# LinkHub [링크 아카이빙 및 공유 서비스]

2023.10.10 ~ 2023.12.05

#Java17 #SpringBoot3.1.5 #JPA #QueryDSL #MySQL #Redis #Junit5 #Docker #AWS #Prometheus #Grafana #Loki

서비스 링크: https://link-hub.site

GitHub 링크: https://github.com/Team-TenTen/LinkHub-BE

## 개요

구글을 통해 많은 정보를 얻어야 하는 개발직과 같은 직군들은 정리되지 않은 많은 크롬 탭과 함께 생활하고 있습니다. 뿐만 아니라 스터디원들과 슬랙에서 좋은 자료 공유 시 다른 텍스트 메세지들에 의해 묻히는 경우도 많습니다. 또한 지역 맛집 혹은 쇼핑몰 등을 검색하면 양질의 정보가 아닌 많은 광고들로 인해 정보를 필터링하는 추가적인 비용 소모도 많습니다.

LinkHub 프로젝트는 이러한 문제점들을 해결하고자 시작된 프로젝트입니다. LinkHub는 편리한 링크 아카이빙 및 태그를 통한 필터링, 그리고 초대를 통해 언제든지 팀원들과 함께 편집할 수 있는 공유 환경을 제공합니다. 또한, 좋아요 및 즐겨찾기 수를 통해 양질의 자료들이 사용자들에게 공유될 수 있는 환경을 조성하며, 메인에 노출된 링크 저장소들은 즐겨찾기 및 가져오기 기능을 통해 구독 및 복사할 수 있도록 하여 사용자의 편의성을 높였습니다.

## 세부사항

### 1) SWAGGER를 이용한 API 문서화

- API 문서는 주로 프론트엔드 개발자들이 클라이언트
- 따라서 프론트엔드 팀원들의 사용 경험과 편의성을 조사한 결과 UI가 직관적이며, 직접 요청을 보내볼 수 있는 SWAGGER를 채택하여 문서화 진행

### 2) 로그인이 필요한 API와 필요하지 API 분리하여 개발

- 로그인이 필요한 API와 필요하지 않은 API를 분리하여 개발하여 서비스에 처음 접근하는 유저가 부담감 없이 서비스를 이용해 볼 수 있도록 기획

### 3) 트랜잭션 안의 파일 업로드 로직 제거

#### 상황

트랜잭션 내에 S3 파일 업로드 로직이 포함되어 트랜잭션을 유지 시간이 길어졌던 문제가 있었습니다.

#### 실행

Facade 패턴을 이용하여 파일 업로드 로직을 트랜잭션 내에서 분리하였습니다.

#### 결과

트랜잭션을 짧게 유지할 수 있게 되었으며, 이로 인해 DB 커넥션을 효율적으로 사용할 수 있게 되었습니다.

### 4) 링크 저장소 검색 시 발생하는 Table Full Scan을 해결하기 위한 Full-Text Index 도입 및 최적화

### (100만 건 데이터 상황에서 1,634ms → 235ms)

#### 상황

100만 건의 데이터가 저장되어 있는 상황에서 검색 쿼리 수행 시 1,634ms이 소요되었으며, 쿼리의 where절의 like %검색어% 문으로 인해 B+tree 인덱스를 사용하지 못하는 것이 원인이었습니다.

#### 실행

문제 해결을 위해 Inverted Index 기반의 MySQL Full-Text Index를 이용하였으며, 아래와 같은 Full-Text Search 옵션을 같이 사용하였습니다.

- 능동적인 검색이 가능하게 하기 위해 Stop-word 파서가 아닌 N-gram를 사용
  ('철수' 가 검색어 키워드 시 Stop-word 파서는 '철수를'을 반환하지 않음)
- 기존의 like %검색어% 구문과의 동일한 동작을 위해 불린 모드 검색과 +옵션을 사용

  *(해당 문법 → match*(space_name) *AGAINST* ('+검색어' IN BOOLEAN MODE ))

하지만 쿼리 수행 시간이 오히려 기존 대비 2배 이상 길어졌으며, 여러 가지 가설과 검증을 통해 아래와 같은 원인을 파악할 수 있었습니다. (기존: 1,634ms, 수정 후: 3,635ms)

- 원인은 "+옵션" 으로 검색어에 포함된 토큰을 하나라도 소유한 모든 데이터를 가져와 이후에 필터링을 하기 때문
  (+옵션은 키워드가 반드시 포함된 결과만을 반환)
  (ex. 검색 조건이 '+java +개발자` 라면 size 2를 기준으로 토큰화되어 "ja" 등의 토큰 중 하나라도 가진 모든 데이터를 찾아 +로 묶인 단어를 기준으로 필터링 하는 것)
- 실제로 검색어에 포함된 토큰이 하나라도 있는 모든 데이터를 조회해 보니 40만 건의 데이터 반환

문제 해결을 위해 클라이언트로부터 받는 검색 keyword를 애플리케이션에서 미리 토큰화 후 각 토큰에 '+'를 붙여 쿼리 하는 방식을 통해 최적화하였습니다.
(ex. keyword가 'java 개발자'인 경우 → '+ja +av +va +개발 +발자'로 변환)

#### 결과

MySQL Full-Text Index를 도입하고 검색 컨디션을 최적화 한 결과, 해당 API의 Latency를 1,634ms 에서 235ms로 줄일 수 있었습니다.

### 5) OneToMany 관계 Fetch Join과 페이징 동시 진행 시 발생하는 데이터 중복 문제 해결

#### 상황

링크 저장소 페이지네이션 시 OneToMany 관계의 엔티티를 Fetch Join 하면 중복된 링크 저장소 데이터가 조회되는 문제가 발생하였습니다.

해당 문제는 1 : N 관계에서 1에서 N을 Fetch Join 하였기에 N만큼의 중복 데이터를 반환하는 것이 원인으로 쿼리의 반환값을 DTO가 아닌 엔티티로 변경하면 Hibernate에 의해 중복 문제는 해결 되겠지만 연관관계가 없는 다른 엔티티의 정보를 같이 반환해야 했기에 해당 방식은 사용할 수 없었습니다.
(엔티티를 반환하는 방식도 에플리케이션 메모리에 수많은 데이터를 올려 페이징 하는 방식이기에 Out Of Memory를 유발할 수 있으므로, 좋은 방법이 아님)

#### 실행

쿼리를 분리하여 N 관계의 엔티티를 In 쿼리를 통해 조회하여 애플리케이션에서 HashMap 자료구조를 이용하여 적절하게 바인딩하여 해결하였습니다.

#### 결과

중복되지 않은 정확한 데이터를 반환할 수 있게 되었습니다.

### 6) 반정규화로 인해 발생했던 동시성 이슈(갱신 손실) 해결

#### 상황

링크 저장소 페이지네이션 API에서 즐겨찾기 개수 및 가져오기 개수 등을 반환해야 했기에 해당
집계 정보를 반정규화 하였습니다. 하지만 이로 인해 갱신 손실(Lost Update) 문제가 발생했습니다.

#### 실행

갱신 손실(Lost Update) 문제를 해결하기 위한 방법들을 조사 하였으며, 아래와 같은 이유로 비관적 락(Locking Reads) 방식을 채택하였습니다.

- **낙관적 락**

  인기 혹은 최신 저장소는 메인 페이지에 노출되며 버튼 클릭 한 번으로 이뤄지기에 충돌이 잦을 것으로 예상. 낙관적 락은 애플리케이션에서 제공하는 락으로 충돌 시 롤백 후 개발자가 정한 전략에 따라 재시도 등을 한다. 즉 충돌이 많을 경우 오버헤드가 커진다. 따라서 낙관적 락은 적절하지 않다고 판단

- **분산락**

  분산락으로 인한 추가 리소스 및 오버헤드가 발생하며, 무엇보다 현재 우리 서버는 분산 서버, 분산 DB 환경이 아님

- **비관적 락** (Locking Reads 방식)

  RDBMS에서 제공하는 락으로 정합성을 실시간으로 확실하게 보장하기에 충돌이 잦은 상황에서 적합하다고 판단

#### 결과

비관적 락(Locking Reads) 방식을 적용한 결과, 갱신 손실(Lost Update) 문제를 효과적으로 해결할 수 있었으며, 데이터의 일관성을 유지하며 정확한 데이터를 저장할 수 있었습니다.

### 7) N:M 관계 테이블 Bulk Insert

#### 상황

링크 저장소 가져오기 API 요청 시 소속 링크와 태그들을 전부 복사 후 생성해야 하기에 3N 건의 Insert
쿼리가 발생하였습니다.

#### 실행

이를 해결하기 위해 아래와 같은 이유로 JdbcTemplate의 batchUpdate를 사용하였습니다.

- JPA는 키 생성 전략 Auto Increment인 경우 Bulk Insert를 지원하지 않음

링크와 태그는 N:M 관계로 관계 테이블이 존재하며, 해당 관계 테이블에 Bulk Insert를 하기 위해 아래와 같이 HashMap 자료구조를 이용하여 구현하였습니다.

- 링크와 태그 각각을 Bulk Insert 후 MySQL의 last_insert_id를 이용하여 Insert된 첫 번째 Id를 구한 다. 이후 Insert된 사이즈를 통해 Id값들 계산
- HashMap을 이용하여 복사한 원본 Id를 key로 Insert된 새로운 Id를 value로 매핑
- 원본 링크와 태그의 Id와 생성된 Id를 매핑한 HashMap을 통해 관계 테이블 Bulk Insert

애플리케이션 서버와 DB 서버의 메모리 등을 고려해 Bulk Insert 가능한 데이터 건 수는 현재는 200으로 제한하였습니다.

#### 결과

3N 건의 insert 쿼리를 3번으로 줄였으며, 이를 통해 DB 부하 및 Latency를 줄일 수 있었습니다.

### 8) @MockBean으로 인한 Application Context 초기화 문제 해결

#### 상황

전체를 대상으로 테스트 코드 실행 시 MockBean으로 인해 기존의 Application Context 이용하지 않고
초기화하여 사용하는 문제가 발생하였습니다.
(Application Context가 총 2번 초기화 되는 문제)

## 실행

추상 클래스를 만들어 Integration Test에서 필요한 모든 Test Double 객체를 정의해두고 해당 추상 클래스를 상속받아 사용하는 방식을 통해 해결하였습니다.

## 결과

전체를 대상으로 테스트 코드 실행 시 Application Context의 초기화를 2번에서 1번으로 줄였습니다.