8. 데이터 접근 기술 - 활용 방안

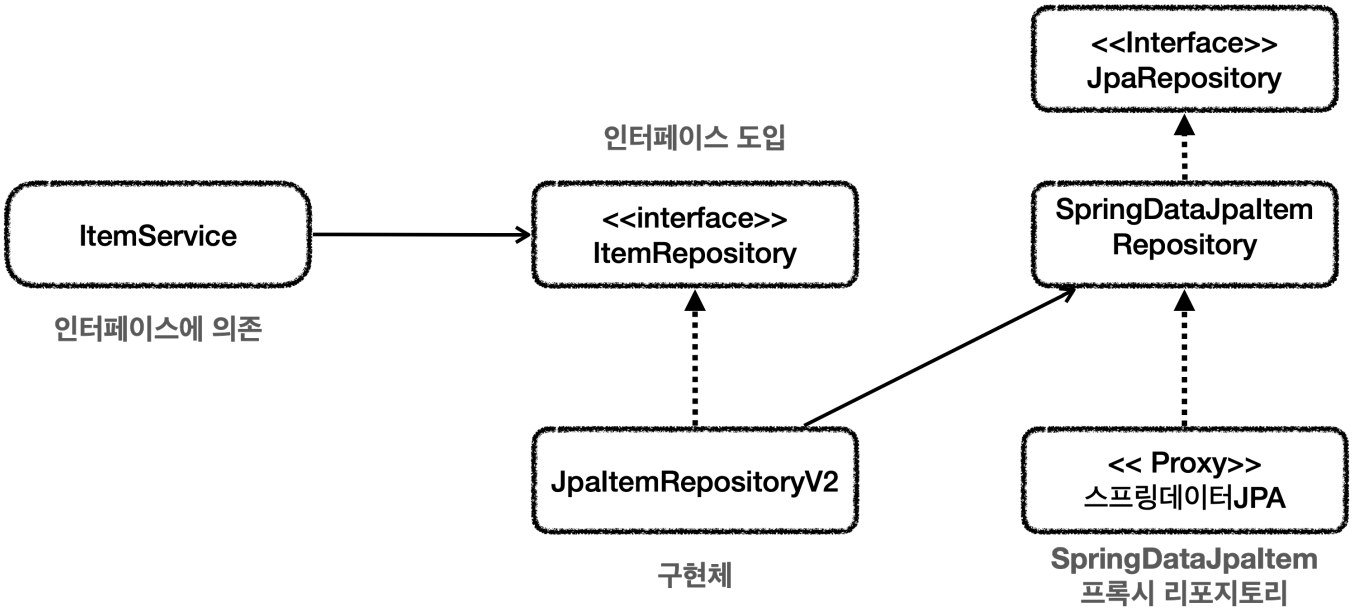
#1.인강/8.스프링 DB 2/강의#

- /스프링 데이터 JPA 예제와 트레이드 오프
- /실용적인 구조
- /다양한 데이터 접근 기술 조합
- /정리

스프링 데이터 JPA 예제와 트레이드 오프

스프링 데이터 JPA 예제를 다시 한번 돌아보자.

클래스 의존 관계



런타임 객체 의존 관계

- 중간에서 `JpaItemRepositoryV2`가 어댑터 역할을 해준 덕분에 `ItemService`가 사용하는 `ItemRepository` 인터페이스를 그대로 유지할 수 있고 클라이언트인 `ItemService`의 코드를 변경하지 않아도 되는 장점이 있다.

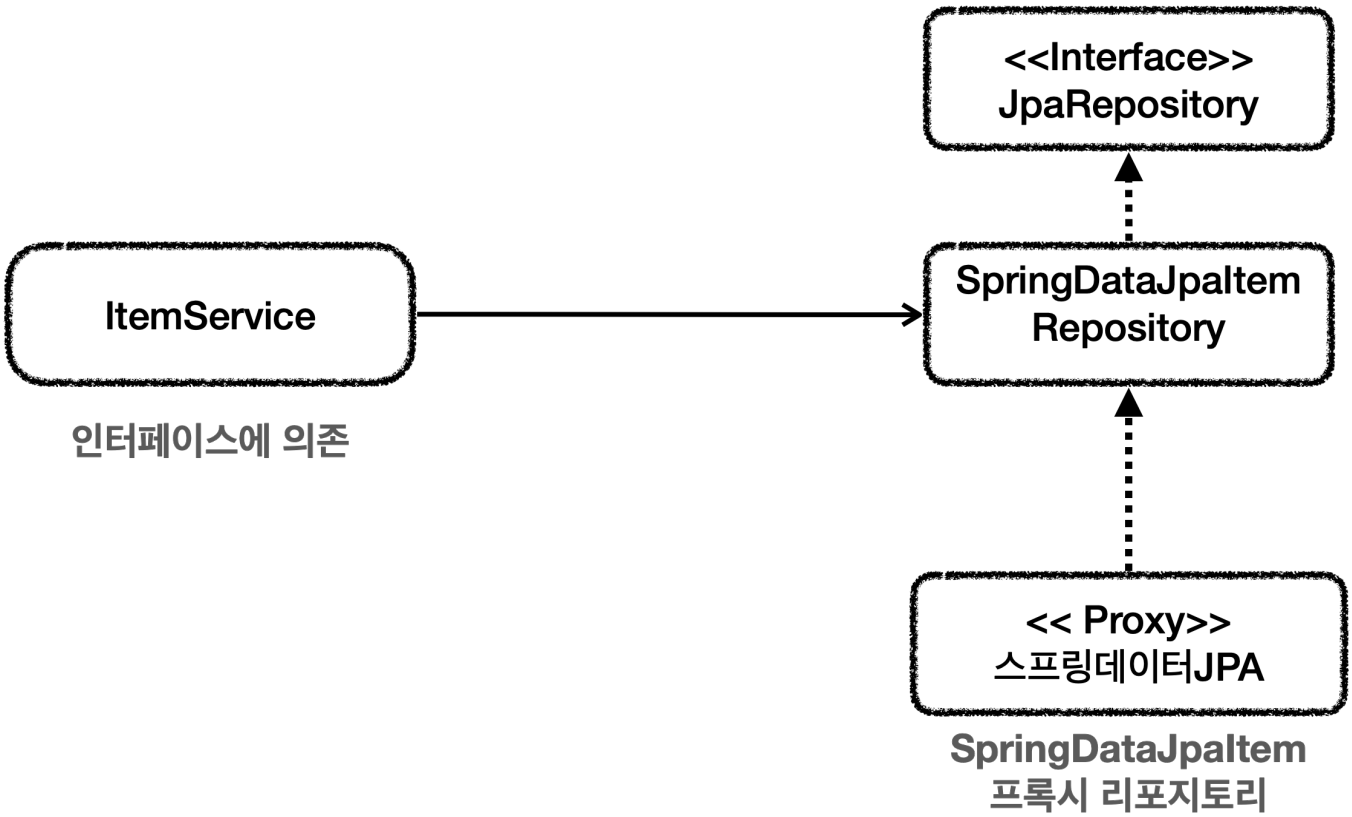
고민

- 구조를 맞추기 위해서, 중간에 어댑터가 들어가면서 전체 구조가 너무 복잡해지고 사용하는 클래스도 많아지는 단점이 생겼다.
- 실제 이 코드를 구현해야하는 개발자 입장에서 보면 중간에 어댑터도 만들고, 실제 코드까지 만들어야 하는 불편함이 생긴다.
- 유지보수 관점에서 `ItemService`를 변경하지 않고, `ItemRepository`의 구현체를 변경할 수 있는 장점이 있다. 그러니까 DI, OCP 원칙을 지킬 수 있다는 좋은 점이 분명히 있다. 하지만 반대로 구조가 복잡해지면서 어댑터 코드와 실제 코드까지 함께 유지보수 해야 하는 어려움도 발생한다.

다른 선택

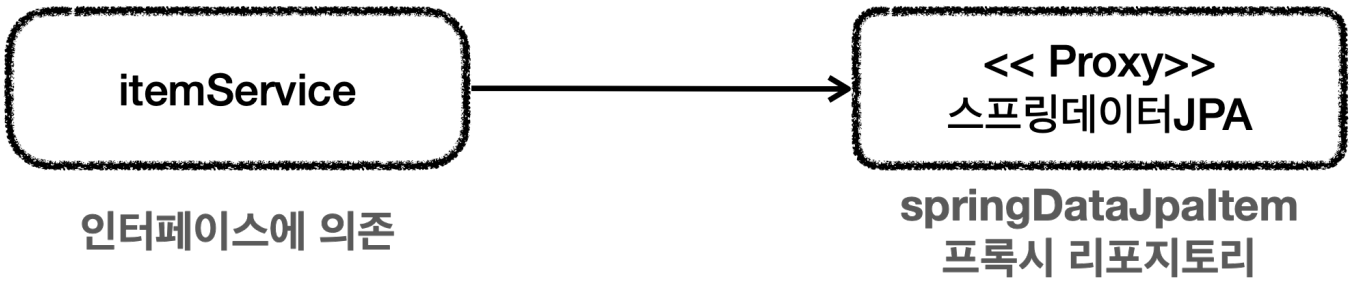
여기서 완전히 다른 선택을 할 수도 있다.
`ItemService` 코드를 일부 고쳐서 직접 스프링 데이터 JPA를 사용하는 방법이다.
DI, OCP 원칙을 포기하는 대신에, 복잡한 어댑터를 제거하고, 구조를 단순하게 가져갈 수 있는 장점이 있다.

클래스 의존 관계



- `ItemService`에서 스프링 데이터 JPA로 만든 리포지토리를 직접 참조한다. 물론 이 경우 `ItemService` 코드를 변경해야 한다.

런타임 객체 의존 관계



트레이드 오프

이것이 바로 트레이드 오프다.

- DI, OCP를 지키기 위해 어댑터를 도입하고, 더 많은 코드를 유지한다.
- 어댑터를 제거하고 구조를 단순하게 가져가지만, DI, OCP를 포기하고, `ItemService` 코드를 직접 변경한다.

결국 여기서 발생하는 트레이드 오프는 구조의 안정성 vs 단순한 구조와 개발의 편리성 사이의 선택이다.
이 둘 중에 하나의 정답만 있을까? 그렇지 않다. 어떤 상황에서는 구조의 안정성이 매우 중요하고, 어떤 상황에서는 단순한 것이 더 나은 선택일 수 있다.

개발을 할 때는 항상 자원이 무한한 것이 아니다. 그리고 어설픈 추상화는 오히려 독이 되는 경우도 많다. 무엇보다 **추상화도 비용이 든다.** 인터페이스도 비용이 든다. 여기서 말하는 비용은 유지보수 관점에서 비용을 뜻한다. 이 추상화 비용을 넘어설 만큼 효과가 있을 때 추상화를 도입하는 것이 실용적이다.

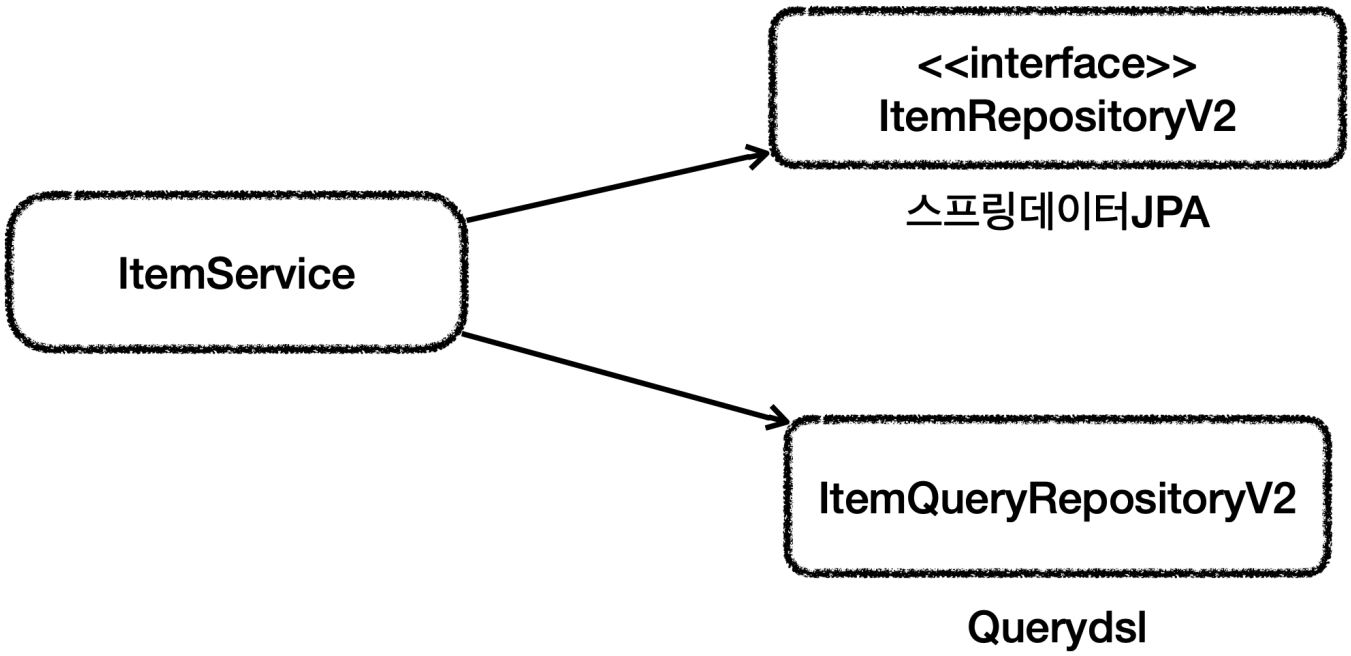
이런 선택에서 하나의 정답이 있는 것은 아니지만, 프로젝트의 현재 상황에 맞는 더 적절한 선택지가 있다고 생각한다. 그리고 현재 상황에 맞는 선택을 하는 개발자가 좋은 개발자라 생각한다.

실용적인 구조

마지막에 Querydsl을 사용한 리포지토리는 스프링 데이터 JPA를 사용하지 않는 아쉬움이 있었다. 물론 Querydsl을 사용하는 리포지토리가 스프링 데이터 JPA 리포지토리를 사용하도록 해도 된다.

이번에는 스프링 데이터 JPA의 기능은 최대한 살리면서, Querydsl도 편리하게 사용할 수 있는 구조를 만들어보겠다.

복잡한 쿼리 분리



- `ItemRepositoryV2`는 스프링 데이터 JPA의 기능을 제공하는 리포지토리이다.
- `ItemQueryRepositoryV2`는 Querydsl을 사용해서 복잡한 쿼리 기능을 제공하는 리포지토리이다.

이렇게 둘을 분리하면 기본 CRUD와 단순 조회는 스프링 데이터 JPA가 담당하고, 복잡한 조회 쿼리는 Querydsl이 담당하게 된다.
물론 `ItemService`는 기존 `ItemRepository`를 사용할 수 없기 때문에 코드를 변경해야 한다.

ItemRepositoryV2

```
package hello.itemservice.repository.v2;

import hello.itemservice.domain.Item;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ItemRepositoryV2 extends JpaRepository<Item, Long> {
}
```

- `ItemRepositoryV2`는 `JpaRepository`를 인터페이스 상속 받아서 스프링 데이터 JPA의 기능을 제공하는 리포지토리가 된다.
- 기본 CRUD는 이 기능을 사용하면 된다.
- 여기에 추가로 단순한 조회 쿼리들을 추가해도 된다.

ItemQueryRepositoryV2

```
package hello.itemservice.repository.v2;

import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQueryFactory;
```

```java
import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemSearchCond;
import org.springframework.stereotype.Repository;
import org.springframework.util.StringUtils;

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

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

@Repository
public class ItemQueryRepositoryV2 {

    private final JPAQueryFactory query;

    public ItemQueryRepositoryV2(EntityManager em) {
        this.query = new JPAQueryFactory(em);
    }

    public List<Item> findAll(ItemSearchCond cond) {
        return query.select(item)
                .from(item)
                .where(
                        maxPrice(cond.getMaxPrice()),
                        likeItemName(cond.getItemName()))
                .fetch();
    }

    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;
    }
}
```

- `ItemQueryRepositoryV2`는 Querydsl을 사용해서 복잡한 쿼리 문제를 해결한다.

- Querydsl을 사용한 쿼리 문제에 집중되어 있어서, 복잡한 쿼리는 이 부분만 유지보수 하면 되는 장점이 있다.

ItemServiceV2

```
package hello.itemservice.service;

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import hello.itemservice.repository.v2.ItemRepositoryV2;
import hello.itemservice.repository.v2.ItemQueryRepositoryV2;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
@Transactional
public class ItemServiceV2 implements ItemService {

    private final ItemRepositoryV2 itemRepositoryV2;
    private final ItemQueryRepositoryV2 itemQueryRepositoryV2;

    @Override
    public Item save(Item item) {
        return itemRepositoryV2.save(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) {
        return itemRepositoryV2.findById(id);
    }
```

```
    @Override
    public List<Item> findItems(ItemSearchCond cond) {
        return itemQueryRepositoryV2.findAll(cond);
    }
}
```

- 기존 `ItemServiceV1` 코드를 남겨두기 위해서 `ItemServiceV2`를 만들었다.
- `ItemServiceV2`는 `ItemRepositoryV2`와 `ItemQueryRepositoryV2`를 의존한다.

V2Config

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jpa.JpaItemRepositoryV3;
import hello.itemservice.repository.v2.ItemQueryRepositoryV2;
import hello.itemservice.repository.v2.ItemRepositoryV2;
import hello.itemservice.service.ItemService;
import hello.itemservice.service.ItemServiceV2;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.persistence.EntityManager;

@Configuration
@RequiredArgsConstructor
public class V2Config {

    private final EntityManager em;
    private final ItemRepositoryV2 itemRepositoryV2; //SpringDataJPA

    @Bean
    public ItemService itemService() {
        return new ItemServiceV2(itemRepositoryV2, itemQueryRepository());
    }

    @Bean
    public ItemQueryRepositoryV2 itemQueryRepository() {
        return new ItemQueryRepositoryV2(em);
    }

    @Bean
    public ItemRepository itemRepository() {
```

```
            return new JpaItemRepositoryV3(em);
        }

    }
```

- `ItemServiceV2`를 등록한 부분을 주의하자. `ItemServiceV1`이 아니라 `ItemServiceV2`이다.
- `ItemRepository`는 테스트에서 사용하므로 여전히 필요하다.

ItemServiceApplication - 변경

```
//@Import(QuerydslConfig.class)
@Import(V2Config.class)
@SpringBootApplication(scanBasePackages = "hello.itemservice.web")
public class ItemServiceApplication {}
```

- `V2Config`를 사용하도록 변경했다.

테스트를 실행하자

먼저 `ItemRepositoryTest`를 통해서 리포지토리가 정상 동작하는지 확인해보자. 참고로 테스트는 `ItemRepository`를 테스트 하는데, 현재 `JpaItemRepositoryV3`가 스프링 빈으로 등록되어 있다. `V2Config`에서 사용한 리포지토리를 테스트 하려면 `ItemQueryRepositoryV2`, `ItemRepositoryV2`용 테스트가 별도로 필요하다.

애플리케이션을 실행하자

`ItemServiceApplication`를 실행해서 애플리케이션이 정상 동작하는지 확인해보자. 애플리케이션은 `ItemServiceV2`를 사용한다.

> **참고**: 스프링 데이터 JPA가 제공하는 커스텀 리포지토리를 사용해도 비슷하게 문제를 해결할 수는 있다.

다양한 데이터 접근 기술 조합

어떤 데이터 접근 기술을 선택하는 것이 좋을까?

이 부분은 하나의 정답이 있다기 보다는, 비즈니스 상황과, 현재 프로젝트 구성원의 역량에 따라서 결정하는 것이 맞다 생각한다. `JdbcTemplate`이나 `MyBatis` 같은 기술들은 SQL을 직접 작성해야 하는 단점은 있지만 기술이 단순하기 때문에 SQL에 익숙한 개발자라면 금방 적응할 수 있다.

JPA, 스프링 데이터 JPA, Querydsl 같은 기술들은 개발 생산성을 혁신할 수 있지만, 학습 곡선이 높기 때문에, 이런 부분을 감안해야 한다. 그리고 매우 복잡한 통계 쿼리를 주로 작성하는 경우에는 잘 맞지 않는다.

개인적으로 추천하는 방향은 JPA, 스프링 데이터 JPA, Querydsl을 기본으로 사용하고, 만약 복잡한 쿼리를 써야 하는데, 해결이 잘 안되면 해당 부분에는 JdbcTemplate이나 MyBatis를 함께 사용하는 것이다.
실무에서 95% 정도는 JPA, 스프링 데이터 JPA, Querydsl 등으로 해결하고, 나머지 5%는 SQL을 직접 사용해야 하니 JdbcTemplate이나 MyBatis로 해결한다. 물론 이 비율은 프로젝트 마다 다르다. 아주 복잡한 통계 쿼리를 자주 작성해야 하면 JdbcTemplate이나 MyBatis의 비중이 높아질 수 있다.

트랜잭션 매니저 선택

JPA, 스프링 데이터 JPA, Querydsl은 모두 JPA 기술을 사용하는 것이기 때문에 트랜잭션 매니저로 `JpaTransactionManager`를 선택하면 된다. 해당 기술을 사용하면 스프링 부트는 자동으로 `JpaTransactionManager`를 스프링 빈에 등록한다.
그런데 `JdbcTemplate`, `MyBatis`와 같은 기술들은 내부에서 JDBC를 직접 사용하기 때문에 `DataSourceTransactionManager`를 사용한다.
따라서 JPA와 JdbcTemplate 두 기술을 함께 사용하면 트랜잭션 매니저가 달라진다. 결국 트랜잭션을 하나로 묶을 수 없는 문제가 발생할 수 있다. 그런데 이 부분은 걱정하지 않아도 된다.

JpaTransactionManager의 다양한 지원

`JpaTransactionManager`는 놀랍게도 `DataSourceTransactionManager`가 제공하는 기능도 대부분 제공한다. JPA라는 기술도 결국 내부에서는 DataSource와 JDBC 커넥션을 사용하기 때문이다. 따라서 `JdbcTemplate`, `MyBatis`와 함께 사용할 수 있다.
결과적으로 `JpaTransactionManager`를 하나만 스프링 빈에 등록하면, JPA, JdbcTemplate, MyBatis 모두를 하나의 트랜잭션으로 묶어서 사용할 수 있다. 물론 함께 롤백도 할 수 있다.

주의점

이렇게 JPA와 JdbcTemplate을 함께 사용할 경우 JPA의 플러시 타이밍에 주의해야 한다. JPA는 데이터를 변경하면 변경 사항을 즉시 데이터베이스에 반영하지 않는다. 기본적으로 트랜잭션이 커밋되는 시점에 변경 사항을 데이터베이스에 반영한다. 그래서 하나의 트랜잭션 안에서 JPA를 통해 데이터를 변경한 다음에 JdbcTemplate을 호출하는 경우 JdbcTemplate에서는 JPA가 변경한 데이터를 읽기 못하는 문제가 발생한다.
이 문제를 해결하려면 JPA 호출이 끝난 시점에 JPA가 제공하는 플러시라는 기능을 사용해서 JPA의 변경 내역을 데이터베이스에 반영해주어야 한다. 그래야 그 다음에 호출되는 JdbcTemplate에서 JPA가 반영한 데이터를 사용할 수 있다.

참고로 방금 설명한 JPA 플러시에 대한 부분은 JPA를 학습해야 이해할 수 있다. 지금은 `JpaTransactionManager`를 사용해서 여러 데이터 접근 기술들을 함께 사용할 수 있다는 점만 기억하자.

정리

`ItemServiceV2`는 스프링 데이터 JPA를 제공하는 `ItemRepositoryV2`도 참조하고, Querydsl과 관련된 `ItemQueryRepositoryV2`도 직접 참조한다. 덕분에 `ItemRepositoryV2`를 통해서 스프링 데이터 JPA 기능을 적절히 활용할 수 있고, `ItemQueryRepositoryV2`를 통해서 복잡한 쿼리를 Querydsl로 해결할 수 있다.

이렇게 하면서 구조의 복잡함 없이 단순하게 개발할 수 있다.
본인이 진행하는 프로젝트의 규모가 작고, 속도가 중요하고, 프로토타입 같은 시작 단계라면 이렇게 단순하면서 라이브러리의 지원을 최대한 편리하게 받는 구조가 더 나은 선택일 수 있다.
하지만 이 구조는 리포지토리의 구현 기술이 변경되면 수 많은 코드를 변경해야 하는 단점이 있다.

이런 선택에서 하나의 정답은 없다. 이런 트레이드 오프를 알고, 현재 상황에 더 맞는 적절한 선택을 하는 좋은 개발자가 있을 뿐이다.
여러분도 이런 트레이드 오프를 고민하고, 현재 상황에 맞는 더 나은 선택을 하기 위해 많이 고민하면 좋겠다. 그 시간들이 쌓이면 분명 좋은 개발자가 되어 있을 것이다.