



# JPA(QueryDsl, JPQL 방식)

분류	JPA
등록일	@2024/03/08



프로젝트에 적용은 **JPQL**의 문제점을 보완한 **QueryDsl** 방식을 사용하여 코드로 작성하였다. 차이점은 아래의 블로그를 참조하자.

## JPQL vs query DSL

오늘은 JPQL과 query DSL의 차이에 대해 알아보도록 하겠다. JPQL은 JPA의 일부로 Query를 Table이 아닌 객체(=엔티티) 기준으로 작성하는 객체지향 쿼리 언어라고 정의할 수 있겠다. JPQL은 객체를 기

<https://velog.io/@cho876/JPQL-vs-query-DSL>



**QueryDSL**

JPA Module

## ORM(Object-Relational Mapping)

객체와 관계형 데이터베이스를 매핑하여 ORM 프레임 워크는 객체와 테이블을 매핑하여 패러다임 불일치 문제를 개발자 대신 해결해준다. 하이버네이트(hibernate.org)라는 오픈소스 ORM 프레임 워크가 등장하면서 하이버네이트를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌다.

# JPA(Java Persistence API)

---

Java 진영의 ORM 기술 표준이다. 즉, ORM을 사용하기 위한 인터페이스를 모아둔 것이라 볼 수 있다.

ORM에 대한 자바 API 규격이며 Hibernate, OpenJPA 등이 JPA를 구현한 구현체이며 Application과 JDBC 사이에서 동작한다. 개발자가 직접 JDBC API를 사용하지 않고 JPA 내부 JDBC API를 사용하여 SQL을 호출하고 DataBase를 간접적으로 조작한다.

- Persistence Context 영속성 컨텍스트
  - Entity를 영구 저장하는 환경이라는 의미
  - Application은 EntityManagerFactory를 통해 DB에 접근하는 트랜잭션이 생길 때마다 EntityManager를 생성하여 영속성 컨텍스트에 접근
  - 동일성 보장
    - 하나의 EntityManager에서 가져온 Entity의 동일성을 보장한다.
  - transactional write-behind 쓰기 지연
    - 한 트랜잭션 내에서 발생하는 insert update delete 에 대해 쓰기 지연 SQL 저장소에 저장된 후 트랜잭션 종료 시점에 한번에 실행한다.
  - DirtyChecking 변경 감지
    - 엔티티의 수정이 일어나도 개발자는 영속성 컨텍스트에 따로 알려주지 않아도 영속성 컨텍스트가 알아서 변경 사항을 체크해준다.
    - 1차 캐시에 entity를 저장할 때 스냅샷 필드도 따로 저장하여 commit이나 flush를 할 때 해당 entity와 스냅샷을 비교하여 변경 사항이 있으면 알아서 UPDATE SQL을 만들어서 DB에 전송한다.
  - 1차 캐시
    - 1차 캐시는 ID를 key, Entity를 값으로 하는 Map형식으로 구성되어 있다.

- 데이터 조회 시 1차 캐시에 데이터가 존재하는 경우 DB에 SELECT 쿼리를 보내는 것이 아니라 1차 캐시에서 데이터를 가져온다.
  - 1차 캐시에 존재하지 않는 데이터를 조회하는 경우에는 DB에서 SELECT 쿼리를 보내 조회하고 1차 캐시에 저장한 후 리턴한다.
- Entity Manager
  - 영속성 컨텍스트 내에서 Entity를 관리
- Entity Life Cycle
  - 비영속( new/transist)
    - Entity 객체를 생성한 상태
  - 영속(managed)
    - EntityManager의 persist(Entity)를 통해 영속성 컨텍스트에 넣어진 상태
  - 준영속(detached)
    - 영속성 컨텍스트에 저장되었다가 분리된 상태
    - 식별자가 존재하며 merge()로 다시 영속성 상태 가능
    - 1차 캐시, 쓰기 지연, 변경 감지, 지연 로딩을 포함한 영속성 컨텍스트가 제공하는 기능에 대해 동작 하지 않는다.
  - 삭제(remove)
    - 영속성 컨텍스트와 DB에서 해당 Entity를 삭제한 상태
- Entity Life Cycle



- flush 란?

영속성 컨텍스트의 변경 내용을 데이터베이스에 반영하는 것을 의미한다.

- 트랜잭션에 commit이 발생할 때
- EntityManager의 flush 메서드를 호출했을 때
- JPQL 쿼리가 실행될 때

flush가 발생하면 쓰기 지연 저장소에 저장된 SQL(INSERT/UPDATE/DELETE)이 데이터베이스로 전송된다. IO처리 시 `FileStream.flush()` 메서드를 사용하는 경험에 있어 flush 발생 시 영속성 컨텍스트가 비워지는 것이 아닌지 헷갈릴 수 있으나, 그렇지 않다. flush가 발생한다고 해서 영속성 컨텍스트가 비워지는 것이 아니고, 변경 사항을 DB와 동기화 하는 것을 의미한다. 그러한 이유로 flush가 발생해도 1차 캐시는 유지된다.

**! 세가지 경우에서 JPQL 쿼리가 실행될 때 flush가 발생하는 이유**

```
em.persist(userA); // 영속성 컨텍스트에 userA가 추가됨
em.persist(userB); // 영속성 컨텍스트에 userB가 추가됨
em.persist(userC); // 영속성 컨텍스트에 userC가 추가됨

// JPQL 실행
query = em.createQuery("select u from User u", User.class);

// 결과 조회 메소드 호출 시점에 쿼리 실행 후 결과 반환
// (실제로 데이터베이스에 쿼리를 날리는 시점)
List<User> resultList = query.getResultList();
```

## 개발 환경

- IDE : IntelliJ
- Jdk : OpenJdk 17
- DB : Mysql 8.0

- gradle
- spring boot : 3.1.8
  - spring-boot-starter-data-jpa
  - com.querydsl jpa, apt : 5.0.0 : jakarta
  - jakarta.annotation:jakarta.annotation-api
  - jakarta.persistence:jakarta.persistence-api
  - mysql:mysql-connector-java:8.0.33


Springboot 3.0.0 이상 버전을 사용할 시 아래 블로그 내용을 참조해서 Querydsl을 적용해야 한다.

(

**Querydsl 플로그인 이슈 관련 내용)**

velog

개발자들을 위한 블로그 서비스. 어디서 글 쓸지 고민하지 말고 벨로그에서 시작하세요.

 <https://velog.io/@juhyeon1114/Spring-QueryDsl-gradle-설정-Spring-boot-3.0-이상>

velog

## Querydsl

"Springboot 3.0.0 버전 이상일 때"

### 1. build.gradle 의존성 추가

온라인 상에 있는 방식은 플로그인을 추가해서 사용하지만 이 방식은 gradlew로 빌드 시 오류가 발생한다. Spring을 로컬에서 실행, 빌드하는 건 문제가 없지만 배포를 위해 **gradlew**를 사용해서 빌드를 하면, 동일한 Q 파일을 생성하지 말라는 오류가 발생한다.

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.1.8'
```

```
id 'io.spring.dependency-management' version '1.1.4'
// id 'com.ewerk.gradle.plugins.querydsl' version '1.0.10'
}
```

```
// 의존성 주입
dependencies {
    // Spring Boot
    implementation 'org.springframework.boot:
    // JPA
    implementation 'org.springframework.boot:
    compileOnly 'org.projectlombok:lombok'

    // === QueryDsl 시작 ===

    implementation 'com.querydsl:querydsl-jpa
    annotationProcessor "com.querydsl:queryds
    annotationProcessor "jakarta.annotation:j
    annotationProcessor "jakarta.persistence:

    // === QueryDsl 끝 ===
}
```

```
// === Querydsl 빌드 옵션 (선택사항) ===
def querydslDir = "$buildDir/generated/querydsl"

sourceSets {
    main.java.srcDirs += [ querydslDir ]
}

tasks.withType(JavaCompile) {
    options.annotationProcessorGeneratedSourcesDirectory = fi
}

clean.doLast {
```

```
file(querydslDir).deleteDir()  
}
```

## 2. QueryDSLConfig 생성

경로 : ./프로젝트명/config;



```
import com.querydsl.jpa.impl.JPAQueryFactory;  
import jakarta.persistence.EntityManager;  
import lombok.RequiredArgsConstructor;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
@RequiredArgsConstructor  
public class QueryDSLConfig {  
    // EntityManager를 주입받음  
    private final EntityManager entityManager;  
  
    @Bean // Bean으로 등록  
    public JPAQueryFactory jpaQueryFactory() {  
        return new JPAQueryFactory(entityManager);  
    }  
}
```



```

    }
}

```

### 3. Repository, RepositoryImpl 추가

```

// Repository

import com.hoboom.shop.shop_server.data.entity.UserEntity;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<UserEntity> {

    // 아이디로 조회
    UserEntity findById(String userId);
}

```

```

// RepositoryImpl

import com.hoboom.shop.shop_server.data.entity.UserEntity;
import com.hoboom.shop.shop_server.data.repository.UserRepository;
import com.querydsl.jpa.impl.JPAQueryFactory;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;

import static com.hoboom.shop.shop_server.data.entity.QUserEntity.userEntity;

@Repository // 레포지토리 빈으로 등록
@RequiredArgsConstructor // final로 선언된 필드를 자동으로 생성자 생성
public abstract class UserRepositoryImpl implements UserRepository {

    // JPAQueryFactory를 주입받음
    private final JPAQueryFactory query;

    @Override // 메서드 재정의
    public UserEntity findById(String userId) {
        return query.select(userEntity) // userEntity

```

```

        .from(userEntity) // userE
        .where(userEntity.userId.eq(userId)) // userI
        .fetchOne(); // 하나의
    }
}

```

#### 4. Controller, Service, ServiceImpl 추가

- Controller

```

// swagger에서 테스트

import com.hoboom.shop.shop_server.data.dto.UserDto;
import com.hoboom.shop.shop_server.service.UserService;
import io.swagger.v3.oas.annotations.Parameter;
import io.swagger.v3.oas.annotations.tags.Tag;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@Tag(name = "회원 조회", description = "회원 조회 테스트 API")
@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired // 의존성 주입
    UserService userService;

    @Parameter(name = "id", description = "userId", required = true)
    @RequestMapping(value = "/select", method = RequestMethod.GET)
    public String selectUser(UserDto userdto, @RequestParam String resStr = "");

    try {
        userdto.setuserId(id);
    }
}

```

```

        userService.selectUser(userdto);
        resStr = "회원 조회 완료\n" + "id : [" + userdto.
    } catch (Exception e) {
        resStr = e.getMessage();
    }

    return resStr;
}
}

```

- Service

```

@Service // 서비스 빈으로 등록
public interface UserService {
    // 사용자 조회
    public void selectUser(UserDto userDto);
}

```

- ServiceImpl

```

@RequiredArgsConstructor
public class UserSeviceImpl implements UserService{

    // 사용자 조회
    @Override
    public void selectUser(UserDto userDto){
        UserEntity userEntity = userRepository.findByUserI
    }
}

```

## 5. test 로그 확인

