메인 경계를 코드에서 먼저 분리해 두면 추후 일부 기능만 별도 서비스로 분리하더라도 의존성을 최소화한 채 옮길 수 있도록 의도한 설계입니다.

데이터베이스 설계에서는 "나중에 데이터를 어떻게 조회할 것인가"를 먼저 상정한 뒤 스키마를 잡았습니다. 하나의 인스턴스에서 여러 프로젝트와 유닛, 성우, 장소, 방문 기록, 라이브 이벤트, 미디어·뉴스, 커뮤니티 포스트를 모두 관리해야 하기 때문에, 처음부터 EAV(Entity-Attribute-Value)나 과도한 다중 JOIN에 의존하면 쿼리가 깨지기 쉬운 구조가 됩니다. 대신, 프로젝트·유닛·장소·이벤트는 각각 명확한 정체성을 가진 엔티티로 두고, 그 사이의 관계는 "조회 패턴이 실제로 필요한 곳에서만" 테이블 수준으로 풀어내도록 설계했습니다. 외래키 제약은 일부 핵심 관계에만 제한적으로 사용하고, 나머지는 애플리케이션 레벨에서

무결성을 관리하는 쪽을 택했습니다. 이렇게 하면 초기에는 다소 수고가 들지만, 나중에 특정 관계를 다른 방식으로 모델링하거나 분리 서비스로 옮길 때 DB 제약 때문에 발이 묶이지 않도록 하기 위함입니다.

위치 기반 서비스라는 특성상 공간 데이터와 관련된 문제도 미리 고려했습니다. 성지 장소는 단순히 위도·경도만 저장하는 것이 아니라, "작품/밴드 기준의 성지 목록"과 "실제 사용자의 현재 위치 기준 근접 검색"을 모두 지원해야 합니다. 시간이 지나면서 장소가 많아지면 PostGIS 인덱스가 비대해지고, 단순 거리 계산만으로는 쿼리 비용이 크게 증가할 수 있습니다. 이를 피하기 위해 장소 테이블에 PostGIS Geography 타입을 사용해 구면 좌표 연산을 정확하게 처리하면서도, 활성 상태인 장소만을 인덱싱하는 부분 인덱스를 적용해 인덱스 크기와 검색 비용을 조절했습니다. 또, 지역은 행정구역이 아닌 "작품/밴드 기준의 영역"으로 계층화된 region 모델로 별도 분리해, "특정 밴드 성지만 모아서 보기" 같은 도메인 중심 조회를 공간 계산과 분리할 수 있도록 했습니다.

캐싱은 "읽기 요청이 특정 도메인에 쏠릴 때 DB와 네트워크가 어디서 먼저 병목이 날지"를 미리 가정하고 설계했습니다. 실제 공개 서비스 전이라 구체적인 트래픽 수치는 없지만, 기획 단계에서부터 성지 목록·장소 상세·프로젝트/유닛 메타데이터는 조회 비중이 높고 쓰기 빈도는 상대적으로 낮을 수밖에 없다고 보았습니다. 이 패턴을 염두에 두고, Caffeine 기반의 L1 캐시와 Redis L2 캐시를 조합한 다단계 캐싱 구조를 도입했습니다. 애플리케이션 내부에서는 요청당 반복 조회를 줄이기 위해 Caffeine으로 단기 캐시를 두고, 서비스 전체에서 공유해야 하는 데이터는 Redis를 통해 캐시하는 구조입니다. 여기서 중요하게 본 부분은 "캐싱 그 자체"보다, 나중에 데이터 일관성과 캐시 무효화가 발목을 잡지 않도록 하는 설계였습니다.

예를 들어, 장소 정보가 변경되거나 비공개 전환이 되었을 때, 단순히 전체 캐시를 비우면 초기 트래픽에서 캐시 스톰이 발생하고, 반대로 무효화를 느슨하게 하면 사용자가 오래된 장소 정보를 보게 될 위험이 있습니다. 이를 막기 위해 캐시 키 네임스페이스를 도메인 단위로 설계하고, 장소·프로젝트·유닛 각각에 대해 "어떤 이벤트가 발생했을 때 어떤 키 집합을 비워야 하는지"를 코드와 문서에 함께 정의했습니다. 또한, 캐시는 기본적으로 cache-aside 패턴을 사용해 DB가 항상 진실의 근원으로 남도록 하고, 향후 트래픽 패턴을 계측한 뒤에만 더 공격적인 캐싱(예: pre-warm, write-through 등)을 고려할 수 있도록 확장 여지만 열어 둔 상태입니다.

결론

아직 Girls Band Tabi는 실 서비스 단계에 도달하지 않았고, 실제 트래픽이나 수치 기반의 성능 튜닝이 이뤄진 상태는 아닙니다. 그럼에도 RailNetwork와 Girls Band Tabi를 설계·구현하면서, "나중에 어디에서 병목과 일관성 문제가 터질 것인지"를 먼저 가정하고, 아키텍처·DB·캐싱 설계에서 이를 선제적으로 줄이는 방향으로 선택해 왔다고 생각합니다. 귀사에서도 도메인을 빠르게 이해하는 것에 그치지 않고, 몇 단계 뒤에 생길 문제까지 함께 상정하며 구조와 데이터를 설계하는 백엔드 개발자로 성장하겠습니다. 감사합니다.

손 호 영 드림