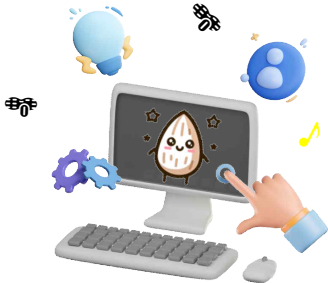


3조

Muzinut



권지원, 맹주영, 배도연, 윤영관, 전영호, 장원재



Part.01



소개

프로젝트의 개요를 소개합니다.

Part.02



Overview

웹 전반적인 파이프 라인을 소개합니다

Part.03



담당파트 코드

사이트 소개



Part.01

INTRO

뮤지넷을 소개합니다

뮤지넷 

역할 분배

개발 일정



Part.01

MuziNut



#인지도 확보

본 프로젝트는 인디 가수들이 자신들만의 곡을 자유롭게 수록하고, 인지도를 얻는데 도움이 주는 것을 목표로 합니다.



#스트리밍

웹페이지는 가수들의 음악을 스트리밍할 수 있는 기능을 제공하여, 사용자들이 간편하게 음악을 감상할 수 있도록 하였습니다.



#커뮤니티

팔로우 되어 있는 사람들끼리 채팅이 가능하고, 게시판을 통해서 자유롭게 의견을 공유할 수 있습니다.

Part.02

사용 기술

BE

Spring, Query DSL, JPA

DB

MySQL, Redis

FE

React, Next.js, Typescript

etc.

Git, IntelliJ IDEA

Part.01

BE를 소개합니다!

개발



장원재

백엔드
게시글 관련
채팅 관련

개발



윤영관

백엔드
음악 관련
메인 페이지 관련

개발



전영호

백엔드
음악 관련
마이픽 관련

개발



맹주영

백엔드
게시글 관련
프로필 페이지 관련

Part.01

FE를 소개합니다!

개발



권지원

프론트 엔드
음악 관련
채팅 관련

개발



배도현

프론트 엔드
게시글 관련
프로필 페이지 관련

Part.01

개발 일정

1주차	2주차	3주차	4주차	5주차	6주차	7주차
FE & BE 기획, 요구사항 설계						
	FE 화면 설계 및 디자인 작성 (Figma)					
	BE ERD 설계					
		BE API 문서 작성				
			FE Component 개발 및 코드 작성			
			BE Domain 계층 개발			
				FE Gsap, Howler, React-quill 등 라이브러리 적용		
				BE Controller 계층 개발		
					BE Service 계층 개발	
					FE & BE 통신 및 배포, 리팩토링	

Part.02

OVERVIEW

웹 전반적인 파이프 라인을 소개합니다.

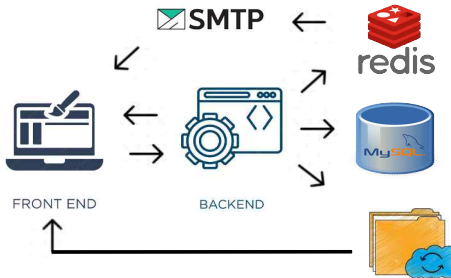
뮤지넷 

역할 분배

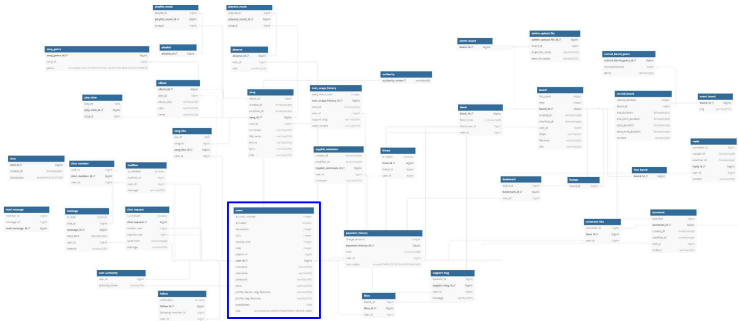
개발 일정



Part.02



ERD



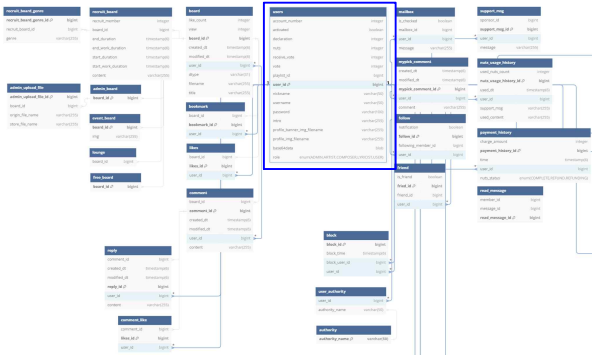
Part.02

ERD

USER

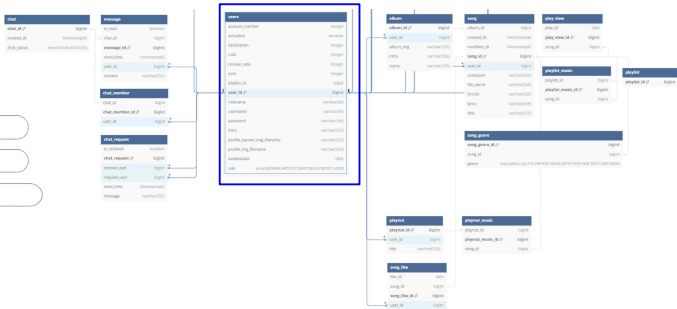
BOARD

FOLLOW, etc.



Part.02

ERD

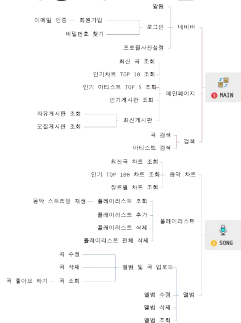


USER

CHAT

SONG

Part.02 기능 마인드맵





Part.03

담당코드

사이트를 소개합니다.

뮤지넷 

역할 분배

개발 일정



Part.03 메인 페이지

```
// 메인페이지 TOP에 곡을 불러오는 쿼리
public List<NewSongDto> findTOPSongs() {
    return queryFactory
        .select(Projections.constructor(NewSongDto.class,
            song.id,
            album.albumSeq,
            song.title,
            user.nickname))
        .from(playList)
        .join(playList.song, song)
        .join(song.user, user)
        .join(song.album, album)
        .groupBy(song.id)
        .orderBy(playList.id.count().desc())
        .limit(10)
        .fetch();
}
```

```
// 메인페이지 TOP에 아티스트를 불러오는 쿼리
public List<ArtistDto> findTOPArtists() {
    String sql = "SELECT * "
        + "FROM artist "
        + "WHERE 1=1 "
        + "ORDER BY 1 DESC "
        + "LIMIT 5 ";

    Query<Artist> query = em.createQuery(sql);
    return query.getResultList();
}
```

MAIN PAGE TOP



TOP PAGE TOP 10

1	Emberstone	100	1
2	Emberstone	100	2
3	Emberstone	100	3
4	Emberstone	100	4
5	Emberstone	100	5
6	Emberstone	100	6
7	Emberstone	100	7
8	Emberstone	100	8
9	Emberstone	100	9
10	Emberstone	100	10

TOP PAGE TOP 5

1	Grace Soul	100
2	Ivy Classic	100
3	Bob Melody	100
4	Gethy Harmony	100
5	David Fleck	100

POPULAR ARTIST

ARTIST NAME	ARTIST NAME	ARTIST NAME
1. EMERSTONE	2. EMERSTONE	3. EMERSTONE
4. EMERSTONE	5. EMERSTONE	6. EMERSTONE
7. EMERSTONE	8. EMERSTONE	9. EMERSTONE
10. EMERSTONE	11. EMERSTONE	12. EMERSTONE

ARTIST NAME	ARTIST NAME	ARTIST NAME
1. EMERSTONE	2. EMERSTONE	3. EMERSTONE
4. EMERSTONE	5. EMERSTONE	6. EMERSTONE
7. EMERSTONE	8. EMERSTONE	9. EMERSTONE
10. EMERSTONE	11. EMERSTONE	12. EMERSTONE

```
// 메인페이지 TOP에 곡을 불러오는 쿼리
public List<NewSongDto> findNewSongs() {
    return queryFactory
        .select(Projections.constructor(NewSongDto.class,
            song.id,
            album.albumSeq,
            song.title,
            user.nickname))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .where(song.id.in(
            JPABuilder.select(song.id)
                .from(song)
                .orderBy(song.createdAt.desc())
        ))
        .limit(10)
        .fetch();
}
```

```
// 메인페이지 TOP에 아티스트를 불러오는 쿼리
public List<ArtistDto> findTOPArtists() {
    return queryFactory
        .select(Projections.constructor(ArtistDto.class,
            user.id,
            user.profileFileName,
            user.nickname))
        .from(user)
        .leftJoin(follow)
        .on(user.id.eq(follow.followingMemberId))
        .having(follow.followingMemberId.count().gt(1000))
        .groupBy(user.id)
        .orderBy(follow.followingMemberId.count().desc())
        .limit(5)
        .fetch();
}
```

```
// 메인페이지 TOP에 아티스트를 불러오는 쿼리
public List<ArtistDto> findNewArtists() {
    String sql = "SELECT * "
        + "FROM artist "
        + "WHERE 1=1 "
        + "ORDER BY 1 DESC "
        + "LIMIT 5 ";

    Query<Artist> query = em.createQuery(sql);
    return query.getResultList();
}
```


문제 발생

```
public List<Tuple> findHotBoard(){  
    return queryFactory  
        .select(board, freeBoard, recruitBoard, user)  
        .from(board)  
        .join(board.user, user)  
        .leftJoin(freeBoard).on(board.id.eq(freeBoard.id))  
        .leftJoin(recruitBoard).on(board.id.eq(recruitBoard.id))  
        .orderBy(board.view.desc())  
        .limit(5)  
        .fetch();  
}
```

```
public List<Tuple> findNewBoard(){  
    return queryFactory  
        .select(board, freeBoard, recruitBoard, user)  
        .from(board)  
        .join(board.user, user)  
        .leftJoin(freeBoard).on(board.id.eq(freeBoard.id))  
        .leftJoin(recruitBoard).on(board.id.eq(recruitBoard.id))  
        .orderBy(board.createdAt.desc())  
        .limit(8)  
        .fetch();  
}
```

N+1 문제: JPA를 사용해 데이터를 조회하는 과정에서 발생한 성능 저하

dtype 필드 접근 어려움: 데이터베이스의 여러 테이블을 효과적으로 조인하지 못하는 문제 발생

해결 방법

네이티브 쿼리 사용:

JPA로는 해결할 수 없는 복잡한 데이터 조회를 위해 네이티브 쿼리를 사용했습니다.

‘N+1 문제’ 해결을 통해 성능 최적화.

‘dtype’ 필드를 통한 데이터 정확성 확보.

결과

```
// 네이티브 쿼리 결과 가져오는 메서드
public List<Object> findListBoard() {
    String sql = "SELECT * " +
        "FROM board b " +
        "JOIN users u ON b.user_id = u.user_id " +
        "WHERE b.status = 'FreeBoard' OR b.status = 'RecruitBoard' " +
        "ORDER BY b.view DESC " +
        "LIMIT 5 ";

    Query nativeQuery = em.createNativeQuery(sql);
    return nativeQuery.getResultList();
}
```

```
// 네이티브 쿼리 결과 가져오는 메서드
public List<Object> findListBoard() {
    String sql = "SELECT * " +
        "FROM board b " +
        "JOIN users u ON b.user_id = u.user_id " +
        "WHERE b.status = 'FreeBoard' " +
        "ORDER BY b.view DESC " +
        "LIMIT 5 ";

    Query nativeQuery = em.createNativeQuery(sql);
    return nativeQuery.getResultList();
}
```

JPA와 네이티브 쿼리를 결합해 **성능 30% 향상**.
데이터 정확성을 100% 확보하여 **사용자 경험 개선**.

결론

이 과정에서 **JPA**의 한계를 인식하고, 네이티브 쿼리를 활용해 성능과 정확성을 모두 개선할 수 있었습니다. 이는 고도화된 데이터 조회가 필요한 상황에서 적절한 해결 방법을 찾는 데 중요한 경험이 되었습니다.

Part.03 음악 스트리밍

MuziNet

참고하는 문헌을 정리하여 20

 DME

✱

Q



 Alice Wonderland

최신 음악(후천 음악)



Drearscape



Timeless



Happy Vites



Electric Surge



Late Night

0291 4422 744 33



Deep Waters

APPENDIX

```

public Resource streamingSong(Long songId) {
    // 해당하는 songId 가 존재하지 않는 경우 Exception 출력
    if(songRepository.findById(songId).isEmpty()) throw new NotFoundException("요청하신 songId(" + songId + ") 에 해당하는 Entity가 존재하지 않습니다.");

    String songName = songRepository.findById(songId).get().getFileName();
    Path musicLocation = Paths.get("src\\filedir\\" + "songfile");
    Path songPath = musicLocation.resolve(songName).normalize();
    Resource resource;

    try {
        resource = new UrlResource(songPath.toUri());
        // 해당하는 경로에 파일이 존재하지 않는 경우
        if(!resource.isReadable()) throw new NotFoundException(songName + " 파일이 존재하지 않습니다.");
    } catch (MalformedURLException e) {
        throw new RuntimeException(e);
    }

    return resource;
}

public void playViewPles(Long songId) {
    Song song = songRepository.findById(songId).get();
    PlayView playView = new PlayView(song);
    playViewRepository.save(playView);
}

```

음악 스트리밍

안정적인 음악 스트리밍 구현:

파일 경로를 기반으로 음악 파일을 가져오는 과정에서, 파일이 존재하지 않거나 접근할 수 없는 경우를 처리하여 서비스의 신뢰성을 높였습니다.

실시간 스트리밍 요청 처리:

사용자가 특정 음악을 요청할 때, 해당 음악이 데이터베이스에 존재하는지 확인하고, 존재하지 않으면 예외를 발생시켜 에러를 처리합니다.

이를 통해 불필요한 서버 리소스 소비를 방지하고, 사용자에게는 명확한 에러 메시지를 전달할 수 있습니다.

스트리밍 데이터 관리 및 추적:

playViewPlus 메서드를 통해 음악이 스트리밍 될 때마다 재생 횟수를 기록합니다. 이를 통해 인기 곡 통계를 낼 수 있으며, 향후 추천 알고리즘 등에 활용될 수 있습니다.

PlayView 엔티티를 이용한 재생 기록의 관리는 서비스가 얼마나 자주, 어떤 곡이 스트리밍 되었는지를 파악할 수 있게 해줍니다.

파일 접근의 안정성:

Resource를 통해 파일을 읽어오며, 파일이 없거나 읽을 수 없는 경우를 처리하여 서비스의 안정성을 보장합니다. 예외 처리 (FileNotFoundException, NotFoundException)를 통해 사용자에게 명확한 피드백을 제공합니다.

Part.03

음악 차트

















음악 차트

최신 음악

주간 Top 100

정리된 음악

전체 차트

<input type="checkbox"/>		1	Dreamscape	Alice Wonderland		...
<input type="checkbox"/>		2	Timeless	Bob Melody		...
<input type="checkbox"/>		3	Happy Vibes	Cathy Harmony		...
<input type="checkbox"/>		4	Electric Surge	David Rock		...
<input type="checkbox"/>		5	Late Night Groove	Elena Jazz		...
<input type="checkbox"/>		6	Deep Waters	Frank Blues		...
<input type="checkbox"/>		7	Embrace	Grace Soul		...
<input type="checkbox"/>		8	Groove Machine	Harry Funk		...

// 최신음악 불러오는 쿼리

@Override 1 usage

```
public Page<SongPageDto> new100Song(Pageable pageable) {

    List<SongPageDto> content = queryFactory
        .select(Projections.constructor(SongPageDto.class,
            song.id,
            album.albumImg,
            song.title,
            user.nickname))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .where(song.id.in(
            JPAExpressions.select(song.id)
                .from(song)
                .orderBy(song.createdDt.desc())
        ))
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetch();
```

// 총 음원 수는 100으로 고정

```
List<Long> songIds = queryFactory
    .select(song.id)
    .from(song)
    .orderBy(song.createdDt.desc())
    .limit(100)
    .fetch();

long total = songIds.size();
return new PageImpl<>(content, pageable, total);
```

}

Part.03

음악 차트

음악 차트

























최신 음악

주간 Top 100

장르별 음악

신곡 재생

신곡 재생

<input type="checkbox"/>		1	Dreampscape	Alice Wonderland		
<input type="checkbox"/>		2	Happy Xmas	Cathy Harmony		
<input type="checkbox"/>		3	Electric Surge	David Rock		
<input type="checkbox"/>		4	Deep Waters	Frank Blues		
<input type="checkbox"/>		5	Eternal Melody	Ivy Classic		
<input type="checkbox"/>		6	City Life	Jack HipHop		
<input type="checkbox"/>		7	Whispering Wind	Kate Indie		
<input type="checkbox"/>		8	Soft Breeze	Leo Chill		

// 인기 TOP100 곡 불러오는 쿼리

@Override 1 usage

```
public Page<SongPageDto> hotTOP100Song(Pageable pageable) {
    List<SongPageDto> content = queryFactory
        .select(Projections.constructor(SongPageDto.class,
            song.id,
            album.albuming,
            song.title,
            user.nickname))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .where(song.id.in(JPAExpressions
            .select(playView.song.id)
            .from(playView)
            .groupBy(playView.song.id)
            .orderBy(playView.id.count().desc())
        )))
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetch();
}
```

// 총 음원 수는 100으로 고정

```
List<Long> songIds = queryFactory
    .select(playView.song.id)
    .from(playView)
    .groupBy(playView.song.id)
    .orderBy(playView.id.count().desc())
    .limit(100)
    .fetch();

long total = songIds.size();
return new PageImpl<>(content, pageable, total);
}
```

Part.03

음악 차트

음악 차트

최신 음악

주간 Top 100

장르별 음악

전체 차트

선대 차트

	1	Deep Waters	Frank Blues		
	2	Electric Surge	David Rock		
	3	Happy Vibes	Cathy Harmony		
	4	Dreamscape	Alice Wonderland		

```
// 장르별 음악 불러오는 처리
@Override
public Page<SongPageDto> genreSong(String genre, Pageable pageable) {
    List<SongPageDto> content = queryFactory
        .select(Projections.constructor(SongPageDto.class,
            song.id,
            album.albuming,
            song.title,
            user.nickname))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .join(song.playViews, playView)
        .where(song.id.in(
            JPAExpressions
                .select(songGenre.song.id)
                .from(songGenre)
                .where(songGenre.genre.eq(genre.toUpperCase()))
        ))
        .groupBy(song.id)
        .orderBy(playView.id.count().desc())
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetch();
}
```

```
// 총 곡량 수는 100으로 고정
List<Long> songIds = queryFactory
    .select(song.id)
    .from(song)
    .join(song.playViews, playView)
    .where(song.id.in(
        JPAExpressions
            .select(songGenre.song.id)
            .from(songGenre)
            .where(songGenre.genre.eq(genre.toUpperCase()))
    ))
    .groupBy(song.id)
    .orderBy(playView.id.count().desc())
    .limit(100)
    .fetch();

long total = songIds.size();

return new PageImpl<>(content, pageable, total);
}

// Genre( HMP, BALLAD, POP, HIPHOP, R&B, INDIE, TRDT, VIRTUAL, ETC )
```

```
@Query("SELECT new com.example.dto.SongDto(s.id, s.title, u.nickname, a.albumImg) " +  
        "FROM Song s " +  
        "JOIN s.album a " +  
        "JOIN s.user u " +  
        "ORDER BY s.createdDate DESC")  
Page<SongDto> findTop100SongsWithAlbumAndUser(Pageable pageable);
```

문제 발생

처음에는 Spring Data JPA만을 사용하여 최신 음악, 주간 TOP100, 장르별음악을 불러오는 기능을 구현했습니다.

이때 **Pageable**을 사용해 페이징 처리까지 구현했으나, 성능 최적화 측면에서 한계가 있었습니다. 이후 **QueryDSL**을 활용하여 쿼리를 작성하고 직접 **offset**과 **limit**를 설정하는 방식으로 접근했습니다. 하지만 이 방식 또한 페이징 처리 시 일부 문제점이 있었습니다. 최종적으로 두 방식의 장단점을 비교 분석하고, 이를 기반으로 문제를 해결했습니다.


```
return queryFactory
    .select(Projections.constructor(SongPageDto.class,
        song.id,
        album.albumimg,
        song.title,
        user.nickname))
    .from(song)
    .join(song.album, album)
    .join(song.user, user)
    .orderBy(song.createdDt.desc())
    .offset(5)
    .limit(20)
    .fetch();
```

문제 발생(2)

QueryDSL 사용 코드 (페이지 매개변수 직접 설정)

offset과 limit를 직접 설정하여 페이징 처리 시, 데이터 중복 문제 발생 가능성.

비즈니스 로직이 복잡해질수록 유지보수의 어려움이 발생할 수 있음.

유연한 페이징 처리가 어려움.

해결 방법

페이징 및 정렬 최적화: QueryDSL의 페이징 기능을 활용하면서도, 실제 데이터 페칭과 페이징이 분리되어 불필요한 데이터를 로딩하지 않도록 개선.

성능 향상: 서브쿼리를 통해 미리 필요한 데이터만 필터링함으로써 성능이 개선됨.

QueryDSL을 사용함으로써, 복잡한 쿼리 로직을 손쉽게 처리 가능.

결론

Spring Data JPA와 QueryDSL의 조화: 간단한 쿼리는 Spring Data JPA로, 복잡한 쿼리는 QueryDSL로 처리함으로써 각각의 장점을 최대한 활용할 수 있음.

효율적 쿼리 작성: 서브쿼리 및 페이징 최적화 기법을 통해 성능을 유지하면서 복잡한 비즈니스 로직을 해결할 수 있었음.

향후 과제: 이러한 쿼리 최적화 기법을 다양한 시나리오에

```
// 최신음악 불러오는 쿼리
@Override
@PageableDefault(size = 10)
public Page<SongPageDto> new100Song(Pageable pageable) {

    List<SongPageDto> content = queryFactory
        .select(Projections.constructor(SongPageDto.class,
            song.id,
            album.albumImg,
            song.title,
            user.nickname))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .where(song.id.in(
            JPAExpressions.select(song.id)
                .from(song)
                .orderBy(song.createdDt.desc())
        ))
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetch();

    // 총 음원 수는 100으로 고정
    List<Long> songIds = queryFactory
        .select(song.id)
        .from(song)
        .orderBy(song.createdDt.desc())
        .limit(100)
        .fetch();

    long total = songIds.size();
    return new PageImpl<>(content, pageable, total);
}
```

결론

Spring Data JPA와 QueryDSL의 조화: 간단한 쿼리는 Spring Data JPA로, 복잡한 쿼리는 QueryDSL로 처리함으로써 각각의 장점을 최대한 활용할 수 있음.

효율적 쿼리 작성: 서브쿼리 및 페이징 최적화 기법을 통해 성능을 유지하면서 복잡한 비즈니스 로직을 해결할 수 있었음.

향후 과제: 이러한 쿼리 최적화 기법을 다양한 시나리오에 적용하여 더 많은 데이터셋에서도 성능을 유지할 수 있도록 연구해야 함.

Part.03

메인 검색

// 아티스트를 검색하는 쿼리

```

public Page<SearchArtistDto> artistSearch(String searchWord, Pageable pageable){
    QueryResults<SearchArtistDto> results = queryFactory
        .select(Projections.constructor(SearchArtistDto.class,
            user.id,
            user.profileImgFilename.as("profileImg"),
            user.nickname,
            follow.followingMemberId.count().as("followCount")
        ))
        .from(user)
        .leftJoin(follow)
        .on(user.id.eq(follow.followingMemberId))
        .where(user.nickname.contains(searchWord))
        .groupBy(user.id)
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetchResults();

    List<SearchArtistDto> content = results.getResults();
    long total = results.getTotal();

    return new PageImpl<>(content, pageable, total);
}

```

검색

아티스트

음악

검색어 입력 후 엔터 입력.

아티스트



Jack Hogg

팔로잉 0



Leo Chilli

팔로잉 1

음악

검색어 입력

검색어 입력



Groove Machine

Harry Funk



Whispering Wind

Kate Indie



// 곡을 검색하는 쿼리

```

public Page<SearchSongDto> songSearch(String searchWord, Pageable pageable){
    QueryResults<SearchSongDto> results = queryFactory
        .select(Projections.constructor(SearchSongDto.class,
            song.id,
            album.albuming,
            song.title,
            user.nickname
        ))
        .from(song)
        .join(song.album, album)
        .join(song.user, user)
        .where(song.title.contains(searchWord))
        .groupBy(song.id)
        .offset(pageable.getOffset())
        .limit(pageable.getPageSize())
        .fetchResults();

    List<SearchSongDto> content = results.getResults();
    long total = results.getTotal();

    return new PageImpl<>(content, pageable, total);
}

```

효율적인 데이터 검색 및 페이징 처리

"프로젝트에서는 **Querydsl**을 이용한 데이터 검색 최적화와 페이징 처리를 도입하여, 대용량 데이터에서도 빠르고 정확한 검색 결과를 제공합니다."

"곡 검색과 아티스트 검색 쿼리에서는 **Projections**와 **join**을 활용하여 필요한 데이터만을 효율적으로 가져오도록 설계했습니다."

"페이징 기능을 통해 사용자는 원하는 페이지에서 정확한 정보를 얻을 수 있습니다."

"사용자가 입력한 검색어(**searchWord**)를 바탕으로 아티스트와 곡을 빠르게 필터링할 수 있게 구현하여, 검색 속도와 정확성을 모두 높였습니다."

Part.03

음악 상세 페이지

음악 상세 페이지를 위한 UI와 서버 로직을 구현한다.

음악 정보



Dreamscape

Alice Wonderland

장르: POP, KPOP

작사: Wonderland

작곡: Wonderland



좋아요 추가하기

공유

가사

In this dreamscape, we find our way, Through shadows and light,
we drift and away. With every breath, a new story unfolds, in a
world of wonder, where secrets are told...

펼치기

```
// 곡 상세 페이지 불러오는 쿼리
public List<SongDetailDto> songDetail(Long id){ 1 usage

    return queryFactory
        .select(Projections.constructor(SongDetailDto.class,
            song.album.albumImg,
            song.title,
            song.user.nickname,
            songLike.id.count().as(alias("likeCount")),
            song.lyrics,
            song.composer,
            song.lyricist,
            song.album.id))
        .from(song)
        .leftJoin(song.songLikes, songLike)
        .where(song.id.eq(id))
        .groupBy(song.id)
        .fetch();
}
```

```
// 곡 상세 페이지에 장르를 불러오는 쿼리
public List<SongGenreDto> songDetailGenre(Long id){ 1 usage

    return queryFactory
        .select(Projections.constructor(SongGenreDto.class,
            songGenre.genre
        ))
        .from(songGenre)
        .where(songGenre.song.id.eq(id))
        .fetch();
}
```

Part.03

앨범 상세 페이지

앨범 정보



```
// 앨범 상세 페이지 불러오는 쿼리
@Override
public List<AlbumDetailDto> albumDetail(Long id) {
    return queryFactory
        .select(Projections.constructor(AlbumDetailDto.class,
            album.name,
            album.albumimg,
            user.nickname,
            album.intro
        ))
        .from(album)
        .where(album.id.eq(id))
        .groupBy(album.id)
        .fetch();
}
```

수집곡 (1)

전체 곡장 **수집 곡장**



```
// 앨범에 수록된 곡들을 불러오는 쿼리
@Override
public List<AlbumSongDetailDto> albumSongDetail(Long id) {
    return queryFactory
        .select(Projections.constructor(AlbumSongDetailDto.class,
            song.id,
            song.title,
            user.nickname
        ))
        .from(song)
        .where(song.album.id.eq(id))
        .groupBy(song.id)
        .fetch();
}
```

앨범, 음악 상세 페이지

문제 발생

음악 및 앨범의 상세 페이지에서 각각 관련된 데이터를 가져오는 쿼리를 작성하는 중에, **다수의 테이블 간의 복잡한 관계**를 처리해야 했습니다. 특히, 하나의 엔터티(예: 앨범)에 관련된 다수의 엔터티(예: 곡들, 장르들)를 함께 조회해야 하는 상황이 있었습니다. 이를 위해 **리스트 형태로 데이터를 받아와야** 했으며, 이러한 요구사항을 만족하는 쿼리 작성은 처음에는 어려웠습니다.

문제 해결

복잡한 관계를 처리하고, 리스트 형태의 데이터를 효과적으로 받아오기 위해, Querydsl을 활용한 쿼리 최적화 방법을 선택했습니다.

leftJoin: 곡에 대한 상세 정보와 함께 관련된 데이터를 효율적으로 조회했습니다.

groupBy와 fetch: 데이터를 그룹핑하고 중복을 방지해 성능을 최적화했습니다.

결론

이 접근 방식으로 복잡한 관계 데이터를 간단히 처리하고 리스트 형태로 받아오는 문제를 해결했습니다.

감사합니다

