7. 데이터 접근 기술 - Querydsl

#2.인강/8.스프링 DB 2/강의#

Querydsl 소개1 - 기존 방식의 문제점

PPT 자료 참고

실전! Querydsl 강의나 세미나를 통해 해당 내용을 들으셨던 분들은 넘어가셔도 됩니다.

Querydsl 소개2 - 해결

PPT 자료 참고

실전! Querydsl 강의나 세미나를 통해 해당 내용을 들으셨던 분들은 넘어가셔도 됩니다.

Querydsl 설정

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.6.5'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

ext["hibernate.version"] = "5.6.5.Final"

configurations {
```

```
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'

    //JdbcTemplate 추가
    //implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    //MyBatis 추가
    implementation 'org.mybatis.spring.boot:mybatis-spring-boot-starter:2.2.0'
    //JPA, 스프링 데이터 JPA 추가
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

    //Querydsl 추가
    implementation 'com.querydsl:querydsl-jpa'
    annotationProcessor "com.querydsl:querydsl-apt:$
{dependencyManagement.importedProperties['querydsl.version']}:jpa"
    annotationProcessor "jakarta.annotation:jakarta.annotation-api"
    annotationProcessor "jakarta.persistence:jakarta.persistence-api"

    //H2 데이터베이스 추가
    runtimeOnly 'com.h2database:h2'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
```

```
    useJUnitPlatform()
}

//Querydsl 추가, 자동 생성된 Q클래스 gradle clean으로 제거
clean {
    delete file('src/main/generated')
}
```

Querydsl로 추가된 부분은 다음 두 부분이다.

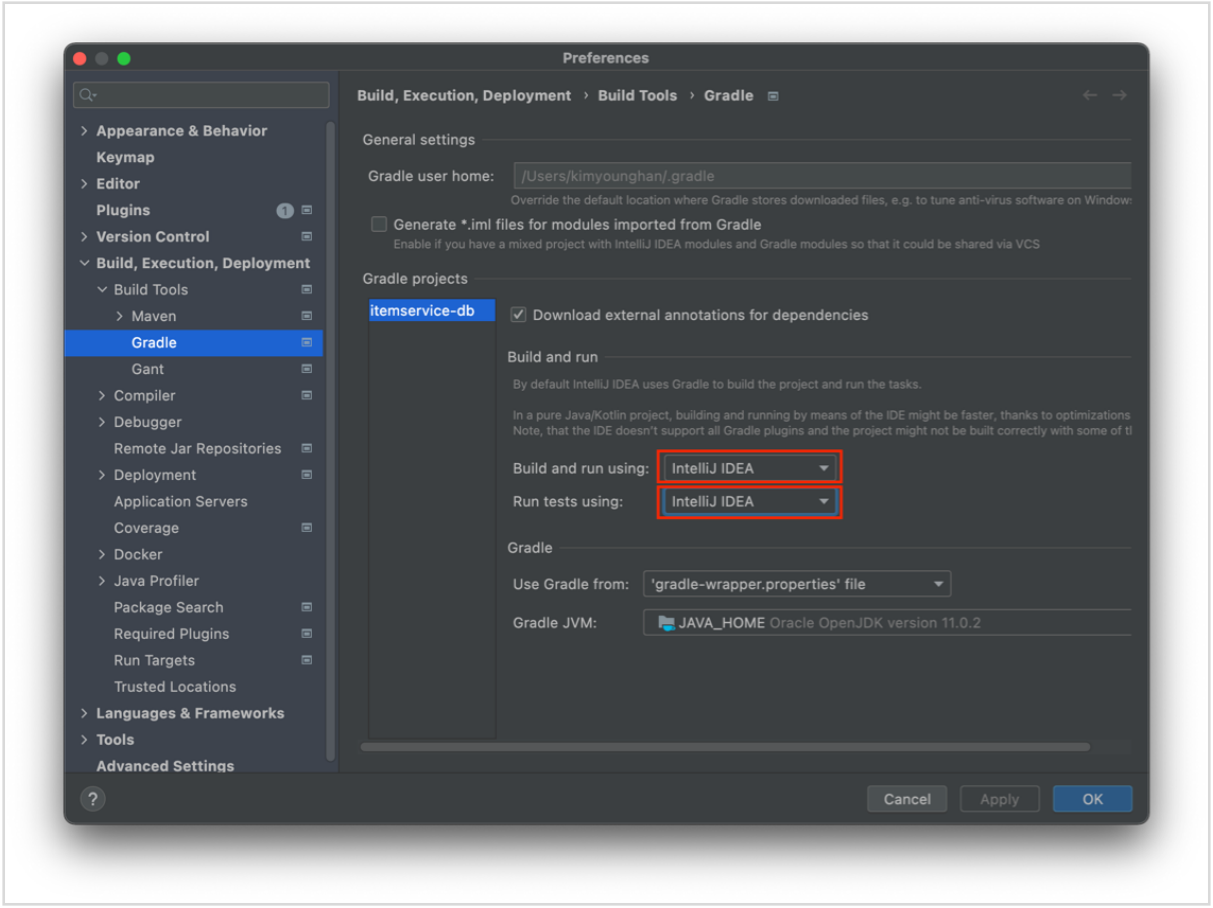
```
dependencies {
    //Querydsl 추가
    implementation 'com.querydsl:querydsl-jpa'
    annotationProcessor "com.querydsl:querydsl-apt:$
{dependencyManagement.importedProperties['querydsl.version']}:jpa"
    annotationProcessor "jakarta.annotation:jakarta.annotation-api"
    annotationProcessor "jakarta.persistence:jakarta.persistence-api"
}
```

- `"com.querydsl:querydsl-apt:$ {dependencyManagement.importedProperties['querydsl.version']}:jpa"` 은 한줄로 붙어있는 코드이다.

```
//Querydsl 추가, 자동 생성된 Q클래스 gradle clean으로 제거
clean {
    delete file('src/main/generated')
}
```

검증 - Q 타입 생성 확인 방법

- Preferences ➜ Build, Execution, Deployment ➜ Build Tools ➜ Gradle



여기에 가면 크게 2가지 옵션을 선택할 수 있다. 참고로 옵션은 둘다 같게 맞추어 두자.

- 1. **Gradle**: Gradle을 통해서 빌드한다.
- 2. **IntelliJ IDEA**: IntelliJ가 직접 자바를 실행해서 빌드한다.

옵션 선택1 - Gradle - Q타입 생성 확인 방법

Gradle IntelliJ 사용법

- `Gradle -> Tasks -> build -> clean`
- `Gradle -> Tasks -> other -> compileJava`

Gradle 콘솔 사용법

- `./gradlew clean compileJava`

Q 타입 생성 확인

- `build -> generated -> sources -> annotationProcessor -> java/main` 하위에
  - `hello.itemservice.domain.QItem` 이 생성되어 있어야 한다.

> 참고: Q타입은 컴파일 시점에 자동 생성되므로 버전관리(GIT)에 포함하지 않는 것이 좋다.
> gradle 옵션을 선택하면 Q타입은 `gradle build` 폴더 아래에 생성되기 때문에 여기를 포함하지 않아야 한다. 대부분 `gradle build` 폴더를 git에 포함하지 않기 때문에 이 부분은 자연스럽게 해결된다.

Q타입 삭제

`gradle clean`을 수행하면 `build` 폴더 자체가 삭제된다. 따라서 별도의 설정은 없어도 된다.

옵션 선택2 - IntelliJ IDEA - Q타입 생성 확인 방법

`Build -> Build Project` 또는
`Build -> Rebuild` 또는
`main()`, 또는 테스트를 실행하면 된다.

`src/main/generated` 하위에

- `hello.itemservice.domain.QItem`이 생성되어 있어야 한다.

> 참고: Q타입은 컴파일 시점에 자동 생성되므로 버전관리(GIT)에 포함하지 않는 것이 좋다.
> IntelliJ IDEA 옵션을 선택하면 Q타입은 `src/main/generated` 폴더 아래에 생성되기 때문에 여기를 포함하지 않는 것이 좋다.

Q타입 삭제

```
//Querydsl 추가, 자동 생성된 Q클래스 gradle clean으로 제거
clean {
  delete file('src/main/generated')
}
```

`IntelliJ IDEA` 옵션을 선택하면 `src/main/generated`에 파일이 생성되고, 필요한 경우 Q파일을 직접 삭제해야 한다.
`gradle`에 해당 스크립트를 추가하면 `gradle clean` 명령어를 실행할 때 `src/main/generated`의 파일도 함께 삭제해준다.

참고

Querydsl은 이렇게 설정하는 부분이 사용하면서 조금 귀찮은 부분인데, IntelliJ가 버전업 하거나 Querydsl의 Gradle 설정이 버전업 하면서 적용 방법이 조금씩 달라지기도 한다. 그리고 본인의 환경에 따라서 잘 동작하지 않기도 한다. 공식 메뉴얼에 소개 되어 있는 부분이 아니기 때문에, 설정에 수고로움이 있지만 `querydsl gradle`로 검색하면 본인 환경에 맞는 대안을 금방 찾을 수 있을 것이다.

Querydsl 적용

JpaItemRepositoryV3

```
package hello.itemservice.repository.jpa;

import com.querydsl.core.BooleanBuilder;
import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQueryFactory;
import hello.itemservice.domain.Item;
import hello.itemservice.domain.QItem;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;

import javax.persistence.EntityManager;
import java.util.List;
import java.util.Optional;

import static hello.itemservice.domain.QItem.*;

@Repository
@Transactional
public class JpaItemRepositoryV3 implements ItemRepository {

    private final EntityManager em;
    private final JPAQueryFactory query;

    public JpaItemRepositoryV3(EntityManager em) {
        this.em = em;
        this.query = new JPAQueryFactory(em);
    }

    @Override
    public Item save(Item item) {
        em.persist(item);
```

```
        return item;
    }

    @Override
    public void update(Long itemId, ItemUpdateDto updateParam) {
        Item findItem = findById(itemId).orElseThrow();
        findItem.setItemName(updateParam.getItemName());
        findItem.setPrice(updateParam.getPrice());
        findItem.setQuantity(updateParam.getQuantity());
    }

    @Override
    public Optional<Item> findById(Long id) {
        Item item = em.find(Item.class, id);
        return Optional.ofNullable(item);
    }

    public List<Item> findAllOld(ItemSearchCond itemSearch) {

        String itemName = itemSearch.getItemName();
        Integer maxPrice = itemSearch.getMaxPrice();

        QItem item = QItem.item;
        BooleanBuilder builder = new BooleanBuilder();
        if (StringUtils.hasText(itemName)) {
            builder.and(item.itemName.like("%" + itemName + "%"));
        }
        if (maxPrice != null) {
            builder.and(item.price.loe(maxPrice));
        }

        List<Item> result = query
                .select(item)
                .from(item)
                .where(builder)
                .fetch();
        return result;
    }
```

```
    @Override
    public List<Item> findAll(ItemSearchCond cond) {

        String itemName = cond.getItemName();
        Integer maxPrice = cond.getMaxPrice();

        List<Item> result = query
                .select(item)
                .from(item)
                .where(likeItemName(itemName), maxPrice(maxPrice))
                .fetch();

        return result;
    }

    private BooleanExpression likeItemName(String itemName) {
        if (StringUtils.hasText(itemName)) {
            return item.itemName.like("%" + itemName + "%");
        }
        return null;
    }

    private BooleanExpression maxPrice(Integer maxPrice) {
        if (maxPrice != null) {
            return item.price.loe(maxPrice);
        }
        return null;
    }

}
```

공통

- Querydsl을 사용하려면 `JPAQueryFactory`가 필요하다. `JPAQueryFactory`는 JPA 쿼리인 JPQL을 만들기 때문에 `EntityManager`가 필요하다.
- 설정 방식은 `JdbcTemplate`을 설정하는 것과 유사하다.
- 참고로 `JPAQueryFactory`를 스프링 빈으로 등록해서 사용해도 된다.

save(), update(), findById()

기본 기능들은 JPA가 제공하는 기본 기능을 사용한다.

findAllOld

Querydsl을 사용해서 동적 쿼리 문제를 해결한다.
`BooleanBuilder`를 사용해서 원하는 `where` 조건들을 넣어주면 된다.
이 모든 것을 자바 코드로 작성하기 때문에 동적 쿼리를 매우 편리하게 작성할 수 있다.

findAll

앞서 `findAllOld`에서 작성한 코드를 깔끔하게 리팩토링 했다. 다음 코드는 누가 봐도 쉽게 이해할 수 있을 것이다.

```
List<Item> result = query
        .select(item)
        .from(item)
        .where(likeItemName(itemName), maxPrice(maxPrice))
        .fetch();
```

- Querydsl에서 `where(A,B)`에 다양한 조건들을 직접 넣을 수 있는데, 이렇게 넣으면 AND 조건으로 처리된다. 참고로 `where()`에 `null`을 입력하면 해당 조건은 무시한다.
- 이 코드의 또 다른 장점은 `likeItemName()`, `maxPrice()`를 다른 쿼리를 작성할 때 재사용 할 수 있다는 점이다. 쉽게 이야기해서 쿼리 조건을 부분적으로 모듈화 할 수 있다. 자바 코드로 개발하기 때문에 얻을 수 있는 큰 장점이다.

이제 설정하고 실행해보자.

QuerydslConfig

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jpa.JpaItemRepositoryV3;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV1;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.persistence.EntityManager;
```

```
@Configuration
@RequiredArgsConstructor
public class QuerydslConfig {

    private final EntityManager em;

    @Bean
    public ItemService itemService() {
        return new ItemServiceV1(itemRepository());
    }

    @Bean
    public ItemRepository itemRepository() {
        return new JpaItemRepositoryV3(em);
    }

}
```

ItemServiceApplication - 변경

```
//@Import(SpringDataJpaConfig.class)
@Import(QuerydslConfig.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {}
```

- `QuerydslConfig`를 사용하도록 변경했다.

테스트를 실행하자

먼저 `ItemRepositoryTest`를 통해서 리포지토리가 정상 동작하는지 확인해보자. 테스트가 모두 성공해야 한다.

애플리케이션을 실행하자

`ItemServiceApplication`를 실행해서 애플리케이션이 정상 동작하는지 확인해보자.

예외 변환

`Querydsl`은 별도의 스프링 예외 추상화를 지원하지 않는다. 대신에 JPA에서 학습한 것 처럼 `@Repository`에서 스프링 예외 추상화를 처리해준다.

정리

Querydsl 장점

Querydsl 덕분에 동적 쿼리를 매우 깔끔하게 사용할 수 있다.

```
List<Item> result = query
        .select(item)
        .from(item)
        .where(likeItemName(itemName), maxPrice(maxPrice))
        .fetch();
```

- 쿼리 문장에 오타가 있어도 컴파일 시점에 오류를 막을 수 있다.
- 메서드 추출을 통해서 코드를 재사용할 수 있다. 예를 들어서 여기서 만든 `likeItemName(itemName)`, `maxPrice(maxPrice)` 메서드를 다른 쿼리에서도 함께 사용할 수 있다.

Querydsl을 사용해서 자바 코드로 쿼리를 작성하는 장점을 느껴보았을 것이다. 그리고 동적 쿼리 문제도 깔끔하게 해결해보았다. Querydsl은 이 외에도 수 많은 편리한 기능을 제공한다. 예를 들어서 최적의 쿼리 결과를 만들기 위해서 DTO로 편리하게 조회하는 기능은 실무에서 자주 사용하는 기능이다. JPA를 사용한다면 스프링 데이터 JPA와 Querydsl은 실무의 다양한 문제를 편리하게 해결하기 위해 선택하는 기본 기술이라 생각한다.
Querydsl에 대한 자세한 내용은 **실전! Querydsl** 강의를 참고하자.