이완수
아키텍트팀
2020년 11월 26일

JPA를 사용한 조회 성능 최적화
2020년 연구과제

목차

1. 과제 선정 이유

Java 진영에서 JPA는 가장 인기있는 기술 중 하나 입니다. JPA를 사용하면 SQL에 의존적이지 않은 객체지향 중심의 개발이 가능합니다.

또한, 요즘 개발 트렌드인 MSA와 DDD 기반 개발에 용이합니다. 이러한 JPA의 장점들은 개발자들에게 높은 생산성을 제공합니다.

하지만 JPA는 많은 장점을 가지고 있지만 그만큼 러닝 커브가 높은 편입니다. 특히 JPA를 사용할 때 많은 개발자들이 조회 성능을 최적화 할 때 많은 어려움을 겪습니다.

저 역시 아키텍트팀 자산 관리 시스템을 개발하면서 위와 어려움을 겪었고 이를 해결하면서 JPA를 사용할때 조회 성능 최적화 기법들을 익힐 수 있었습니다.

따라서, 개발 과정 중에서 얻은 결과들을 공유하기 위해서 'JPA를 사용한 조회 성능 최적화 기법'이란 주제를 연구과제로 선정했습니다.

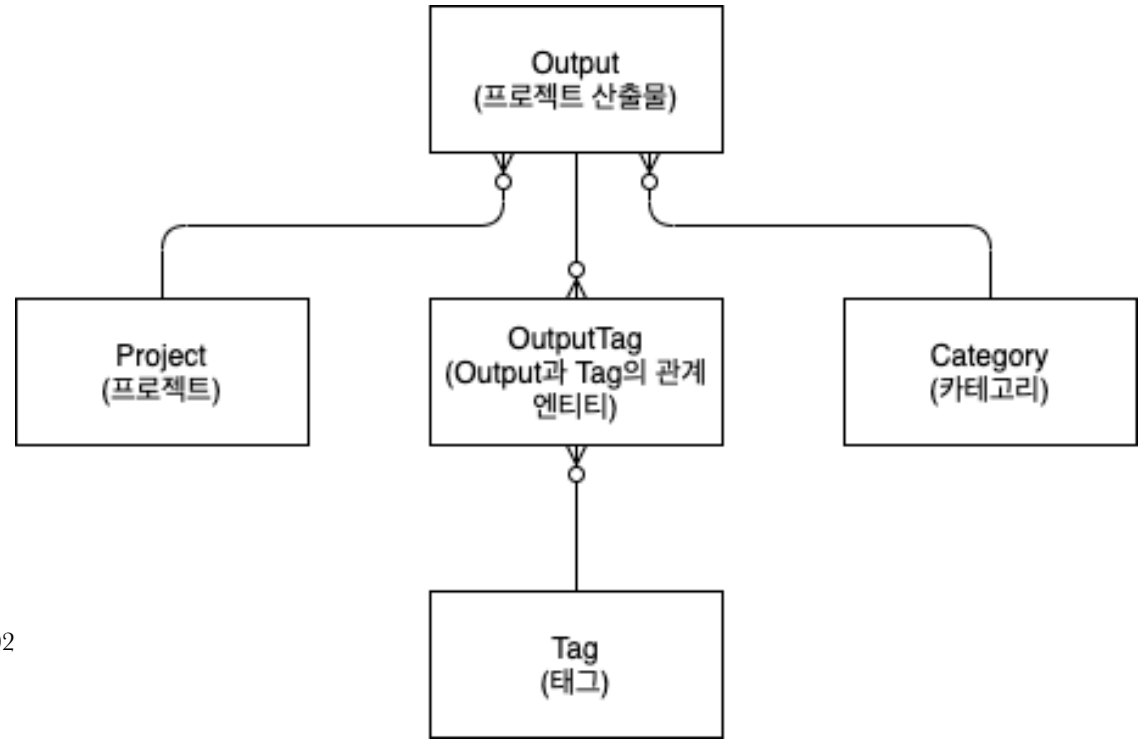
* 연구과제는 JPA의 기본적인 사용 방법을 아는 분들을 대상으로 작성했습니다.

2. 예제 소개

예제 코드 : https://github.com/wxxsoo/jpa-performance-optimization

아키텍트팀 프로젝트 산출물 조회 시스템을 기반으로 예제 코드를 작성했습니다.

- 엔티티 Class 구조



예제에는 총 5개의 엔티티가 있습니다. Output(프로젝트 산출물)은 예제에서 조회 대상이되는 엔티티입니다. 프로젝트를 수행하면서 발생한 산출물 자료(설계서, 계획서 등)을 의미합니다. 그 외 엔티티는 Output과 연관관계가 있는 엔티티입니다.

Project는 Output이 발생시킨 프로젝트 정보를 나타내는 엔티티입니다 .Output과 Project 간 관계는 다대일입니다.(Output[N], Project[1])

Category는 Output의 종류 정보를 나타내는 엔티티입니다.Output과 Category의 관계는 다대일 입니다.(Output[N], Category[1])

Tag는 Output의 태그 정보를 나타내는 엔티티입니다. Output과 Tag의 관계는 다대다 입니다.(Output[N], Tag[M]) 특이사항으로, Output과 Tag 사이의 다대다 관계를 맺어주는 OutputTag 엔티티가 있습니다.

- API 스펙

예제 코드는 다음과 같은 API 스펙을 가지고 프로젝트 산출물 목록을 조회합니다.

```
{
    "outputId": 10,
    "outputName": "Output A from project A",
    "project": {
        "projectId": 4,
        "projectName": "project A"
    },
    "category": {
        "categoryId": 1,
        "categoryName": "category A"
    },
    "tags": [
        {
            "tagId": 7,
            "tagName": "tag A"
        },
        {
            "tagId": 8,
            "tagName": "tag B"
        }
    ]
},
```

필드명	설명
outputId	Output 엔티티의 ID
outputName	프로젝트 산출물 이름
project	Output과 연관된 Project 엔티티의 DTO
projectId	Project 엔티티의 ID
projectName	프로젝트 이름
category	Output과 연관된 Category 엔티티의 DTO
categoryId	Category 엔티티의 Id
categoryName	카레고리 이름
tags	Output과 연관된 Tag 엔티티의 Dto 리스트
tagId	Tag 엔티티의 Id
tagName	태그명

이 연구과제는 위에 설명된 엔티티 구조와 API 스펙을 기반으로 작성했습니다.

3. 지연로딩 설정

JPA를 사용할 때 엔티티 간 연관 관계를 맺을 때 즉시 로딩이 아닌 반드시 **지연 로딩**을 사용해야합니다. 연구 과제 예제에서도 Output 엔티티는 다른 엔티티와 연관 관계를 맺을 때 지연 로딩을 사용합니다..

지연로딩과 즉시로딩의 차이는 다음과 같습니다. 지연로딩은 Output 엔티티를 조회 할 때 JPA 내부적으로 프록시 객체를 상속해서 가져옵니다. Output 엔티티에서 다른 연관관계 엔티티(Project, Category)를 가져오기 위해 getProject(), getCategory() 메소드가 발생할 때 마다 조회 쿼리를 실행합니다. 즉, 지연로딩을 사용하면 연관관계 엔티티들을 한번에 가져오지 않습니다. 대신 소스 코드에서 연관관계 엔티티 데이터에 접근할 시점에 (getProject(), getCategory()이 실행될 때) 조회 쿼리를 실행합니다.

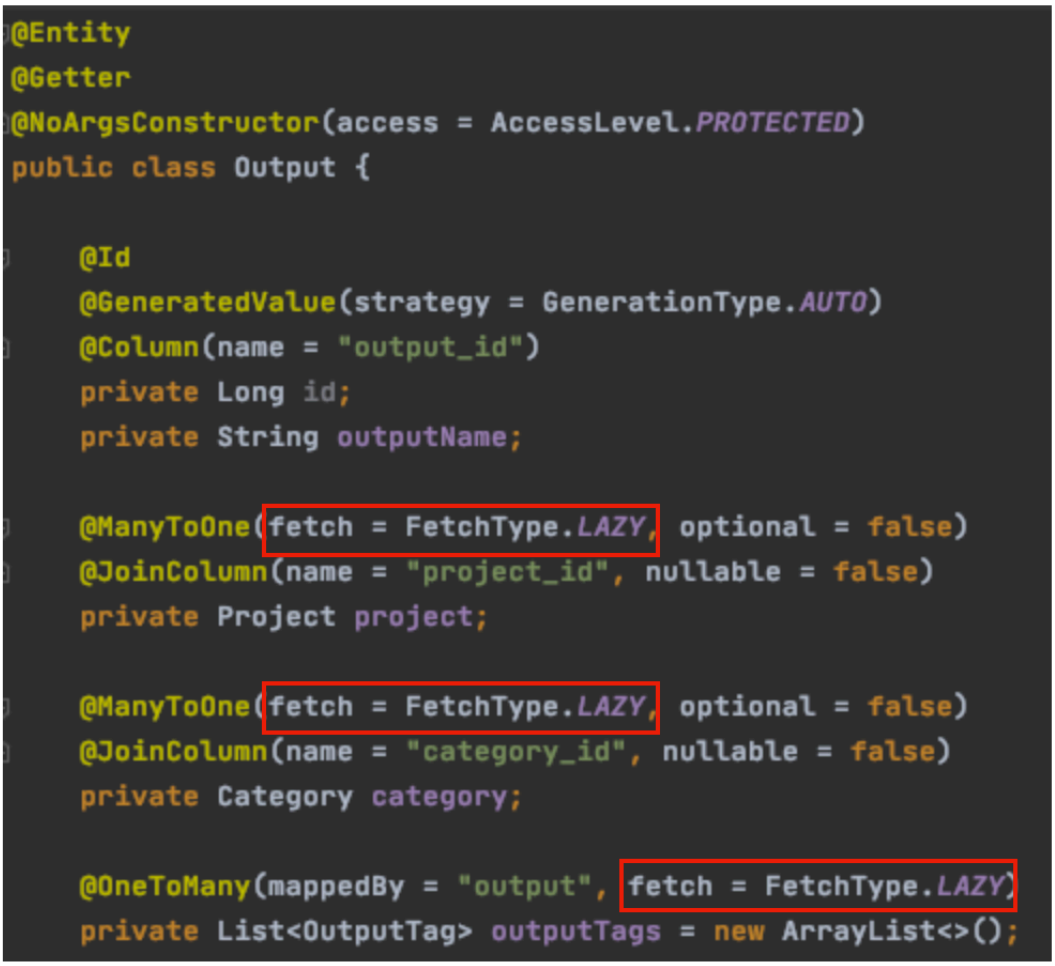
반면에 즉시로딩은 엔티티를 조회할 때 연관관계에 있는 모든 엔티티들을 조인하여 한번에 조회합니다. 모든 엔티티들을 한번에 가져오는 즉시로딩이 지연로딩이 더 이점이 있어보이지만 실무에서는 절대로 즉시 로딩을 사용하면 안됩니다.

즉시로딩을 사용하면 안되는 이유는 **예상하지 못한 SQL**이 발생하기 때문입니다. 위 예제는 연관관계가 3개 뿐이지만 실제 운영하는 서비스에서는 그보다 더 많은 연관관계가 있을 수 있습니다. 이러한 상황에서 엔티티 간 연관 관계 매핑이 즉시로딩으로 설정되어 있을경우 연관관계가 있는 모든 엔티티와 조인하여 엔티티를 조회합니다. 연관된 조회 테이블이 많을 수록 성능 이슈의 원인이 됩니다.

내용을 요약하면 다음과 같습니다.

1. 즉시로딩을 사용하면 불필요한 연관 엔티티들 까지 모두 조인하여 데이터를 가져오기 때문에 성능 이슈의 원인이 될 수 있다.
2. 엔티티 간 연관관계 매핑을 맺을 때 즉시로딩 대신 지연로딩을 설정한다.

따라서, 예제에서도 Output 엔티티는 연관관계가 있는 모든 엔티티는 지연로딩으로 설정합니다.



```java
@Entity
@Getter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Output {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "output_id")
    private Long id;
    private String outputName;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "project_id", nullable = false)
    private Project project;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "category_id", nullable = false)
    private Category category;

    @OneToMany(mappedBy = "output", fetch = FetchType.LAZY)
    private List<OutputTag> outputTags = new ArrayList<>();
```

Output은 다른 엔티티와 연관관계를 맺을 때 지연 로딩(fetch = fetchType.Lazy)을 사용하고 있다.

4. 엔티티를 DTO로 변환

JPA를 사용해 API를 개발할 때 가장 주의해야할 점은 엔티티를 프레젠테이션 계층(API)에 노출하면 안되는 것입니다. 이 말은 다음과 같이 컨트롤러에서 엔티티 형태로 바로 리턴하면 안됩니다.

그 이유는 다음과 같습니다.

첫째로, 엔티티에 프레젠테이션 계층을 위한 검증 로직이 추가가 필요할 경우 문제가 생깁니다.
예를들어, 'API a'에서 특정 필드 검증을 위해 @NotEmpty 어노테이션이 필요합니다. 동일한 데이터를 사용하는 'API b'에서 동일한 필드에 @NotEmpty 필드가 필요없습니다. 이러한 상황에서 동일한 엔티티에서 API 개발이 어렵습니다.

둘째로, 엔티티가 프레젠테이션 계층에 노출되면 엔티티가 API 스펙이 되어버립니다. 동일한 엔티티 필드에 대하여 'API a'는 name이라는 필드명을 사용해야합니다. 'API b'는 username이란 필드명을 사용해야 합니다. 이러한 상황에서 역시 동일한 엔티티에서 API 개발이 어렵습니다.

셋째로, 보여주고 싶지 않은 엔티티 필드를 노출해야합니다. 예를들어, 엔티티에 패스워드 필드와 같은 보여줄 수 없는 필드가 필요할 경우에도 엔티티를 프레젠테이션 계층에 바로 노출할 경우 문제가 생깁니다.

마지막으로, API 스펙이 변경될 때마다 빈번히 엔티티를 수정해야합니다. 엔티티는 어플리케이션의 중요한 비즈니스 로직을 담고있는 경우가 많습니다. 따라서, 중요한 엔티티의 빈번한 수정은 장애의 원인이 될 수 있습니다.

이러한 문제점들을 해결하기 위해서는 DTO를 사용해야합니다.

엔티티를 프레젠테이션 계층에 노출하지 않는다!

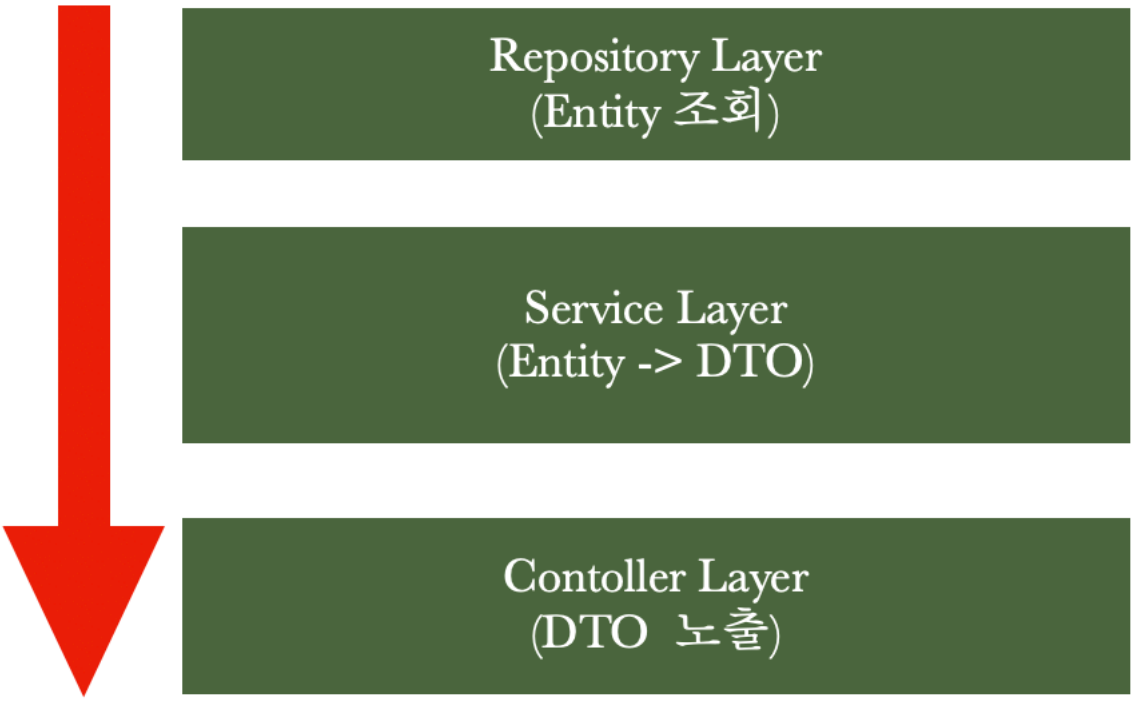
```java
@GetMapping("/api/v0/outputs")
public ResponseEntity<List<Output>> findOutputsV0() {
    return new ResponseEntity<>(outputRepository.findAll(), HttpStatus.OK);
}
```

대신 DTO를 사용!

```java
@GetMapping("/api/v1/outputs")
public ResponseEntity<List<OutputResponseDto>> findOutputsV1() {
    return new ResponseEntity<>(outputService.findOutputsV1(), HttpStatus.OK);
}
```

API 스펙에 맞춰 DTO 클래스를 생성 후 Service 계층에서 엔티티를 DTO로 변환합니다. 그 후 프레젠테이션 계층(컨트롤러 계층)에서 변환된 DTO를 클라이언트에게 노출해야하는 개발 패턴을 사용해야 합니다.

이러한 개발 패턴을 사용하면 위의 문제들을 모두해결 할 수 있으며, API 스펙에 맞춰 유동성 있는 개발을 가능하도록 합니다.

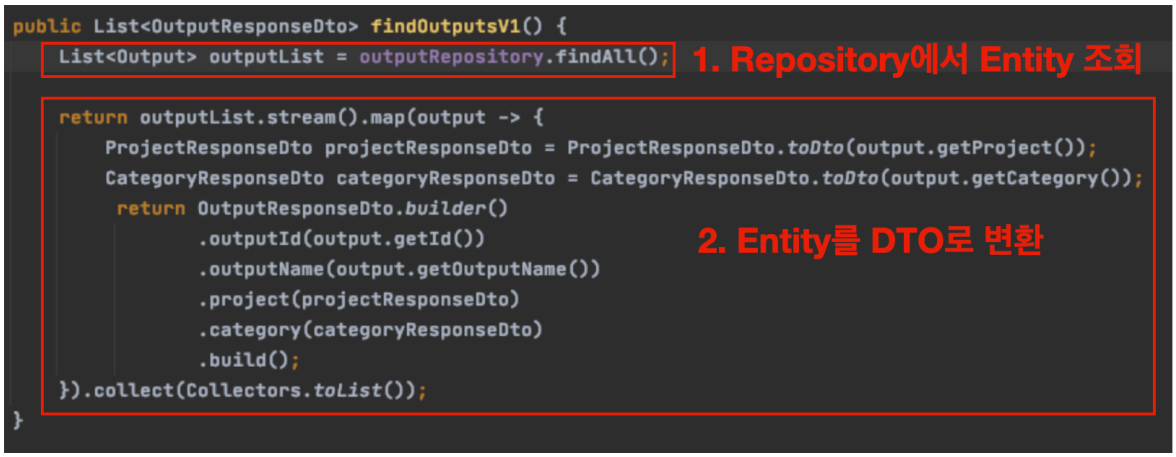


위 그림과 같이 JPA를 사용할 때, Repository 계층에서 엔티티를 조회합니다. 그 후 서비스 계층에서 엔티티를 DTO로 변환합니다. 마지막으로 컨트롤러에서 DTO를 반환합니다. 연구과제 예제에서 역시 위와 같은 개발 패턴을 적용하고 있습니다.

```java
@Data
public class OutputResponseDto {
    private Long outputId;
    private String outputName;
    private ProjectResponseDto project;
    private CategoryResponseDto category;
    private List<TagResponseDto> tags = new ArrayList<>();

    @Builder
    public OutputResponseDto(Long outputId, String outputName, ProjectR
        this.outputId = outputId;
        this.outputName = outputName;
        this.project = project;
        this.category = category;
        this.tags = tags;
    }
}
```

Output Dto 클래스

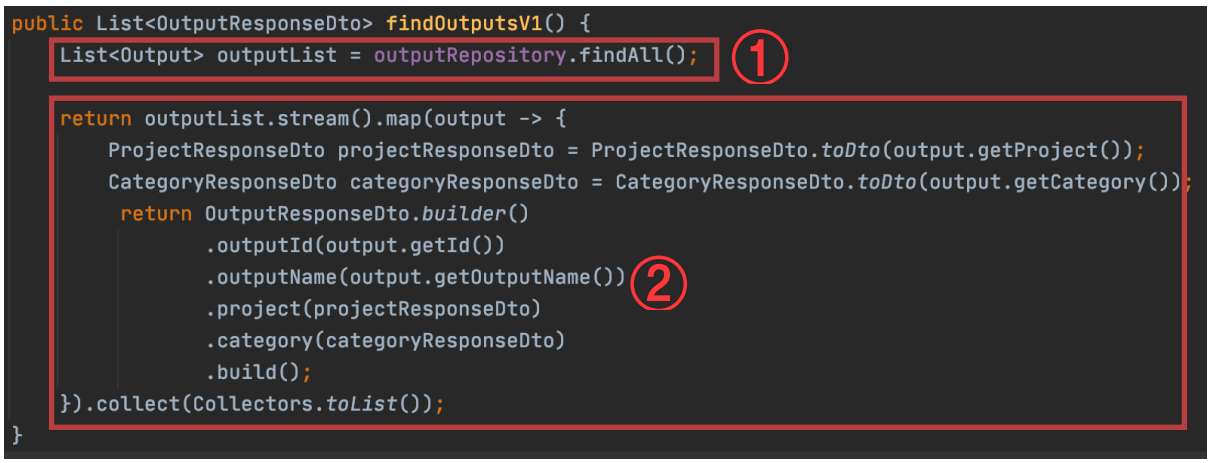



```
public List<OutputResponseDto> findOutputsV1() {
    List<Output> outputList = outputRepository.findAll();

    return outputList.stream().map(output -> {
        ProjectResponseDto projectResponseDto = ProjectResponseDto.toDto(output.getProject());
        CategoryResponseDto categoryResponseDto = CategoryResponseDto.toDto(output.getCategory());
         return OutputResponseDto.builder()
                .outputId(output.getId())
                .outputName(output.getOutputName())
                .project(projectResponseDto)
                .category(categoryResponseDto)
                .build();
    }).collect(Collectors.toList());
}
```

서비스 계층에서 Entity를 DTO로 변환

5. N+1 문제




```
public List<OutputResponseDto> findOutputsV1() {
    List<Output> outputList = outputRepository.findAll();

    return outputList.stream().map(output -> {
        ProjectResponseDto projectResponseDto = ProjectResponseDto.toDto(output.getProject());
        CategoryResponseDto categoryResponseDto = CategoryResponseDto.toDto(output.getCategory());
         return OutputResponseDto.builder()
                .outputId(output.getId())
                .outputName(output.getOutputName())
                .project(projectResponseDto)
                .category(categoryResponseDto)
                .build();
    }).collect(Collectors.toList());
}
```

위 메소드 findOutputV1()은 태그 정보를 제외하고 프로젝트 산출물(Output) 정보와 연관된 카테고리와 프로젝트 정보를 가져오는 서비스 계층의 메소드입니다. 메소드는 다음과 같은 로직을 가지고 있습니다.

1. Repository 계층에서 Output 리스트를 조회. Spring Data JPA가 제공하는 findAll() 메소드를 사용.
2. Output 리스트를 순회해서 Output DTO로 변환. 각 루프마다 Output과 연관된 Project와 Category 엔티티에 접근해 Project DTO, Category DTO로 변환해 OutputDTO를 생성.

```
@GetMapping("/api/v1/outputs")
public ResponseEntity<List<OutputResponseDto>> findOutputsV1() {
    return new ResponseEntity<>(outputService.findOutputsV1(), HttpStatus.OK);
}
```

컨트롤러 계층에서 서비스 계층의 findOutputV1을 사용하는 API를 만든 후 요청 결과를 확인합니다.

```
[
    {
        "outputId": 10,
        "outputName": "Output A from project A",
        "project": {
            "projectId": 4,
            "projectName": "project A"
        },
        "category": {
            "categoryId": 1,
            "categoryName": "category A"
        },
        "tags": null
    },
    {
        "outputId": 11,
        "outputName": "Output B from project B",
        "project": {
            "projectId": 5,
            "projectName": "project B"
        },
        "category": {
            "categoryId": 2,
            "categoryName": "category B"
        },
        "tags": null
    },
    {
        "outputId": 12,
        "outputName": "Output C from project C",
        "project": {
            "projectId": 6,
            "projectName": "project C"
        },
        "category": {
            "categoryId": 3,
            "categoryName": "category C"
        },
        "tags": null
    }
]
```

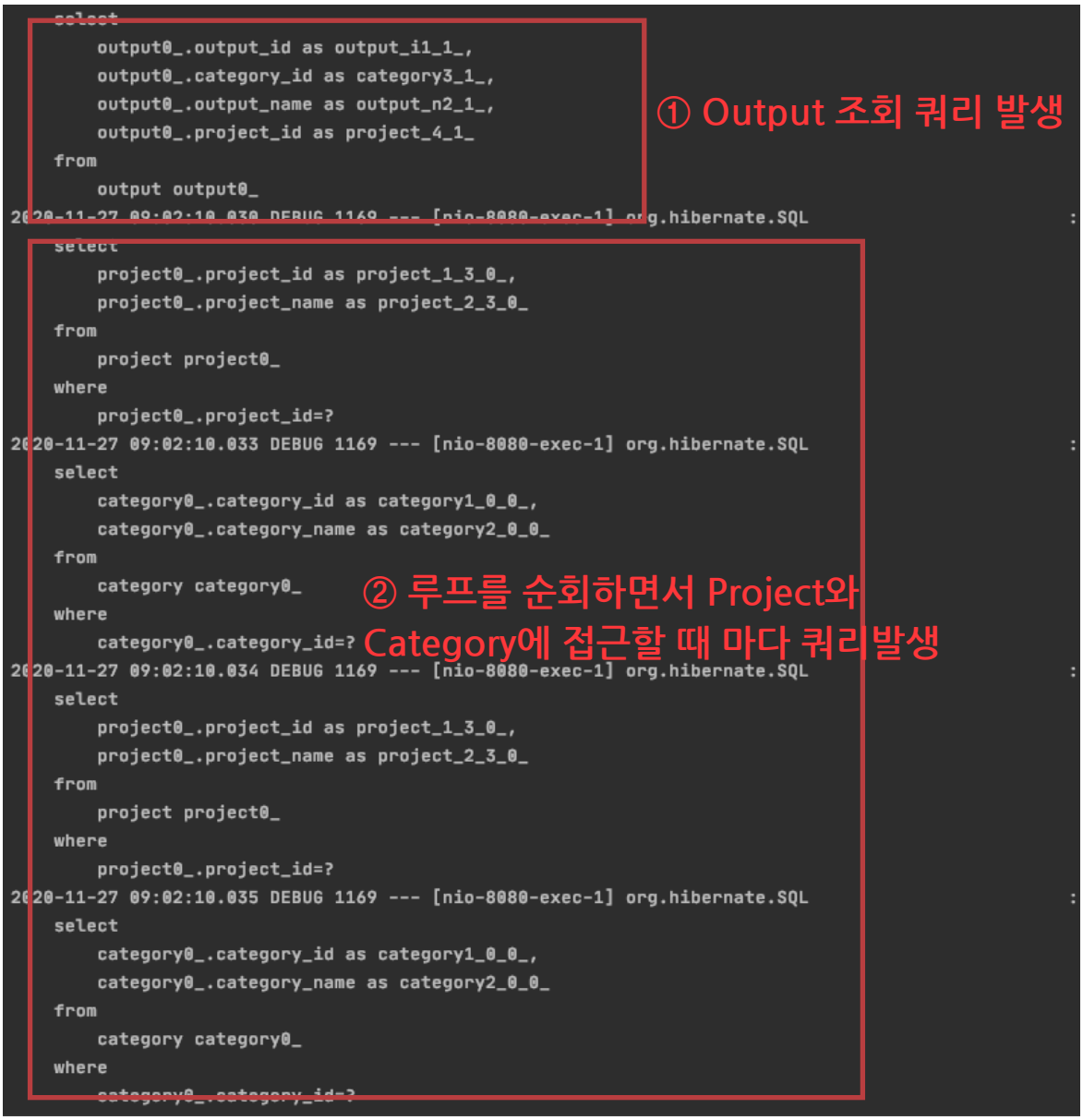

/api/v1/outputs의 결과

의도한대로 산출물 목록이 조회되었습니다. 코드를 작성하지 않은 Tag 정보를 제외하고 Output 엔티티, Project 엔티티, Category 엔티티 정보가 DTO로 변환되어 보여지고 있습니다.

예제코드에서는 3건의 Output 엔티티가 존재하고 각각의 Output 엔티티마다 서로 다른 Project와 Category를 매핑한 데이터를 초기화 시켜 놓았습니다.

그 결과, Output 3건과 연관된 Project, Category 정보가 보여지고 있습니다.

하지만 이 API는 치명적인 결함을 가지고있습니다. API를 호출했을 때 발생한 쿼리를 살펴보면 그 이유를 알 수 있습니다.



위 사진은 API를 호출했을 때 발생한 쿼리의 일부를 캡쳐했습니다. 총 발생한 쿼리는 7개입니다. 첫째로 Output 목록을 조회하는 select 쿼리가 1개를 실행합니다. 그 후 Output 목록을 순회하면서 Project와 Category에 접근하는 시점(getProject(), getCategory() 메소드 호출 시점)마다 Project와 Category를 조회하는 쿼리 6개가 실행되었습니다.

API 호출 한번에 7개의 쿼리가 발생했습니다. 이는 API 호출 한번에 어플리케이션과 데이터베이스가 7번의 통신이 이루어진 것을 의미합니다.

예제는 데이터의 수가 적어서 성능에 크게 문제가 없겠지만 만약 Output의 수가 10억개라고 가정하면 심각한 성능 이슈를 예상할 수 있습니다. 최악의 경우 Output 조회 쿼리 1개 + 루프를 돌면서 발생한 Project 조회 쿼리 10억개 + 루프를 돌면서 발생한 Category 조회 쿼리 10억개, API 호출 한 번에 20억개의 쿼리가 발생합니다.

* 참고 - 지연 로딩은 영속성 컨텍스트에 있으면 영속성 엔티티에 있는 엔티티를 사용하고 없으면 쿼리를 실행한다. 따라서 같은 영속성 컨텍스트에서 이미 로딩한 Project와 Category에 추가로 접근할 때 쿼리를 실행하지 않는다.

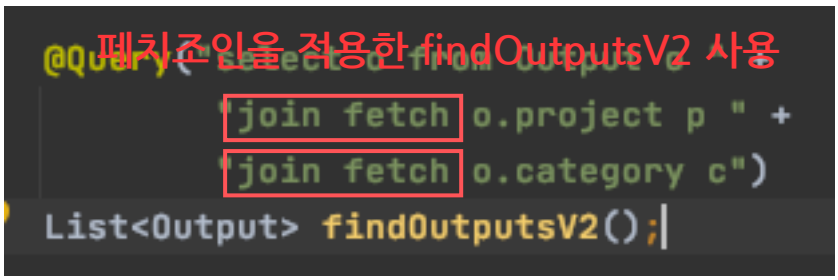
처음부터 Output 목록을 가져올 때 연관된 Project와 Category를 한번에 데이터베이스에서 가져오면 좋겠지만 앞서 언급한대로 불필요한 엔티티 조인을 피하기 위해 모든 엔티티를 지연로딩으로 설정해서 이는 불가능합니다.

JPA는 이를 N+1문제라고 합니다. 예제에서 처럼 Output 조회 쿼리 1개 + 연관된 엔티티를 조회하면서 발생하는 쿼리 N개가 발생하기 때문에 N+1 문제라 불립니다.

6. 페치 조인을 사용한 N+1 문제 해결

N+1 문제를 해결하기 위해서 페치 조인을 사용해야합니다. 페치 조인은 JPQL에서 성능 최적화를 위해 제공하는 기능입니다. 연관된 엔티티를 한번에 같이 조회하는 기능인데 join fetch 명령어로 사용할 수 있습니다.

예제에서 N+1 문제를 해결하기 위해서는 Output 목록을 가져올 때 Spring Data JPA에서 제공하는 findAll() 메소드 대신 페치 조인을 사용하는 메소드를 직접 만들어 대신 사용하면 N+1문제를 해결할 수 있습니다.



페치 조인을 사용한 Output 조회 메소드 개발

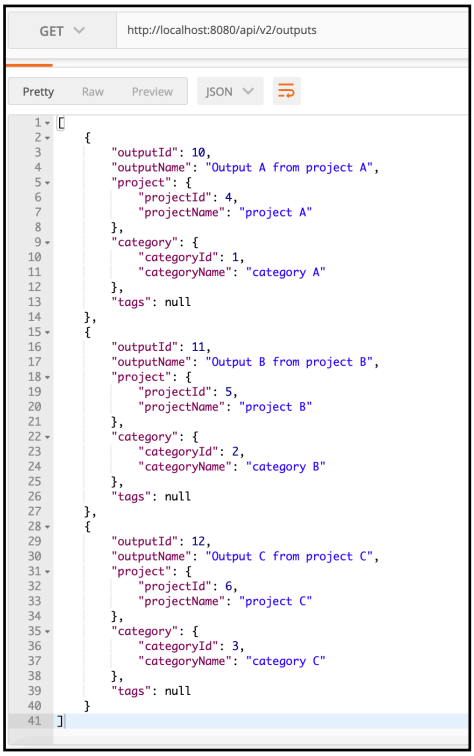
예제에서는 OutputRepository에 페치 조인을 사용하는 findOutputsV2() 메소드를 작성했습니다. findOutputV2() 메소드는 JPQL을 직접 작성할 수 있도록 Spring Data JPA에서 제공하는 @Query어노테이션에서 페치 조인이 적용되는 쿼리를 작성 했습니다. Output과 Project, Category이 페치조인하는 것을 확인할 수 있습니다.

OutputRepository의 findOutputsV2를 사용하는 서비스 계층의 메소드 findOutputsV2를 만듭니다.

```java
public List<OutputResponseDto> findOutputsV2() {
    List<Output> outputList = outputRepository.findOutputsV2();

    return outputList.stream().map(output -> {
        ProjectResponseDto projectResponseDto = ProjectResponseDto.toDto(output.getProject());
        CategoryResponseDto categoryResponseDto = CategoryResponseDto.toDto(output.getCategory());
        return OutputResponseDto.builder()
                .outputId(output.getId())
                .outputName(output.getOutputName())
                .project(projectResponseDto)
                .category(categoryResponseDto)
                .build();
    }).collect(Collectors.toList());
}
```

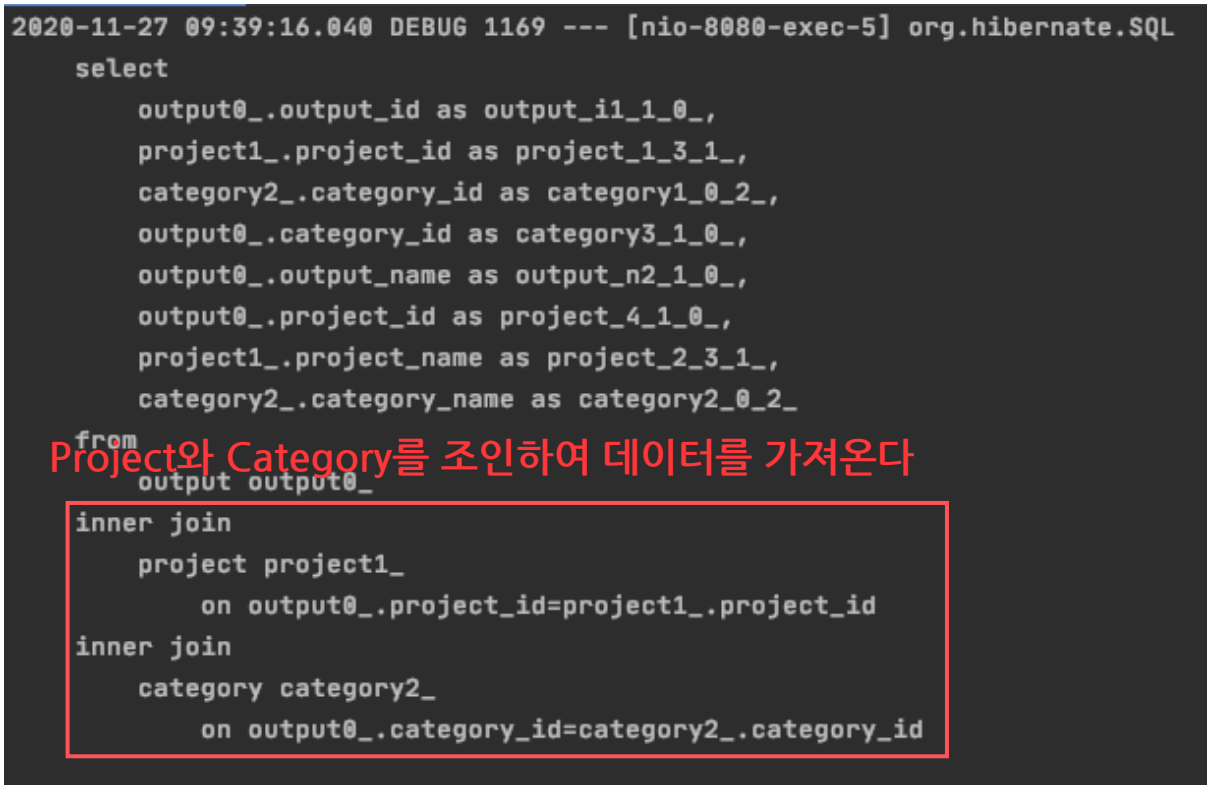
컨트롤러에 동일하게 서비스 계층의 findOutputsV2를 사용하는 API를 만든 후 API를 호출하고 동일한 경과를 확인합니다.



동일한 결과

페치 조인을 적용한 findOutputsV2() 메소드를 사용했을 때 원하는대로 Output과 Project, Category 정보가 보여지는 것을 확인할 수 있습니다.

이제 가장 중요한 발생한 쿼리 로그를 확인합니다.



단 한개의 쿼리가 발생한 것을 확인할 수 있습니다. 또한 이전과 다르게 Output을 조회하는 쿼리에서 Project와 Category와 inner join이 된 것을 확인할 수 있습니다.

지금까지의 내용을 요약하면 다음과 같습니다.

1. 지연로딩으로 연관관계가 매핑된 엔티티에 접근할 때마다 쿼리가 발생한다.
2. 이는 N+1문제라 불리며 성능 이슈에 원인이 된다.
3. N+1문제를 해결하기 위해서 레포지토리 계층에서 연관된 엔티티를 페치조인하는 메소드를 개발해 사용한다.

7. 컬렉션 조회 최적화 1

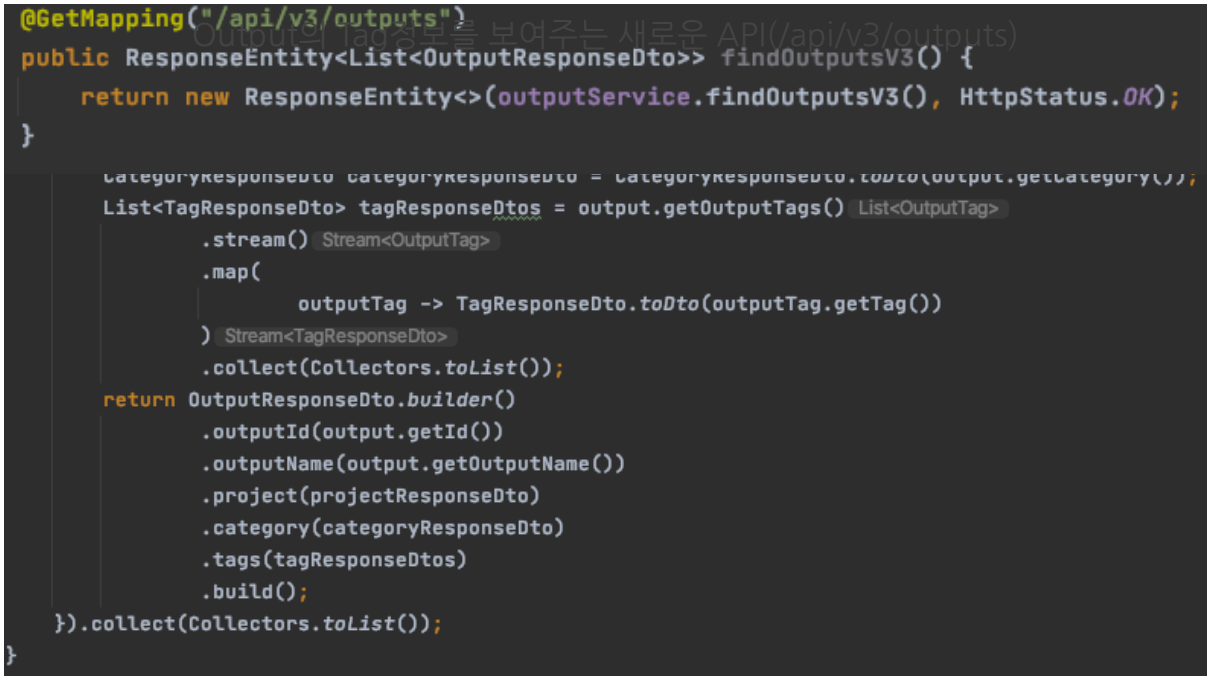
이제 Output 목록을 조회할 때 Tag 목록 정보도 같이 보여지도록 변경합니다.

Output과 Tag의 연관관계는 다대다입니다. 또한, 중간에 Output과 Tag사이에 관계형 엔티티인 OutputTag가 존재합니다. 즉, Output에서 Tag에 접근하려면 Output -> OutputTags -> Tag 순서로 접근해야합니다.

해당 로직을 수행하는 findOutputsV3()를 서비스 계층에 추가합니다.

이전과 동일한 소스코드에 Tag에 접근해 DTO로 변환하는 로직을 추가했습니다.

서비스 계층의 findOutputsV3() 메소드를 사용하는 API를 컨트롤러에 추가합니다.


@GetMapping("/api/v3/outputs")
Output의 Tag정보를 보여주는 새로운 API(/api/v3/outputs)
public ResponseEntity<List<OutputResponseDto>> findOutputsV3() {
return new ResponseEntity<>(outputService.findOutputsV3(), HttpStatus.OK);
}
List<TagResponseDto> tagResponseDtos = output.getOutputTags() List<OutputTag>
.stream() Stream<OutputTag>
.map(
outputTag -> TagResponseDto.toDto(outputTag.getTag())
) Stream<TagResponseDto>
.collect(Collectors.toList());
return OutputResponseDto.builder()
.outputId(output.getId())
.outputName(output.getOutputName())
.project(projectResponseDto)
.category(categoryResponseDto)
.tags(tagResponseDtos)
.build();
}).collect(Collectors.toList());
}

```
[
    {
        "outputId": 10,
        "outputName": "Output A from project A",
        "project": {
            "projectId": 4,
            "projectName": "project A"
        },
        "category": {
            "categoryId": 1,
            "categoryName": "category A"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 8,
                "tagName": "tag B"
            }
        ]
    },
    {
        "outputId": 11,
        "outputName": "Output B from project B",
        "project": {
            "projectId": 5,
            "projectName": "project B"
        },
        "category": {
            "categoryId": 2,
            "categoryName": "category B"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 9,
                "tagName": "tag C"
```


/api/v3/outputs의 결과

API를 호출하면 Output 정보와 Project, Category 그리고 Tag정보가 보여지는 것을 확인할 수 있습니다.

하지만 Output과 연관된 엔티티인 OutputTags와 OutputTag의 Tag와 페치조인이 되어있지 않기 때문에 N+1문제가 발생합니다.

```
    select
        output0_.output_id as output_i1_1_0_,
        project1_.project_id as project_1_3_1_,
        category2_.category_id as category1_0_2_,
        output0_.category_id as category3_1_0_,
        output0_.output_name as output_n2_1_0_,
        output0_.project_id as project_4_1_0_,
        project1_.project_name as project_2_3_1_,
        category2_.category_name as category2_0_2_
    from
        output output0_
    inner join
        project project1_
            on output0_.project_id=project1_.project_id
    inner join
        category category2_
            on output0_.category_id=category2_.category_id
2020-11-27 10:02:46.894 DEBUG 1263 --- [nio-8080-exec-1] org.hibernate.SQL
    select
        outputtags0_.output_id as output_i2_2_0_,
        outputtags0_.output_tag_id as output_t1_2_0_,
        outputtags0_.output_tag_id as output_t1_2_1_,
        outputtags0_.output_id as output_i2_2_1_,
        outputtags0_.tag_id as tag_id3_2_1_
    from
        output_tag outputtags0_
    where
        outputtags0_.output_id=?
2020-11-27 10:02:46.903 DEBUG 1263 --- [nio-8080-exec-1] org.hibernate.SQL
    select
        tag0_.tag_id as tag_id1_4_0_,
        tag0_.tag_name as tag_name2_4_0_
    from
        tag tag0_
    where
        tag0_.tag_id=?
2020-11-27 10:02:46.904 DEBUG 1263 --- [nio-8080-exec-1] org.hibernate.SQL
    select
        tag0_.tag_id as tag_id1_4_0_,
        tag0_.tag_name as tag_name2_4_0_
```

위 화면을 보면 Output의 OutputTags에 접근하고 OutputTags에서 루프를 순회하여 Tag에 접근할 때 마다 조회 쿼리가 발생해 N+1문제가 일어나는 것을 확인할 수 있습니다.

N+1문제를 해결하기 위해 레포지토리 계층에 Output과 OutputTag, Tag를 페치조인 하는 findOutputV3()를 추가합니다.

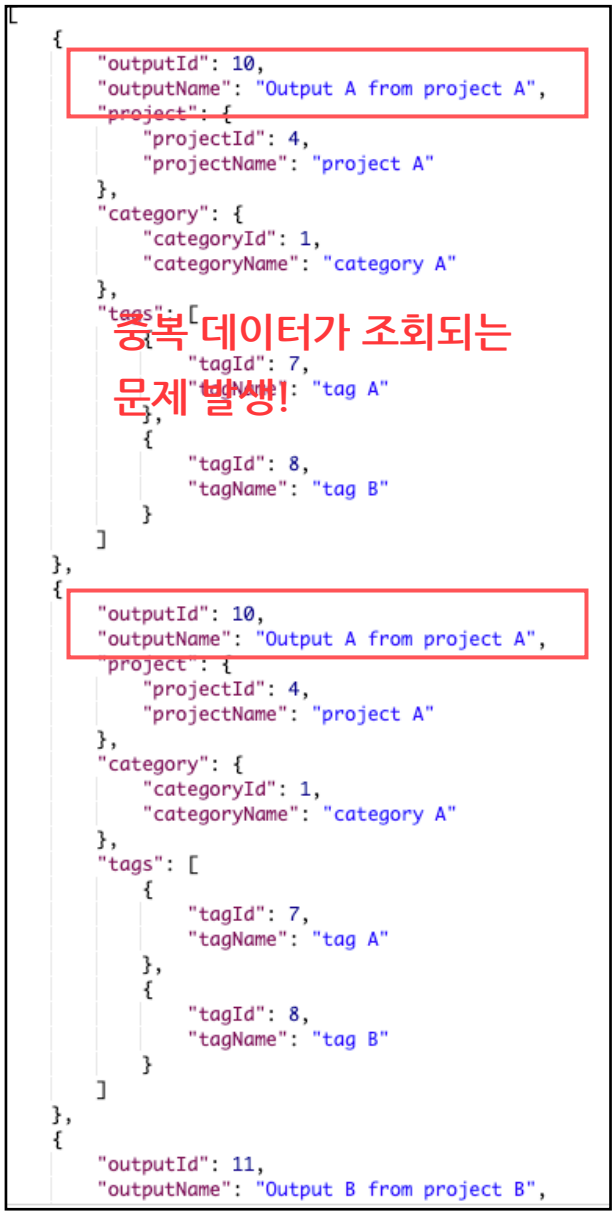
```
@Query("select o from Output o " +
        "join fetch o.project p " +
        "join fetch o.category c " +
        "join fetch o.outputTags ot " +
        "join fetch ot.tag")
List<Output> findOutputsV3();
```

OutputTag와 Tag를 페치조인

서비스 계층에서 OutputTag와 Tag를 페치조인 하지 않는 findOutputsV2() 대신 findOutputsV3()를 사용하도록 변경합니다.

```
public List<OutputResponseDto> findOutputsV3() {
//      List<Output> outputList = outputRepository.findOutputsV2();
    List<Output> outputList = outputRepository.findOutputsV3();
    return outputList.stream().map(output -> {
        ProjectResponseDto projectResponseDto = ProjectResponseDto.toDto(output.getProject());
        CategoryResponseDto categoryResponseDto = CategoryResponseDto.toDto(output.getCategory());
        List<TagResponseDto> tagResponseDtos = output.getOutputTags() List<OutputTag>
                .stream() Stream<OutputTag>
                .map(
                        outputTag -> TagResponseDto.toDto(outputTag.getTag())
                ) Stream<TagResponseDto>
                .collect(Collectors.toList());
        return OutputResponseDto.builder()
                .outputId(output.getId())
                .outputName(output.getOutputName())
                .project(projectResponseDto)
                .category(categoryResponseDto)
                .tags(tagResponseDtos)
                .build();
    }).collect(Collectors.toList());
}
```

그리고 API를 호출해 결과를 확인합니다.




```
[
    {
        "outputId": 10,
        "outputName": "Output A from project A",
        "project": {
            "projectId": 4,
            "projectName": "project A"
        },
        "category": {
            "categoryId": 1,
            "categoryName": "category A"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 8,
                "tagName": "tag B"
            }
        ]
    },
    {
        "outputId": 10,
        "outputName": "Output A from project A",
        "project": {
            "projectId": 4,
            "projectName": "project A"
        },
        "category": {
            "categoryId": 1,
            "categoryName": "category A"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 8,
                "tagName": "tag B"
            }
        ]
    },
    {
        "outputId": 11,
        "outputName": "Output B from project B",
```


예상하는 Output의 수는 3개입니다 하지만 총 6개의 Output 결과가 보여집니다.
또한, 각 Output마다 중복된 데이터를 보여주고 있습니다.

```
select
    output0_.output_id as output_i1_1_0_,
    project1_.project_id as project_1_3_1_,
    category2_.category_id as category1_0_2_,
    outputtags3_.output_tag_id as output_t1_2_3_,
    tag4_.tag_id as tag_id1_4_4_,
    output0_.category_id as category3_1_0_,
    output0_.output_name as output_n2_1_0_,
    output0_.project_id as project_4_1_0_,
    project1_.project_name as project_2_3_1_,
    category2_.category_name as category2_0_2_,
    outputtags3_.output_id as output_i2_2_3_,
    outputtags3_.tag_id as tag_id3_2_3_,
    outputtags3_.output_id as output_i2_2_0__,
    outputtags3_.output_tag_id as output_t1_2_0__,
    tag4_.tag_name as tag_name2_4_4_
from
    output output0_
inner join
    project project1_
        on output0_.project_id=project1_.project_id
inner join
    category category2_
        on output0_.category_id=category2_.category_id
inner join
    output_tag outputtags3_
        on output0_.output_id=outputtags3_.output_id
inner join
    tag tag4_
        on outputtags3_.tag_id=tag4_.tag_id
```

API를 호출 시 발생한 쿼리입니다. 페치조인을 적용해 1개의 쿼리만 실행시키는 것을 확인할 수 있습니다. 이 쿼리를 데이터베이스에서 실행해보면 다음과 같은 결과를 얻습니다.

on outputtags3_.tag_id=tag4_.tag_id;

OUTPUT_I1_1_0_	PROJECT_1_3_1_	CATEGORY1_0_2_	OUTPUT_T1_2_3_	TAG_ID1_4_4_	CATEGORY3_1_0_	OUTPUT_N2_1_0
10	4	1	13	7	1	Output A from project A
10	4	1	14	8	1	Output A from project A
11	5	2	15	7	2	Output B from project B
11	5	2	16	9	2	Output B from project B
12	6	3	17	8	3	Output C from project C
12	6	3	18	9	3	Output C from project C

(6 rows, 5 ms)

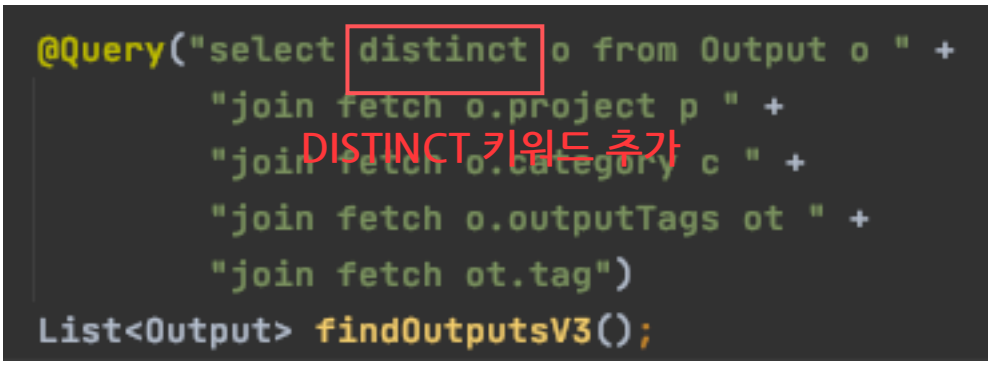
데이터 베이스에서 결과 역시 중복된 데이터가 존재하는 것을 확인할 수 있습니다.

JPA에서 일대다 관계와 같은 컬렉션 조회가 이루어 질때 조인시 연관된 데이터 수 만큼 row수가 증가합니다.

예를들어, 예제에서는 Output과 일대다 연관관계를 가지고 있는 List<OutputTag> outputTags가 존재합니다. 그리고 각각의 Output은 2개의 OutputTag를 가지고 있습니다. JPA는 Output의 하위 엔티티인 OutputTag가 2개이므로 Output을 2번 조회해서 각각의 Output Tag 데이터를 가져옵니다.

이렇게 데이터가 중복되는 문제를 해결하기 위해서 JPQL의 **DISTINCT**를 사용합니다. 사용방법은 sql의 DISTINCT와 유사합니다.

레포지토리 계층의 findOutputsV3()에서 작성된 jpql에 DISTINCT 키워드를 추가합니다.

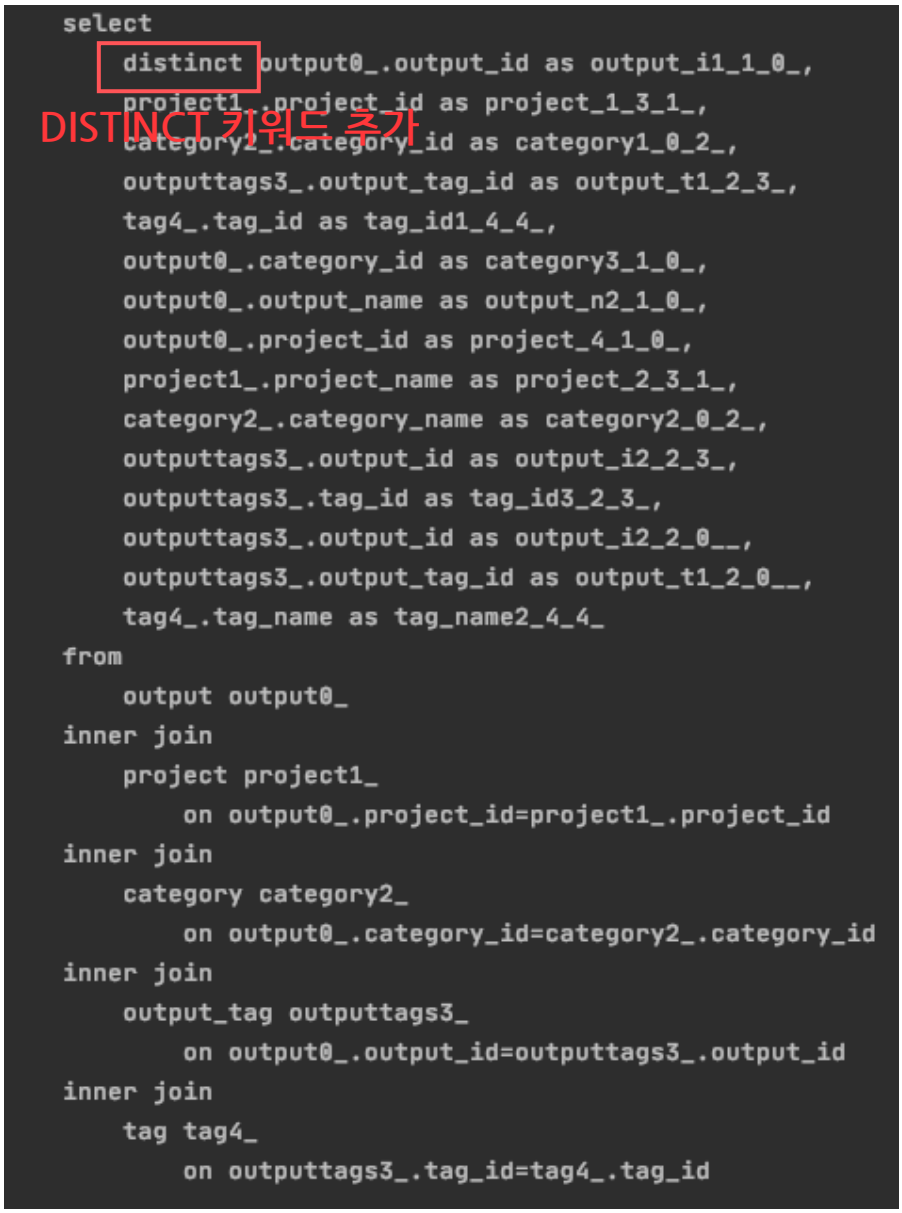


레포지토리 메소드를 수정 후 API를 호출하면 정상적인 결과를 확인할 수 있습니다.

```
[
    {
        "outputId": 10,
        "outputName": "Output A from project A",
        "project": {
            "projectId": 4,
            "projectName": "project A"
        },
        "category": {
            "categoryId": 1,
            "categoryName": "category A"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 8,
                "tagName": "tag B"
            }
        ]
    },
    {
        "outputId": 11,
        "outputName": "Output B from project B",
        "project": {
            "projectId": 5,
            "projectName": "project B"
        },
        "category": {
            "categoryId": 2,
            "categoryName": "category B"
        },
        "tags": [
            {
                "tagId": 7,
                "tagName": "tag A"
            },
            {
                "tagId": 9,
                "tagName": "tag C"
            }
        ]
    },
    {
        "outputId": 12,
        "outputName": "Output C from project C",
```


Distinct를 사용하여 중복이 제거된 결과

로그를 확인하면 쿼리의 distinct 키워드가 추가된 것을 확인할 수 있습니다.




```
select
    distinct output0_.output_id as output_i1_1_0_,
    project1_.project_id as project_1_3_1_,
    category2_.category_id as category1_0_2_,
    outputtags3_.output_tag_id as output_t1_2_3_,
    tag4_.tag_id as tag_id1_4_4_,
    output0_.category_id as category3_1_0_,
    output0_.output_name as output_n2_1_0_,
    output0_.project_id as project_4_1_0_,
    project1_.project_name as project_2_3_1_,
    category2_.category_name as category2_0_2_,
    outputtags3_.output_id as output_i2_2_3_,
    outputtags3_.tag_id as tag_id3_2_3_,
    outputtags3_.output_id as output_i2_2_0__,
    outputtags3_.output_tag_id as output_t1_2_0__,
    tag4_.tag_name as tag_name2_4_4_
from
    output output0_
inner join
    project project1_
        on output0_.project_id=project1_.project_id
inner join
    category category2_
        on output0_.category_id=category2_.category_id
inner join
    output_tag outputtags3_
        on output0_.output_id=outputtags3_.output_id
inner join
    tag tag4_
        on outputtags3_.tag_id=tag4_.tag_id
```

Distinct 키워드가 추가

8. 컬렉션 조회 최적화 2 - 페이징 처리 시 페치 조인의 한계

산출물 조회 시 페이징 기능을 추가하겠습니다. 레포지토리 계층에 다음과 같은 findOutputsV4() 메소드를 추가합니다.

```
@Query("select distinct o from Output o " +
        "join fetch o.project p " +
        "join fetch o.category c " +
        "join fetch o.outputTags ot " +
        "join fetch ot.tag")
List<Output> findOutputsV4(Pageable pageable);
```

레포지토리 계층에 findOutputsV4를 추가

소스코드는 기존의 findOutputsV3()와 유사합니다. 다른 점은 파라미터에 Pageable이 추가된 것을 확인할 수 있습니다. JPA는 Pageable 파라미터를 인식하여 자동으로 페이징 처리를 제공합니다. 편리한 페이징 처리는 JPA의 장점 중 하나입니다.

레포지토리 계층의 findOutputsV4를 사용하는 서비스 계층의 메소드를 추가합니다.

```
public List<OutputResponseDto> findOutputsV4(Pageable pageable) {
    List<Output> outputList = outputRepository.findOutputsV4(pageable);

    return outputList.stream().map(output -> {
        ProjectResponseDto projectResponseDto = ProjectResponseDto.toDto(output.getProject());
        CategoryResponseDto categoryResponseDto = CategoryResponseDto.toDto(output.getCategory());
        List<TagResponseDto> tagResponseDtos = output.getOutputTags() List<OutputTag>
                .stream() Stream<OutputTag>
                .map(
                        outputTag -> TagResponseDto.toDto(outputTag.getTag())
                ) Stream<TagResponseDto>
                .collect(Collectors.toList());
        return OutputResponseDto.builder()
                .outputId(output.getId())
                .outputName(output.getOutputName())
                .project(projectResponseDto)
                .category(categoryResponseDto)
                .tags(tagResponseDtos)
                .build();
    }).collect(Collectors.toList());
}
```

findOutputsV4 사용하여 페이징 처리

서베스 계층에 findOutputsV4를 추가

서비스 계층의 findOutputsV4를 사용하는 컨트롤러 계층에 페이징 처리를 제공하는 API를 추가합니다.

```java
@GetMapping("/api/v4/outputs")
public ResponseEntity<List<OutputResponseDto>> findOutputsV4(Pageable pageable) {
    return new ResponseEntity<>(outputService.findOutputsV4(pageable), HttpStatus.OK);
}
```

API(/api/v4/outputs/) 추가

API를 호출합니다. 페이징 처리를 위해 Request 파라미터를 추가합니다. page를 1, size를 2로 설정해 /api/v4/outputs?page=1&size=2를 호출합니다.
그 결과 정상적으로 페이징 처리된 것을 확인할 수 있습니다.


```json
[
    {
        "outputId": 12,
        "outputName": "Output C from project C",
        "project": {
            "projectId": 6,
            "projectName": "project C"
        },
        "category": {
            "categoryId": 3,
            "categoryName": "category C"
        },
        "tags": [
            {
                "tagId": 8,
                "tagName": "tag B"
            },
            {
                "tagId": 9,
                "tagName": "tag C"
            }
        ]
    }
]
```


/api/v4/outputs?page=1&size=2의 결과

페이징 처리가 적용된 정상적인 API인 것 같지만 이 API 역시 성능 이슈가 존재합니다. API를 요청할 때 쿼리를 확인하면 다음과 같습니다.

```
select
    distinct output0_.output_id as output_i1_1_0_,
    project1_.project_id as project_1_3_1_,
    category2_.category_id as category1_0_2_,
    outputtags3_.output_tag_id as output_t1_2_3_,
    tag4_.tag_id as tag_id1_4_4_,
    output0_.category_id as category3_1_0_,
    output0_.output_name as output_n2_1_0_,
    output0_.project_id as project_4_1_0_,
    project1_.project_name as project_2_3_1_,
    category2_.category_name as category2_0_2_,
    outputtags3_.output_id as output_i2_2_3_,
    outputtags3_.tag_id as tag_id3_2_3_,
    outputtags3_.output_id as output_i2_2_0__,
    outputtags3_.output_tag_id as output_t1_2_0__,
    tag4_.tag_name as tag_name2_4_4_
from
    output output0_
inner join
    project project1_
        on output0_.project_id=project1_.project_id
inner join
    category category2_
        on output0_.category_id=category2_.category_id
inner join
    output_tag outputtags3_
        on output0_.output_id=outputtags3_.output_id
inner join
    tag tag4_
        on outputtags3_.tag_id=tag4_.tag_id
```

쿼리를 확인하면 페이징 처리를 하는 내용이 없는 것을 확인할 수 있습니다.
로그에 나오는 쿼리를 데이터베이스에서 직접 실행한 결과 역시 페이징 처리된 결과가 나오지 않고 모든 데이터가 조회되고 있습니다.

on outputtags3_.tag_id=tag4_.tag_id;

OUTPUT_I1_1_0_	PROJECT_1_3_1_	CATEGORY1_0_2_	OUTPUT_T1_2_3_	TAG_ID1_4_4_	CATEGORY3_1_0_	OUTPUT_N2_1_0
10	4	1	13	7	1	Output A from project A
10	4	1	14	8	1	Output A from project A
11	5	2	15	7	2	Output B from project B
11	5	2	16	9	2	Output B from project B
12	6	3	17	8	3	Output C from project C
12	6	3	18	9	3	Output C from project C

(6 rows, 5 ms)

데이터베이스 결과값에서 페이징 처리가 되지 않고 심지어 중복된 데이터까지 존재하는 것을 확인할 수 있습니다.

```
firstResult/maxResults specified with collection fetch; applying in memory!
```

또한, 다음과 같은 로그도 확인할 수 있습니다.
로그를 보면 알수있듯이 JPA에서 페치 조인된 컬렉션을 페이징 처리를 **데이터베이스가 아닌 메모리에서 페이징 처리**가 이루어지는 것을 확인할 수 있습니다. 중복된 수많은 데이터를 메모리 단계에서 페이징 처리를 하면 어플리케이션에 큰 부하를 줄 수 있어 성능 이슈의 원인이 될 수 있습니다.

페치 조인한 컬렉션을 페이징 처리를 할 경우 메모리에서 페이징 처리를 해 성능 이슈에 원인이 될 수 있는 단점이 있습니다. 또한 그외에도 컬렉션 페치 조인은 1개만 사용할 수 있습니다. 컬렉션 둘 이상에 페치 조인을 사용하면 데이터가 부정합하게 조회될 수 있습니다.

9. 컬렉션 조회 최적화 3 - 페이징 처리 최적화

컬렉션 조회가 이루어질 경우 페이징 처리가 불가능한 한계가 있었습니다.
내용을 요약하면 다음과 같습니다.

1. 컬렉션을 페치 조인하면 일대다 조인이 발생하므로 중복된 데이터가 예측할 수 없이 증가한다.
2. 일대다에서 일(1, Output)을 기준으로 페이징해야하는데, 데이터베이스에서 결과는 다(N, OutputTag) 기준으로 row가 생성된다.
3. Output 기준으로 페이징 하고 싶은데, 다(N)인 OutputTag를 조인하면 OutputTag가 기준이 되어버린다.
4. 결국 JPA는 모든 DB 데이터를 어플리케이션에서 가져와 페이징을 시도해 어플리케이션에 부하를 일으킨다.

이렇게 컬렉션 조회를 페이징 처리할 때 발생하는 문제점들은 해결해 조회 성능을 최적하하는 방법은 의외로 간단합니다.

컬렉션 조회를 페이징 처리할 때 성능 최적화 방법을 간단하게 요약하면 다음과 같습니다.

- ToOne(OneToOne, ManyToOne) 관계를 모두 페치 조인한다.
- 컬렉션은 지연 로딩으로 조회한다.
- hibernate.default_batch_size 옵션을 어플리케이션 설정에 추가한다.

- ToOne(OneToOne, ManyToOne) 관계를 모두 페치 조인한다.

예제의 Output과 ManyToOne으로 매핑되어있는 Project와 Category 엔티티에 해당됩니다.
ToOne 관계는 row수를 증가시키지 않으므로 페이징 쿼리에 영향을 주지 않습니다.

- 컬렉션은 지연 로딩으로 조회한다.
예제의 Output과 OneToMany로 매핑되어있으며 List로 선언되어있는 OutputTag에 해당됩니다. 이러한 컬렉션은 지연 로딩으로 설정하되 페치 조인은 하지 않습니다. N+1문제는 3번 방법에서 해결할 수 있습니다.

- 지연 로딩 성능 최적화를 위해 hibernate.default_batch_size 옵션을 어플리케이션 설정체 구한다.

hibernate.default_batch_size 설정을 추가하면 어플리케이션에서 글로벌하게 지연 로딩 성능이 최적화됩니다. 엔티티에서 @BatchSize 어노테이션을 추가해 개별 최적화도 가능합니다. 이 옵션을 사용하면 **컬렉션이나, 프록시 객체를 설정한 size 만큼 IN 쿼리로 조회**합니다.

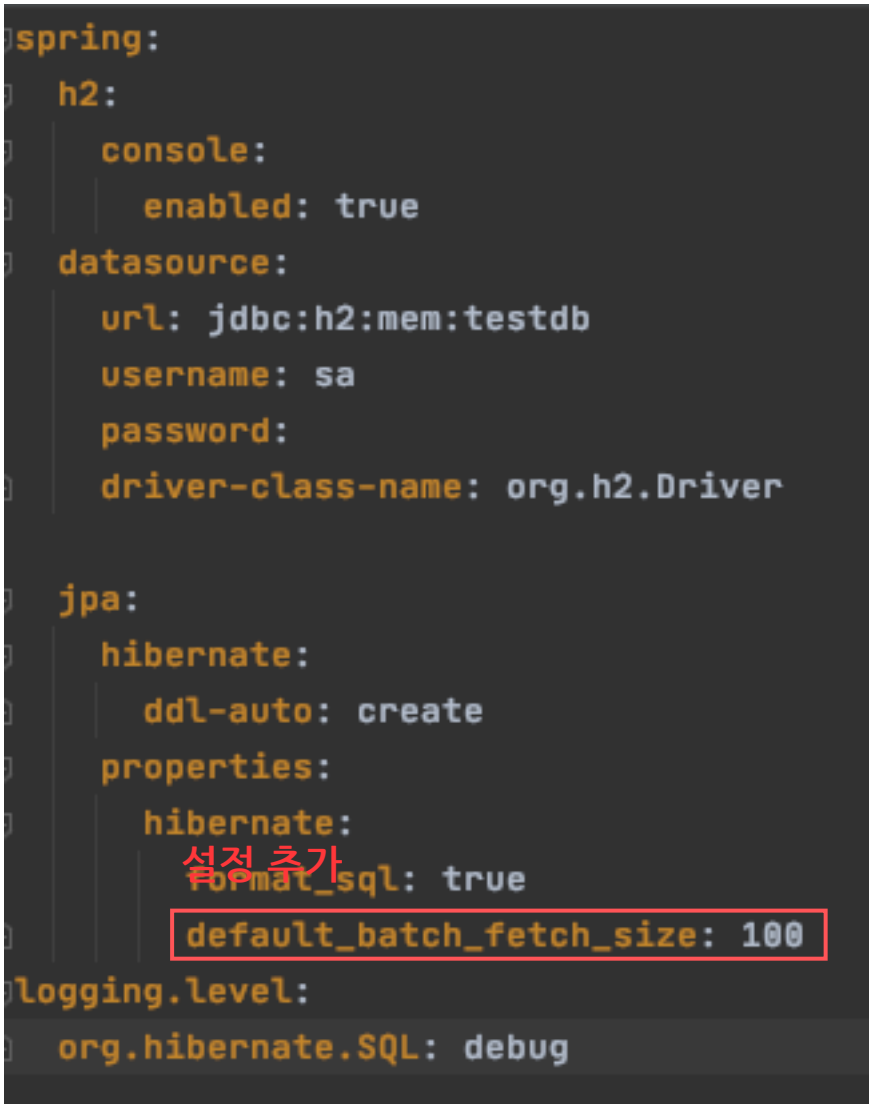
위 방법대로 예제 코드를 리팩토링 합니다.

레포지토리 계층의 findOutputsV4 메소드에서 컬렉션인 OutputTag를 페치조인 하는 부분을 제거합니다. ToOne 관계인 Project와 Category는 페치조인 합니다.

```
@Query("select o from Output o " +
        "join fetch o.project p " +
        "join fetch o.category c")
List<Output> findOutputsV4(Pageable pageable);
```

ToOne관계만 페치조인 적용

application.yml에 hibernate.default_batch_size 옵션을 추가합니다.



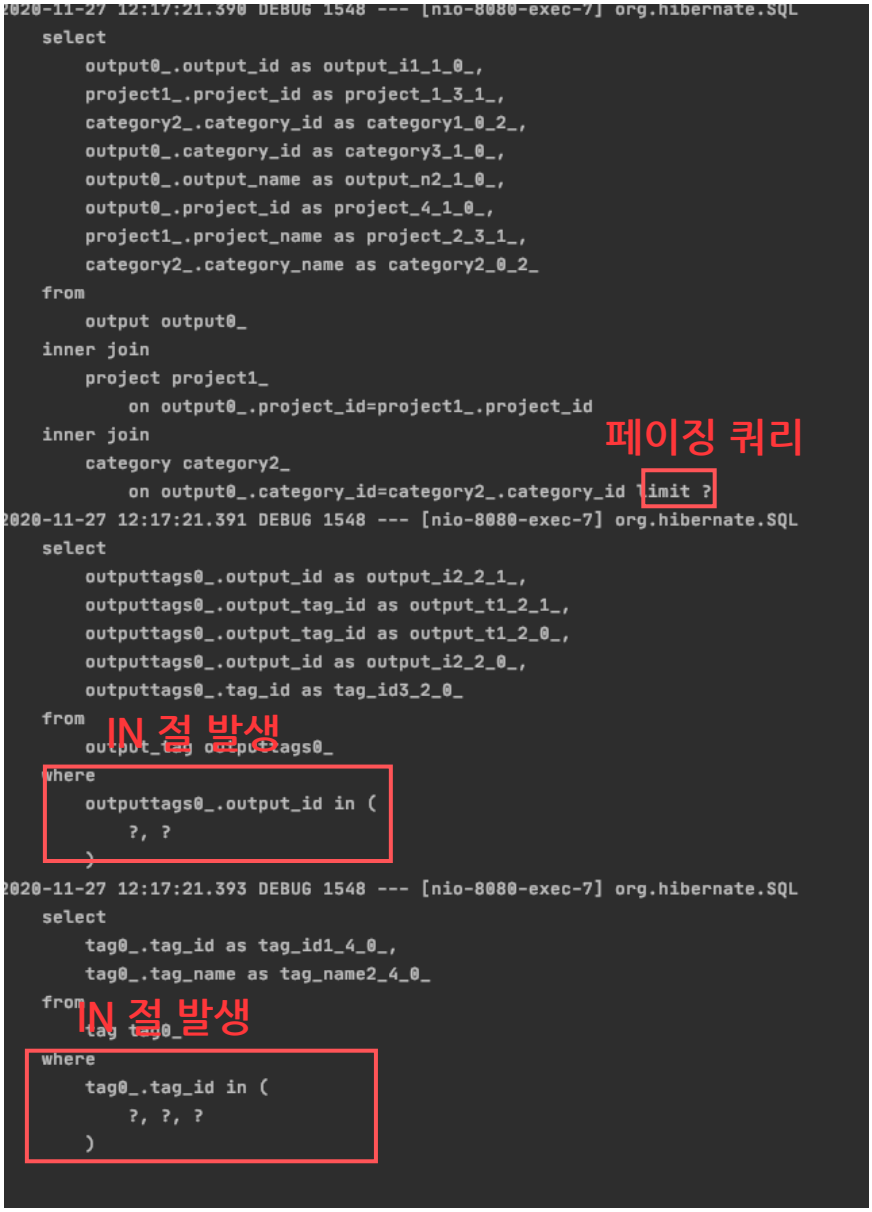
예제에서는 hibernate.default_batch_size를 100으로 지정했습니다. 보통 100~1000 사이를 선택하는 것을 권장합니다. 사이즈만큼 SQL IN 절을 사용하는데, 데이터베이스에 따라 IN 절 파라미터를 1000으로 제한하기도 합니다. 1000개 이상의 경우도 있지만, 한번에 많은 파라미터가 들어가므로 데이터베이스의 부하가 증가할 수 있습니다. 애플리케이션은 100이든 1000이든 결국 전체 데이터를 로딩해야 하므로 메모리 사용량은 동일합니다. 데이터베이스와 어플리케이션의 순간 부하를 측정하고 적당한 사이즈로 결정하는 것을 권장합니다.

findOutputV4 메소드와 hibernate.default_batch_size 설정을 추가한 후 API를 호출합니다. 첫번째 페이지와 사이즈는 2로 설정하여 API를 호출합니다. (/api/v4/outputs?page=0&size=2)

/api/v4/outputs?page=0&size=2의 결과

첫번째 페이지의 2건의 Output 데이터가 정상적으로 동작하는 것을 확인할 수 있습니다.

쿼리 로그를 확인합니다.



Output 목록을 조회할 때 페이징 쿼리가 추가된 것을 확인할 수 있습니다.
또한, 일대다 관계의 OutputTag를 조인하지 않기때문에 중복된 데이터 row수가 증가하지도 않습니다.

데이터베이스에서 동일한 쿼리를 실행했을 때도 컬렉션을 조인하지 않았기 때문에 row수가 증가하지 않는 것을 확인할 수 있습니다. 따라서 페이징 처리가 가능합니다.

OUTPUT_I1_1_0_	PROJECT_1_3_1_	CATEGORY1_0_2_	CATEGORY3_1_0_	OUTPUT_N2_1_0_	PROJECT_4_1_0_	PROJECT_2_3_1_	CATEGORY2_0_2_
10	4	1	1	Output A from project A	4	project A	category A
11	5	2	2	Output B from project B	5	project B	category B

Project와 Category에 접근할 때 발생해 쿼리가 N+1 문제가 발생한다고 생각할 수 있지만 옵션에 설정한 사이즈만큼 IN Query가 발생하기 때문에 N건의 쿼리가 발생하지 않습니다.
Output 리스트의 루프를 순회하면서 최초로 OutputTag와 Tag에 접근할 때만 In Query가 발생하고 그 이후는 추가적인 쿼리가 발생하지 않습니다.
즉, **쿼리 호출 수가 N+1에서 1+1로 최적화** 됩니다.
이 방법은 페치 조인을 할 경우보다 DB 데이터 전송량이 최적화됩니다. Output과 OutputTag를 조인하면 Output이 OutputTag 수 만큼 중복해서 조회되지만 이 방법은 조인을 하지 않으므로 DB 데이터 전송량이 감소합니다.

10. Querydsl

지금까지 JPA 성능 최적화를 위해 레포지토리 계층에서 JPQL을 직접 작성했습니다. 이렇게 직접 JPQL을 작성하면 다음과 같은 단점이 있습니다.

1. IDE의 도움을 받을 수 없다.(코드 자동완성, 문법 오류 발견)
2. 복잡한 동적 쿼리 작성이 어렵다.

Querydsl은 이러한 단점을 해결해주는 라이브러리입니다
다음은 findOutputsV4를 Querydsl을 사용해서 리팩토링한 코드입니다.

```
@Override
public List<Output> findOutputsV5(Pageable pageable) {
    return queryFactory.selectFrom(output)
            .join(output.category, category).fetchJoin()
            .join(output.project, project).fetchJoin()
            .offset(pageable.getOffset()).limit(pageable.getPageSize())
            .fetch();
}
```

연관된 엔티티를 페치조인하는 JPQL을 **Java 코드로 작성**한 것을 확인할 수 있습니다.
Querydsl을 사용해서 JPQL을 Java 코드로 작성하면 다음과 같은 장점을 얻을 수 있습니다.

1. 직관적인 문법
2. 컴파일 시점에 빠른 문법 오류 발견
3. 코드 자동완성
4. 코드 재사용
5. 동적 쿼리 작성에 용이
6. JPA에서 DTO로 바로 조회 시 깔끔한 조회 제공

이러한 장점들 때문에 Querydsl을 사용하면 높은 개발 생산성을 얻을 수 있습니다.
Querydsl은 JPQL을 코드로 만드는 빌더 역할을 할 뿐이라 JPQL을 잘 이해하면 금방 배울 수 있습니다.
따라서 JPA를 사용할 때 Querydsl을 함께 사용하는 것이 권장됩니다.

11. 정리

JPA를 사용한 조회 성능 최적화 기법을 정리하면 다음과 같습니다.

1. JPA 엔티티를 직접 노출하지말고 DTO로 변환해서 API에 노출한다.
2. 엔티티간 연관관계를 매핑할때 반드시 지연로딩을 사용한다.
3. N+1 문제를 방지하기 위해 ToOne 관계 엔티티는 페치 조인을 적용한다.
4. 컬렉션을 조회할 경우 지연로딩으로 설정후 hibernate.default_batch_size 설정해 지연 로딩 조회를 최적화한다.
5. Querydsl을 적용하여 JPQL 쿼리 작성 생산성을 높인다.

참고 자료
도서 - 자바 ORM 표준 JPA 프로그래밍(김영한)
인프런 - 실전! 스프링 부트와 JPA 활용2(김영한)