포트폴리오: 문제 해결과 성장을 중심으로

안녕하세요, 기술의 근본 원리를 탐구하고 이를 통해 더 나은 사용자 경험을 만드는 개발자 조승제입니다. 저는 주어진 요구사항을 단순히 구현하는 것을 넘어, 그 과정에서 마주치는 기술적 문제의 본질을 파고들어 최적의 해결책을 찾아내는 과정에서 큰 보람을 느낍니다. 저의 이러한 개발 철학을 잘 보여주는 2가지 핵심 프로젝트 경험을 소개합니다.

Project 1: EchoBloom (네이버 해커톤)

"하루 한 문장, 당신의 목소리로 긍정 문구를 스피킹함으로써 정서 회복, 자기 긍정, 행동 변화를 돕는 AI 기반 긍정 확언 스피킹 서비스"

기간: 2025.08.04 - 2025.08.20

개발 스택:

- Language: Java 21
- Framework: Spring Boot 3.x
- Database: PostgreSQL
- Data Access: Spring Data JPA
- API Documentation: Apidog(Swagger에서 이전)
- AI: Naver CLOVA Studio (HyperCLOVA X), CLOVA Speech Recognition(+STT)
- Authentication: JWT, OAuth 2.0

저장소: https://github.com/Seung-zedd/echo-bloom-server.git

STT 기반 발음 정확도 측정 시스템 구축

Why: 무엇을 해결하고 싶었나요?

EchoBloom 서비스의 핵심은 사용자가 긍정 문구를 직접 말하며 스스로 긍정적인 영향을 미치도록 돕는 것입니다. 이를 위해 사용자의 발음이 원문과 일치하는지 피드백하는 기능이 필수적이었습니다.

하지만 단순히 Naver Clova STT의 결과 텍스트와 원문을 100% 문자열 일치(String.equals)로 비교하는 방식은, "괜찮아"를 "괜차나"처럼 미세하게 다르게 발음할

경우에도 실패로 처리했습니다. 이는 사용자에게 지나친 좌절감을 주어 서비스의 핵심 목적인 정서 회복과 자기 긍정을 오히려 방해하는 치명적인 사용자 경험(UX) 문제였습니다.

따라서 기계적인 정답/오답 판정을 넘어, "얼마나 유사한지"를 유연하게 측정할 새로운 채점 기준이 필요했습니다.

What: 그래서 무엇을 만들었나요?

CS 전공 지식인 '레벤슈타인 거리(Levenshtein Distance)' 알고리즘을 활용하여, 두 문자열 간의 유사도를 측정하고 이를 정확도(Accuracy) 점수로 변환하는 자체 채점 시스템을 개발했습니다.

이 시스템을 통해, STT 결과의 정확도가 사전에 정의한 임계값(Threshold, 80%) 이상일 경우에만 사용자의 발음을 성공으로 인정하는 로직을 구현하여, 긍정적인 사용자 경험을 이끌어냈습니다.

How: 이 목표를 어떻게 해결했나요?

1. 이론의 실무 적용: 편집 거리 알고리즘 선정

문제의 본질은 두 문자열이 얼마나 다른가를 '정량적'으로 측정하는 것이라 판단했습니다.

CS 기초 지식에서 배운 편집 거리(Edit Distance) 개념이 이 문제 해결에 가장 적합하다고 판단했고, 그중 가장 대표적인 레벤슈타인 거리 알고리즘을 채택했습니다. 이 알고리즘은 한 문자열을 다른 문자열로 바꾸기 위해 필요한 삽입, 삭제, 치환 연산의 최소 횟수를 계산해 줍니다.

2. 기술 내재화: 동적 프로그래밍(DP) 기반 알고리즘 직접 구현

외부 라이브러리에 의존하여 단순히 API를 호출하는 대신, 기술의 본질을 깊이 이해하고 응용력을 높이고자 `SpeechService.java` 내에 알고리즘의 핵심 로직을 직접 구현했습니다.

2차원 배열(DP 테이블)을 사용하여, 두 문자열의 각 문자를 순회하며 최소 편집 비용을 누적 계산하는 동적 프로그래밍 방식으로 `calculateLevenshteinDistance` 메서드를 완성했습니다.

3. 정확도 변환 수식 고안 및 비즈니스 로직화

계산된 편집 거리(비용)는 숫자가 클수록 유사하지 않음을 의미하므로, 이를 정확도(점수)로 변환할 필요가 있었습니다.

`정확도 = 1.0 - (편집 거리 / 두 문자열 중 더 긴 길이)` 라는 수식을 고안하여, 0.0(완전 다름)에서 1.0(완전 같음) 사이의 점수로 변환했습니다.

4. 사용자 경험을 고려한 임계값(Threshold) 설정

100% 일치(1.0)는 너무 엄격하고, 50%(0.5)는 너무 관대하여 서비스의 연습 목적을 해칠 수 있다고 판단했습니다.

수차례의 자체 테스트를 통해, 사용자가 성취감을 느끼면서도 정확한 발음을 하도록 유도하는 최적의 균형점으로 MIN_ACCURACY_THRESHOLD를 0.8(80%)로 설정했습니다. 이 임계값은 사소한 실수(편집 거리 1~2회)는 허용하되, 문장의 의미가 달라지는 큰 실수는 걸러내는 합리적인 기준이 되었습니다.

5. 결과 및 배운 점

결과: 사소한 발음 실수에 관대한 피드백을 제공함으로써, 사용자가 좌절 없이 긍정 확언 습관을 이어갈 수 있는 긍정적인 사용자 경험(UX)을 성공적으로 구축했습니다.

배운 점: 알고리즘이라는 이론적 지식이 실제 서비스의 구체적인 비즈니스 문제와 사용자 경험을 해결하는 강력한 도구가 될 수 있음을 체감했습니다. 또한, 80%라는 임계값을 설정하는 과정을 통해, 기술적 구현뿐만 아니라 서비스의 목적과 사용자의 감정을 고려한 기술적 의사결정 역량을 기를 수 있었습니다.

Project 2: sbb_board (개인 프로젝트)

"스프링부트 기초 학습을 위해 제작한 웹 게시판"

- 기간: 2025.03 - 2025.06 (테스트는 2025.11에 수행)
- 개발 스택: Java, Spring Boot, Spring Data JPA, H2 Database, Thymeleaf
- 저장소: https://github.com/Seung-zedd/sbb_board

게시판 목록 조회 API 성능 최적화

Why: 무엇을 해결하고 싶었나요?

게시판의 핵심 기능인 '게시글 목록 조회' 기능 구현 후, 쿼리 로그를 분석하는 과정에서 숨겨진 성능 병목을 발견했습니다. 게시글을 조회하는 1개의 쿼리 이후, 각 게시글에 연결된 댓글이나 작성자 정보를 가져오기 위해 “게시글의 수(N)만큼 추가 쿼리가 발생하는 “N+1 문제”였습니다.
이는 데이터가 수백, 수천 건으로 증가할수록 서버에 심각한 부하를 주어 서비스 전체를 마비시킬 수 있는 잠재적인 장애 포인트였습니다.

What: 그래서 무엇을 만들었나요?

N+1 문제를 해결하기 위해 Fetch Join을 적용하여 연관된 엔티티를 단 한 번의 쿼리로 함께 조회하도록 JPQL을 개선했습니다. 나아가, Fetch Join의 잠재적 단점을 보완할 더 나은 대안을 탐구하기 위해 세미 조인(Semi-Join)과 스트림 API를 조합하는 방식과 기존 방식의 성능을 객관적인 데이터로 비교하는 테스트 환경을 설계했습니다.

How: 이 목표를 어떻게 해결했나요?

1. 문제 발견 및 Fetch Join을 통한 1차 해결

- `@SpringBootTest` 환경에서 API를 테스트하며 실행되는 쿼리 로그를 통해 N+1 문제를 명확히 인지했습니다.
- 가장 널리 알려진 해결책인 Fetch Join을 적용하여 즉시 로딩이 필요한 연관 관계의 엔티티를 함께 조회하도록 Repository 코드를 수정하여 1차적으로 문제를 해결했습니다.

2. 이론에 기반한 심화 탐구 및 대안 가설 수립

Fetch Join이 야기할 수 있는 카테시안 곱(Cartesian Product) 문제와 페이징 API와의 충돌 가능성 등 한계를 학습했습니다.
- 카테시안 곱: 1:N 관계에서 게시글이 답변 개수만큼 중복 전송되어 네트워크 비용 증가
- 페이징 한계: DB가 아닌 메모리에서 페이징이 처리되어 setMaxResults가 의도대로 작동하지 않음

이때, 기초 데이터베이스 이론을 공부하며 배웠던 '세미 조인(Semi-Join)'이 조인 연산의 비용을 크게 줄여준다는 사실을 떠올렸습니다.
- 세미 조인의 핵심 원리: 조인에 필요한 최소한의 키(ID) 정보만 먼저 추출하여 데이터 전송량을 줄이는 관계 대수 연산

이 이론적 지식을 바탕으로, '필요한 ID 목록만 먼저 조회하고(세미 조인 원리 응용), 조회된 독립적인 데이터들을 애플리케이션단에서 스트림 API로 조합하면 Fetch Join보다 효율적일 수 있다'는 새로운 가설을 수립했습니다.

스트림 API를 선택한 이유:
- DB 레벨 최적화의 한계 보완: 세미 조인 전략으로 3개의 독립적인 쿼리 결과를 받아오면, 이를 조합하는 과정이 필수적으로 필요합니다.
    - O(n) 시간 복잡도 보장: Collectors.toMap()으로 답변 개수를 Map 구조로 변환하여 O(1) 조회가 가능하게 만들고, stream().map()으로 최종 DTO를 조립합니다. 전통적인 이중 반복문 방식은 $O(n^2)$의 복잡도를 가지지만, 스트림 API와 Map을 활용하면 O(n)으로 개선됩니다.

- 선언적 코드: 함수형 프로그래밍 패러다임을 통해 "무엇을 할 것인가"에 집중하여 코드 가독성과 유지보수성을 높였습니다.

3. 객관적 검증을 위한 테스트 설계

두 방식의 성능을 정확히 비교하기 위해, 실제 프로덕션 환경을 시뮬레이션하는 테스트 인프라를 구축했습니다:
- Local MySQL 8.0 환경 구성 (Docker 대신 로컬 인스턴스 사용으로 I/O 오버헤드 제거)
- 10,000개 게시글, 100,000개 답변의 대용량 더미 데이터 생성 (게시글당 평균 5~20개 답변)
- Hibernate Statistics API를 활용한 정밀한 성능 지표 수집 (쿼리 수, 엔티티 로드 수, 실행 시간)

결과

4. 성능 테스트 결과

1) 더미 데이터 1,000건 기준

```
: 최종 비교 결과
: ================================================================================
: 방식                    |   실행시간(ms) |       쿼리횟수 | Entity Load
: --------------------------------------------------------------------------------
: Semi Join               |         20 |          3 |         40
: Fetch Join              |        683 |          1 |      14277
: Lazy Loading (N+1)      |         75 |         21 |        309
: ================================================================================
:

: ✅ Semi Join이 Fetch Join 대비 97.1% 빠릅니다!
: 쿼리 횟수: Semi Join 3회 vs Fetch Join 1회
: Entity Load: Semi Join 40개 vs Fetch Join 14277개
: [rollback] | 0 ms |
```

- Semi Join: 20ms, 쿼리 3회, 엔티티 로드 40개
- Fetch Join: 683ms, 쿼리 1회, 엔티티 로드 14,277개
- Semi Join이 Fetch Join 대비 97.1% 빠른 성능을 보이며 가설이 검증되었습니다

2) 더미 데이터 5,000건 기준

```
: 최종 비교 결과
: ================================================================================
: 방식                    |   실행시간(ms) |       쿼리횟수 | Entity Load
: --------------------------------------------------------------------------------
: Semi Join            |         16 |          3 |         40
: Fetch Join           |       1430 |          1 |      68125
: Lazy Loading (N+1)   |         54 |         21 |        304
: ====================+===========================================================
:

: ✅ Semi Join이 Fetch Join 대비 98.9% 빠릅니다!
: 쿼리 횟수: Semi Join 3회 vs Fetch Join 1회
: Entity Load: Semi Join 40개 vs Fetch Join 68125개
: [rollback] | 0 ms |
```

- Semi Join: 16ms, 쿼리 3회, 엔티티 로드 40개
- Fetch Join: 1,430ms, 쿼리 1회, 엔티티 로드 68,125개
- Semi Join이 Fetch Join 대비 98.9% 빠른 성능을 보이며 가설이 검증되었습니다

3) 더미 데이터 10,000건 기준

```
: 최종 비교 결과
: ================================================================================
: 방식                    |   실행시간(ms) |       쿼리횟수 | Entity Load
: --------------------------------------------------------------------------------
: Semi Join            |         70 |          3 |         40
: Fetch Join           |       6506 |          1 |     136518
: Lazy Loading (N+1)   |        119 |         21 |        274
: ================================================================================
:

: ✅ Semi Join이 Fetch Join 대비 98.9% 빠릅니다!
: 쿼리 횟수: Semi Join 3회 vs Fetch Join 1회
: Entity Load: Semi Join 40개 vs Fetch Join 136518개
: [rollback] | 0 ms |
```

- Semi Join: 70ms, 쿼리 3회, 엔티티 로드 40개
- Fetch Join: 6,506ms, 쿼리 1회, 엔티티 로드 136,518개

- Semi Join이 Fetch Join 대비 98.9% 빠른 성능을 보이며 가설이 검증되었습니다

5. 데이터 규모별 성능 분석

데이터셋	Semi Join	Fetch Join	성능 개선율	Entity Load 비율
1,000건	20ms	683ms	97.1%	40 vs 14,277 (357배)
5,000건	16ms	1,430ms	98.9%	40 vs 68,125 (1,703배)
10,000건	70ms	6,506ms	98.9%	40 vs 136,518 (3,413배)

5.1.핵심 인사이트

- 스케일 효율성: Semi Join은 데이터 증가에도 안정적 성능 유지 (16-70ms)
  - 이유: 페이지네이션(20개 제한)으로 항상 동일한 엔티티(40개)만 로드

- Cartesian Product의 재앙: Fetch Join은 데이터가 10배 증가할 때 성능이 9.5배 악화 (683ms → 6,506ms)
  - 이유: 질문당 평균 10개 답변 → 10배 중복 전송 (14,277개 → 136,518개)

- 답변 수와의 상관관계: 답변이 많을수록 (평균 5-20개) Fetch Join의 Cartesian Product 문제가 극대화되어 Semi Join의 성능 우위가 더욱 명확해졌습니다.

- 예외 시나리오: 답변이 1-2개로 매우 적은 경우, Fetch Join의 단일 쿼리 장점(네트워크 왕복 1회)이 Semi Join의 3회 쿼리 비용을 상쇄할 수 있습니다 (추가 검증 필요)

6. 실무 적용 가이드 (When to Use Which)

각 쿼리 전략의 Trade-off를 기반으로 한 실무 의사결정 기준:

상황	권장 방식	이유
1:N 관계에서 N이 많은 경우(평균 5개 이상)	Semi Join	Cartesian Product 방지로 네트워크 비용 최소화
페이징이 필수인 API(게시판, 목록 조회)	Semi Join	DB 레벨에서 정확한 페이징 보장
N이 매우 적은 경우(평균 1-2개)	Fetch Join 고려	단일 쿼리로 네트워크 왕복 감소 (단, 검증 필요)

프로덕션 환경	Lazy Loading 금지	N+1 문제로 서비스 마비 위험

6.1.의사결정 프로세스

1. 평균 연관 엔티티 수 확인 (5개 이상 → Semi Join)
2. 페이징 요구사항 확인 (필수 → Semi Join)
3. 성능 테스트로 최종 검증 (본 프로젝트처럼 Hibernate Statistics 활용)

7. 결론

본 성능 테스트를 통해 다음을 달성했습니다:

- 이론의 실증: 데이터베이스 세미 조인 원리와 Java Stream API를 결합한 하이브리드 전략의 우수성을 정량적으로 입증 (97-99% 성능 개선)

- 엔지니어링 사고 프로세스:단순히 알려진 해결책(Fetch Join)을 적용하는 것을 넘어, 기초 이론에 기반한 독자적인 가설 수립 → 실험 설계 → 검증의 과정을 완수

- 실무 적용 가능성: Trade-off 분석을 통해 실제 프로덕션 환경에서의 의사결정 기준 확립

예상 질문 대비

Q: "Fetch Join이 더 나은 경우도 있지 않나요?"

A: 네, 1:N 관계에서 N이 1-2개로 매우 적을 때는 Fetch Join의 단일 쿼리 장점이 있을 수 있습니다. 하지만 본 테스트의 게시판 시나리오(평균 10개 답변)에서는 Cartesian Product 비용이 압도적이어서 Semi Join이 97-99% 빠릅니다.

Q: "3개 쿼리 vs 1개 쿼리, 네트워크 레이턴시는?"

A: 로컬 MySQL 환경에서는 레이턴시가 1ms 미만이지만, 실제 프로덕션(원격 DB)에서 3회 왕복 비용(~15ms)이 발생합니다. 그러나 Fetch Join의 Cartesian Product로 인한 데이터 전송량(136,518 vs 40 엔티티)이 훨씬 큰 병목입니다. 테스트 결과(Semi 70ms vs Fetch 6,506ms)가 이를 증명합니다.

Q: "왜 1K, 5K, 10K를 선택했나요?"

A: 스케일 증가에 따른 성능 추세를 관찰하기 위함입니다. 실제로 1K→10K(10배 증가) 시 Fetch Join은 9.5배 느려졌지만, Semi Join은 페이지네이션으로 3.5배만 느려져 스케일 효율성을 입증했습니다.