5. 데이터 접근 기술 - JPA

#2.인강/8.스프링 DB 2/강의#

- JPA 시작
- ORM 개념1 - SQL 중심적인 개발의 문제점
- ORM 개념2 - JPA 소개
- JPA 설정
- JPA 적용1 - 개발
- JPA 적용2 - 리포지토리 분석
- JPA 적용3 - 예외 변환
- 정리

JPA 시작

스프링과 JPA는 자바 엔터프라이즈(기업) 시장의 주력 기술이다.
스프링이 DI 컨테이너를 포함한 애플리케이션 전반의 다양한 기능을 제공한다면, JPA는 ORM 데이터 접근 기술을 제공한다.

스프링 + 데이터 접근기술의 조합을 구글 트랜드로 비교했을 때

- 글로벌에서는 스프링+JPA 조합을 80%이상 사용한다.
- 국내에서도 스프링 + JPA 조합을 50%정도 사용하고, 2015년 부터 점점 그 추세가 증가하고 있다.

JPA는 스프링 만큼이나 방대하고, 학습해야할 분량도 많다. 하지만 한번 배워두면 데이터 접근 기술에서 매우 큰 생산성 향상을 얻을 수 있다. 대표적으로 JdbcTemplate이나 MyBatis 같은 SQL 매퍼 기술은 SQL을 개발자가 직접 작성해야 하지만, JPA를 사용하면 SQL도 JPA가 대신 작성하고 처리해준다.

실무에서는 JPA를 더욱 편리하게 사용하기 위해 스프링 데이터 JPA와 Querydsl이라는 기술을 함께 사용한다.
중요한 것은 JPA이다. 스프링 데이터 JPA, Querydsl은 JPA를 편리하게 사용하도록 도와주는 도구라 생각하면 된다.
이 강의에서는 모든 내용을 다루지 않고, JPA와 스프링 데이터 JPA, 그리고 Querydsl로 이어지는 전체 그림을 볼 것이다. 그리고 이 기술들을 우리 애플리케이션에 적용하면서 자연스럽게 왜 사용해야 하는지, 그리고 어떤 장점이 있는지 이해할 수 있게 된다.
이렇게 전체 그림을 보고 나면 앞으로 어떻게 공부해야 할지 쉽게 접근할 수 있을 것이다.

참고

각각의 기술들은 별도의 강의로 다룰 정도로 내용이 방대하다. 여기서는 해당 기술들의 기본 기능과, 왜 사용해야 하는지 각각의 장단점을 알아본다. 각 기술들의 자세한 내용은 다음 강의를 참고하자.

- JPA - 자바 ORM 표준 JPA 프로그래밍 - 기본편
- 스프링 데이터 JPA - 실전! 스프링 데이터 JPA
- Querydsl - 실전! Querydsl

ORM 개념1 - SQL 중심적인 개발의 문제점

PPT 자료 참고
JPA 기본편 강의나 세미나를 통해 해당 내용을 들으셨던 분들은 넘어가셔도 됩니다.

ORM 개념2 - JPA 소개

PPT 자료 참고
JPA 기본편 강의나 세미나를 통해 해당 내용을 들으셨던 분들은 넘어가셔도 됩니다.

JPA 설정

`spring-boot-starter-data-jpa` 라이브러리를 사용하면 JPA와 스프링 데이터 JPA를 스프링 부트와 통합하고, 설정도 아주 간단히 할 수 있다.
`spring-boot-starter-data-jpa` 라이브러리를 사용해서 간단히 설정하는 방법을 알아보자.

`build.gradle` 에 다음 의존 관계를 추가한다.

```
//JPA, 스프링 데이터 JPA 추가
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

`build.gradle` 에 다음 의존 관계를 제거한다.

```
//JdbcTemplate 추가
//implementation 'org.springframework.boot:spring-boot-starter-jdbc'
```

`spring-boot-starter-data-jpa`는 `spring-boot-starter-jdbc`도 함께 포함(의존)한다. 따라서 해당 라이브러리 의존관계를 제거해도 된다. 참고로 `mybatis-spring-boot-starter`도 `spring-boot-starter-jdbc`를 포함하기 때문에 제거해도 된다.

build.gradle - 의존관계 전체

```
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'

    //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'
}
```

다음과 같은 라이브러리가 추가된다.

- `hibernate-core` : JPA 구현체인 하이버네이트 라이브러리
- `jakarta.persistence-api` : JPA 인터페이스
- `spring-data-jpa` : 스프링 데이터 JPA 라이브러리

`application.properties`에 다음 설정을 추가하자.

`main - application.properties`

```
#JPA log
```

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

`test - application.properties`

```
#JPA log
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

- `org.hibernate.SQL=DEBUG` : 하이버네이트가 생성하고 실행하는 SQL을 확인할 수 있다.
- `org.hibernate.type.descriptor.sql.BasicBinder=TRACE` : SQL에 바인딩 되는 파라미터를 확인할 수 있다.
- `spring.jpa.show-sql=true` : 참고로 이런 설정도 있다. 이전 설정은 `logger` 를 통해서 SQL이 출력된다. 이 설정은 `System.out` 콘솔을 통해서 SQL이 출력된다. 따라서 이 설정은 권장하지는 않는다. (둘다 켜면 `logger` , `System.out` 둘다 로그가 출력되어서 같은 로그가 중복해서 출력된다.)

JPA 적용1 - 개발

JPA에서 가장 중요한 부분은 객체와 테이블을 매핑하는 것이다. JPA가 제공하는 애노테이션을 사용해서 `Item` 객체와 테이블을 매핑해보자.

Item - ORM 매핑

```
package hello.itemservice.domain;

import lombok.Data;

import javax.persistence.*;

@Data
@Entity
public class Item {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(name = "item_name", length = 10)
    private String itemName;
    private Integer price;
    private Integer quantity;

    public Item() {
    }

    public Item(String itemName, Integer price, Integer quantity) {
        this.itemName = itemName;
        this.price = price;
        this.quantity = quantity;
    }
}
```

- `@Entity` : JPA가 사용하는 객체라는 뜻이다. 이 에노테이션이 있어야 JPA가 인식할 수 있다. 이렇게 `@Entity` 가 붙은 객체를 JPA에서는 엔티티라 한다.
- `@Id` : 테이블의 PK와 해당 필드를 매핑한다.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` : PK 생성 값을 데이터베이스에서 생성하는 `IDENTITY` 방식을 사용한다. 예) MySQL auto increment
- `@Column` : 객체의 필드를 테이블의 컬럼과 매핑한다.
  - `name = "item_name"` : 객체는 `itemName` 이지만 테이블의 컬럼은 `item_name` 이므로 이렇게 매핑했다.
  - `length = 10` : JPA의 매핑 정보로 DDL(`create table`)도 생성할 수 있는데, 그때 컬럼의 길이 값으로 활용된다. (`varchar 10`)
  - `@Column` 을 생략할 경우 필드의 이름을 테이블 컬럼 이름으로 사용한다. 참고로 지금처럼 스프링 부트와 통합해서 사용하면 필드 이름을 테이블 컬럼 명으로 변경할 때 객체 필드의 카멜 케이스를 테이블 컬럼의 언더스코어로 자동으로 변환해준다.
    - `itemName` ➡ `item_name` , 따라서 위 예제의 `@Column(name = "item_name")` 를 생략해도 된다.

JPA는 `public` 또는 `protected` 의 기본 생성자가 필수이다. 기본 생성자를 꼭 넣어주자.

```
public Item() {}
```

이렇게 하면 기본 매핑은 모두 끝난다. 이제 JPA를 실제 사용하는 코드를 작성해보자.

우선 코드를 작성하고 실행하면서 하나씩 알아보자.

JpaItemRepositoryV1

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

import hello.itemservice.domain.Item;
import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.ItemSearchCond;
import hello.itemservice.repository.ItemUpdateDto;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;

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

@Slf4j
@Repository
@Transactional
public class JpaItemRepositoryV1 implements ItemRepository {

    private final EntityManager em;

    public JpaItemRepositoryV1(EntityManager em) {
        this.em = em;
    }

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

    @Override
```

```
    public void update(Long itemId, ItemUpdateDto updateParam) {
        Item findItem = em.find(Item.class, itemId);
        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);
    }

    @Override
    public List<Item> findAll(ItemSearchCond cond) {
        String jpql = "select i from Item i";

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

        if (StringUtils.hasText(itemName) || maxPrice != null) {
            jpql += " where";
        }

        boolean andFlag = false;
        if (StringUtils.hasText(itemName)) {
            jpql += " i.itemName like concat('%',:itemName,'%')";
            andFlag = true;
        }

        if (maxPrice != null) {
            if (andFlag) {
                jpql += " and";
            }
            jpql += " i.price <= :maxPrice";
        }

        log.info("jpql={}", jpql);
```

```
        TypedQuery<Item> query = em.createQuery(jpql, Item.class);
        if (StringUtils.hasText(itemName)) {
            query.setParameter("itemName", itemName);
        }
        if (maxPrice != null) {
            query.setParameter("maxPrice", maxPrice);
        }
        return query.getResultList();
    }
}
```

- `private final EntityManager em` : 생성자를 보면 스프링을 통해 엔티티 매니저(`EntityManager`)라는 것을 주입받은 것을 확인할 수 있다. JPA의 모든 동작은 엔티티 매니저를 통해서 이루어진다. 엔티티 매니저는 내부에 데이터소스를 가지고 있고, 데이터베이스에 접근할 수 있다.
- `@Transactional` : JPA의 모든 데이터 변경(등록, 수정, 삭제)은 트랜잭션 안에서 이루어져야 한다. 조회는 트랜잭션이 없어도 가능하다. 변경의 경우 일반적으로 서비스 계층에서 트랜잭션을 시작하기 때문에 문제가 없다. 하지만 이번 예제에서는 복잡한 비즈니스 로직이 없어서 서비스 계층에서 트랜잭션을 걸지 않았다. JPA에서는 데이터 변경시 트랜잭션이 필수다. 따라서 리포지토리에 트랜잭션을 걸어주었다. 다시한번 강조하지만 일반적으로는 비즈니스 로직을 시작하는 서비스 계층에 트랜잭션을 걸어주는 것이 맞다.

참고: JPA를 설정하려면 `EntityManagerFactory` , JPA 트랜잭션 매니저(`JpaTransactionManager`), 데이터소스 등등 다양한 설정을 해야 한다. 스프링 부트는 이 과정을 모두 자동화 해준다. `main()` 메서드부터 시작해서 JPA를 처음부터 어떻게 설정하는지는 JPA 기본편을 참고하자. 그리고 스프링 부트의 자동 설정은 `JpaBaseConfiguration` 를 참고하자.

먼저 설정을 완료하고 실행한 다음에, 코드를 분석해보자.

JpaConfig

```
package hello.itemservice.config;

import hello.itemservice.repository.ItemRepository;
import hello.itemservice.repository.jpa.JpaItemRepositoryV1;
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
public class JpaConfig {

    private final EntityManager em;

    public JpaConfig(EntityManager em) {
        this.em = em;
    }

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

    @Bean
    public ItemRepository itemRepository() {
        return new JpaItemRepositoryV1(em);
    }

}
```

- 설정은 이해하는데 크게 어렵지 않을 것이다.

ItemServiceApplication - 변경

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

- `JpaConfig`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.

JPA 적용2 - 리포지토리 분석

JpaItemRepositoryV1 코드를 분석해보자.

save() - 저장

```
public Item save(Item item) {
    em.persist(item);
    return item;
}
```

- `em.persist(item)` : JPA에서 객체를 테이블에 저장할 때는 엔티티 매니저가 제공하는 `persist()` 메서드를 사용하면 된다.

JPA가 만들어서 실행한 SQL

```
insert into item (id, item_name, price, quantity) values (null, ?, ?, ?)
또는
insert into item (id, item_name, price, quantity) values (default, ?, ?, ?)
또는
insert into item (item_name, price, quantity) values (?, ?, ?)
```

- JPA가 만들어서 실행한 SQL을 보면 `id`에 값이 빠져있는 것을 확인할 수 있다. PK 키 생성 전략을 `IDENTITY`로 사용했기 때문에 JPA가 이런 쿼리를 만들어서 실행한 것이다. 물론 쿼리 실행 이후에 `Item` 객체의 `id` 필드에 데이터베이스가 생성한 PK값이 들어가게 된다. (JPA가 INSERT SQL 실행 이후에 생성된 ID 결과를 받아서 넣어준다)

PK 매핑 참고

```
@Entity
public class Item {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
}
```

update() - 수정

```
public void update(Long itemId, ItemUpdateDto updateParam) {
    Item findItem = em.find(Item.class, itemId);
    findItem.setItemName(updateParam.getItemName());
    findItem.setPrice(updateParam.getPrice());
    findItem.setQuantity(updateParam.getQuantity());
}
```

JPA가 만들어서 실행한 SQL

```
update item set item_name=?, price=?, quantity=? where id=?
```

- `em.update()` 같은 메서드를 전혀 호출하지 않았다. 그런데 어떻게 UPDATE SQL이 실행되는 것일까?
- JPA는 트랜잭션이 커밋되는 시점에, 변경된 엔티티 객체가 있는지 확인한다. 특정 엔티티 객체가 변경된 경우에는 UPDATE SQL을 실행한다.
- JPA가 어떻게 변경된 엔티티 객체를 찾는지 명확하게 이해하려면 영속성 컨텍스트라는 JPA 내부 원리를 이해해야 한다. 이 부분은 JPA 기본편에서 자세히 다룬다. 지금은 트랜잭션 커밋 시점에 JPA가 변경된 엔티티 객체를 찾아서 UPDATE SQL을 수행한다고 이해하면 된다.
- 테스트의 경우 마지막에 트랜잭션이 롤백되기 때문에 JPA는 UPDATE SQL을 실행하지 않는다.
  테스트에서 UPDATE SQL을 확인하려면 `@Commit` 을 붙이면 확인할 수 있다.

findById() - 단건 조회

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

- JPA에서 엔티티 객체를 PK를 기준으로 조회할 때는 `find()` 를 사용하고 조회 타입과, PK 값을 주면 된다.
  그러면 JPA가 다음과 같은 조회 SQL을 만들어서 실행하고, 결과를 객체로 바로 변환해준다.

JPA가 만들어서 실행한 SQL

```
select
  item0_.id as id1_0_0_,
  item0_.item_name as item_nam2_0_0_,
  item0_.price as price3_0_0_,
  item0_.quantity as quantity4_0_0_
from item item0_
where item0_.id=?
```

JPA(하이버네이트)가 만들어서 실행한 SQL은 별칭이 조금 복잡하다. 조인이 발생하거나 복잡한 조건에서도 문제 없도록 기계적으로 만들다 보니 이런 결과가 나온 듯 하다.

JPA에서 단순히 PK를 기준으로 조회하는 것이 아닌, 여러 데이터를 복잡한 조건으로 데이터를 조회하려면 어떻게 하면 될까?

findAll - 목록 조회

```
public List<Item> findAll(ItemSearchCond cond) {
    String jpql = "select i from Item i";
    //동적 쿼리 생략
    TypedQuery<Item> query = em.createQuery(jpql, Item.class);
    return query.getResultList();
}
```

JPQL

JPA는 JPQL(Java Persistence Query Language)이라는 객체지향 쿼리 언어를 제공한다.
주로 여러 데이터를 복잡한 조건으로 조회할 때 사용한다.
SQL이 테이블을 대상으로 한다면, JPQL은 엔티티 객체를 대상으로 SQL을 실행한다 생각하면 된다.
엔티티 객체를 대상으로 하기 때문에 `from` 다음에 `Item` 엔티티 객체 이름이 들어간다. 엔티티 객체와 속성의 대소문자는 구분해야 한다.
JPQL은 SQL과 문법이 거의 비슷하기 때문에 개발자들이 쉽게 적응할 수 있다.

결과적으로 JPQL을 실행하면 그 안에 포함된 엔티티 객체의 매핑 정보를 활용해서 SQL을 만들게 된다.

실행된 JPQL

```
select i from Item i
where i.itemName like concat('%',:itemName,'%')
```

```
  and i.price <= :maxPrice
```

JPQL을 통해 실행된 SQL

```
select
  item0_.id as id1_0_,
  item0_.item_name as item_nam2_0_,
  item0_.price as price3_0_,
  item0_.quantity as quantity4_0_
from item item0_
where (item0_.item_name like ('%'||?||'%'))
  and item0_.price<=?
```

파라미터

JPQL에서 파라미터는 다음과 같이 입력한다.

- `where price <= :maxPrice`

파라미터 바인딩은 다음과 같이 사용한다.

- `query.setParameter("maxPrice", maxPrice)`

동적 쿼리 문제

JPA를 사용해도 동적 쿼리 문제가 남아있다. 동적 쿼리는 뒤에서 설명하는 Querydsl이라는 기술을 활용하면 매우 깔끔하게 사용할 수 있다. 실무에서는 동적 쿼리 문제 때문에, JPA 사용할 때 Querydsl도 함께 선택하게 된다.

> **참고**
> JPQL에 대한 자세한 내용은 JPA 기본편 강의를 참고하자.

JPA 적용3 - 예외 변환

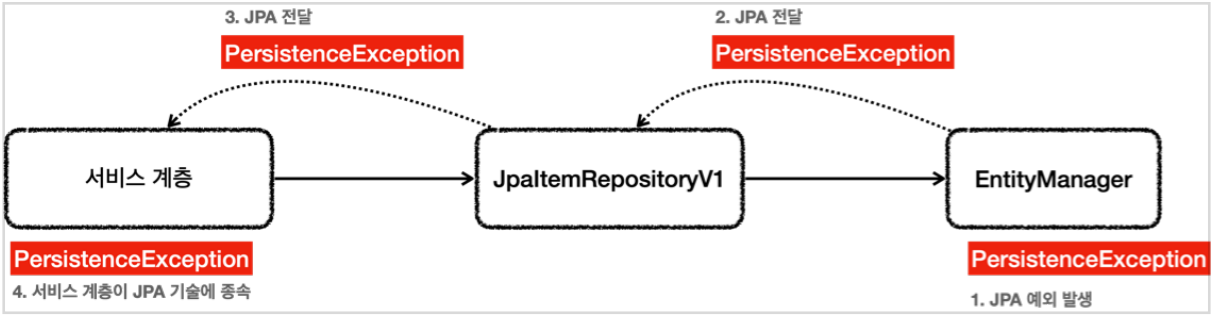
JPA의 경우 예외가 발생하면 JPA 예외가 발생하게 된다.

```
@Repository
@Transactional
```

```java
public class JpaItemRepositoryV1 implements ItemRepository {

    private final EntityManager em;

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

}
```

- `EntityManager`는 순수한 JPA 기술이고, 스프링과는 관계가 없다. 따라서 엔티티 매니저는 예외가 발생하면 JPA 관련 예외를 발생시킨다.
- JPA는 `PersistenceException`과 그 하위 예외를 발생시킨다.
  - 추가로 JPA는 `IllegalStateException`, `IllegalArgumentException`을 발생시킬 수 있다.
- 그렇다면 JPA 예외를 스프링 예외 추상화(`DataAccessException`)로 어떻게 변환할 수 있을까?
- 비밀은 바로 `@Repository`에 있다.

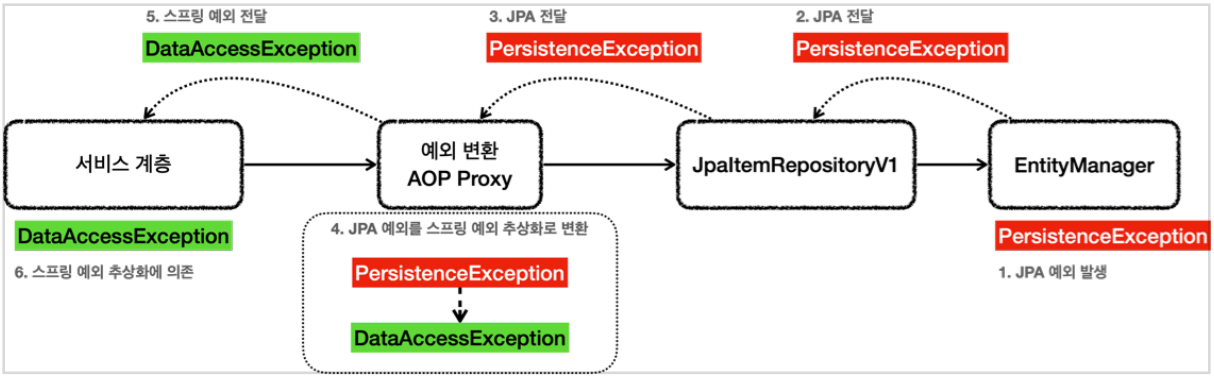
예외 변환 전



@Repository의 기능

- `@Repository`가 붙은 클래스는 컴포넌트 스캔의 대상이 된다.
- `@Repository`가 붙은 클래스는 예외 변환 AOP의 적용 대상이 된다.
  - 스프링과 JPA를 함께 사용하는 경우 스프링은 JPA 예외 변환기(`PersistenceExceptionTranslator`)를 등록한다.
  - 예외 변환 AOP 프록시는 JPA 관련 예외가 발생하면 JPA 예외 변환기를 통해 발생한 예외를 스프링 데이터 접근 예외로 변환한다.

예외 변환 후



결과적으로 리포지토리에 `@Repository` 애노테이션만 있으면 스프링이 예외 변환을 처리하는 AOP를 만들어준다.

> **참고**
> 스프링 부트는 `PersistenceExceptionTranslationPostProcessor`를 자동으로 등록하는데, 여기에서 `@Repository`를 AOP 프록시로 만드는 어드바이저가 등록된다.

> **참고**
> 복잡한 과정을 거쳐서 실제 예외를 변환하는데, 실제 JPA 예외를 변환하는 코드는 `EntityManagerFactoryUtils.convertJpaAccessExceptionIfPossible()`이다.

정리

JPA에 대한 자세한 내용은 **JPA - 자바 ORM 표준 JPA 프로그래밍 - 기본편** 강의를 참고하자.