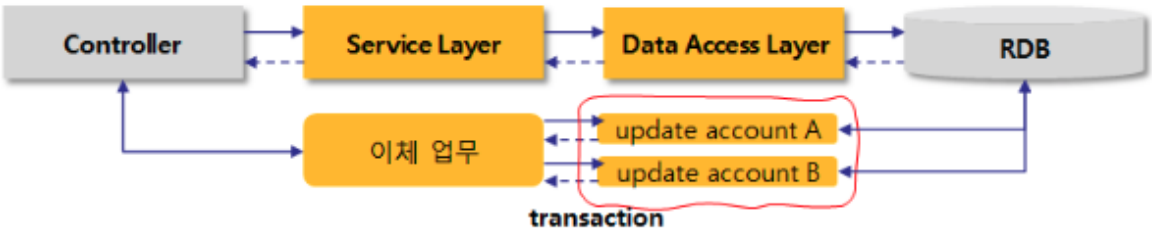
7/13 ~ 7/17 2주차 - 5

Model에서 Service Layer의 역할

오늘 MyBatis를 Spring에 CRUD를 연동하면서 DAO와 Service를 나누는 게 단순해보이고 의미가 없어보였는데 이 내용에 대한 글을 찾았고 정리해보려고한다.

- MVC 디자인 패턴에서 Model은 크게 Service Layer와 Data Access Layer로 구성된다.

일단 MyBatis 를 이용해서 Data Access Layer에 대한 내용은 추후 정리하기.



Service Layer의 역할은 무엇일까 ?

먼저, Data Access Layar는 개별 SQL의 처리를 목표로 한다.

즉, 여러 sql을 묶어서 사용하지 않는다.

하지만 실제 요청 처리를 할대는 여러개의 sql들이 뭉쳐서 동작한다. (이래서 나눠야하는구나...)

글에 있는 예시로..

Country 테이블에서 특정 국가 정보 삭제라는 업무를 생각해보자!

Country는 City에 의해 참조되고 있기 때문에 (도시는 당연히 나라 정보가 있어야한다.)

먼저 City 테이블에서 해당 Country에 속하는 데이터를 모두 지운후 (참조 무결성이니까 먼저)

Country 테이블에서 삭제해야 한다.

만약 A라는 코드의 국가를 지운다고 가정을 해보면, 개별 sql로는 delete from city where countryCode='A'와 delete from country where code='A' 2개의 sql문이 필요하다.

그런데...

만약 City에서의 삭제가 잘 진행된 후 어떤 원인에 의해 Country에서 삭제를 실패했다면

어떻게 될까?

그 상태로 DB에 반영 되버린다면 데이터는 엉망이 된다.

그래서 필요한 것이 Transaction의 개념이다

한마디로 All 또는 Nothin → 다 되거나 하나도 안되거나 | .

위에서 가정한 동작이 모두 성공한다면 commit으로 반영 확정 실패하면 Rollback으로 되돌리기를 처리 해줘야 한다.

그럼 위 상황은 롤백해서 City 테이블을 다시 살려야한다..

그럼 여기서 Data Access Layer는 개별 SQL들을 처리하면 끝나버리기 때문에 요청이나 업무에 엮인 다른 SQL들을 살펴볼 수가 없다.

따라서 여러 동작을 하나로 묶어서 관리할 개념이 필요했고

이게 바로 Service Layer

왜 나눠서 사용하는 지 알게 되었고 앞으로 사용할 다양한 요청에 따라 응용이 가능할 것 같다.

스프링 이전의 Service Layer의 문제점

→ Data Access Layer가 가진 문제점과 동일하다.

```
public void delete(String code){
  Connection con = DBUtil.getUtil().getConnection();

  try{
    con.setAutoCommit(false);
    // 실제 비지니스 로직은 단 두줄
    cityRepo.delete(con, code);
    countryRepo.delete(con, code);

    con.commit();
  }catch(Exception e){
    DBUtil.getUtil().rollback(con);
  }finally{
    DBUtil.getUtil().close(con);
  }
}
```

위 예에서 보듯 실제 비지니스 로직은 단 두 줄이고 나머지는 필요에 따라 추가된 횡단 관심사들이다. 스프링에서는 AOP를 이용해서 횡단 관심사 코드를 제거하고 비지니스 로직만으로 Service Layer를 구성할 수 있게 한다.

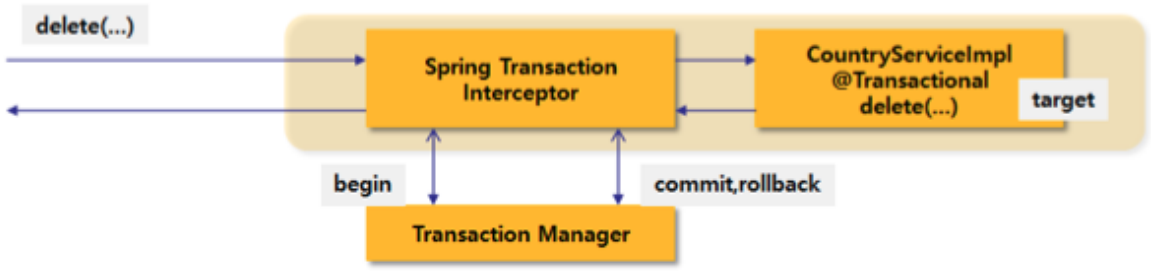
@Transactional

- 여기서 트랜젝션 처리를 한다는 애너테이션
- 클래스 레벨과 메서드 레벨에 선언할 수 있다.

- @Service의 클래스 레벨레 선언하면 서비스에 포함된 모든 메서드들이 트랙젝션 하에서 동작하게 되고 메서드 레벨에서 선언하면 해당 메서드만 트랙젝션 하에서 동작한다.
- 단순 조회 과정은 트렌젝션 하에서 처리할 필요는 없음.

- 이 애너테이션이 사용되면 스프링은 해당 @Service 클래스에 대한 proxy 객체를 생성하고 Around advice를 구성한다.
- proxy 객체가 하는 일은 다음과 같다.
  - 타겟 메서드 호출 이전에 트랜젝션 시작
  - 타겟 메서드 정상 종료후 commit 실행
  - 타겟 메서드에서 RuntimeException 발생 시 rollback 실행

즉 스프링 이전의 서비스 코드에서 비지니스 로직을 제외하는 부분은 다 AOP에서 처리하는 구조다.



> 앞서 Data Access Layer의 단위 테스트에서도 @Transaction을 사용했는데 **단위 테스트에서 사용된 @Transaction은 언제나 rollback**을 유발시키는 차이점이 있다.

@Transcational 적용

예) city에 대한 delete 기능 처리

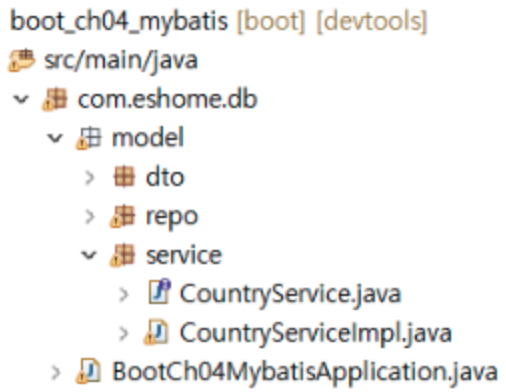
```
<delete id="delete" parameterType="string">
  delete from city where countrycode=#{countryCode}
```

```
</delete>
```

mapper에는 CountryCode를 전달받아 해당 countryCode에 속한 city들을 삭제할 수 있게 한다.

```
@Mapper
public interface CityRepo {
  int delete(String countryCode);
}
```

Service Layer



CountryService에서는 필요한 메서드를 선언한다.

```
package com.eshome.db.model.service;

public interface CountryService {
    int delete(String code);
}
```

```java
package com.eshome.db.model.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.eshome.db.model.repo.CityRepo;
import com.eshome.db.model.repo.CountryRepo;

@Service
public class CountryServiceImpl implements CountryService {

    @Autowired
    CityRepo ciRepo;

    @Autowired
    CountryRepo coRepo;

    @Override
    public int delete(String code) {
        // 먼저 도시 삭제 후
        ciRepo.delete(code);
        // 국가 삭제
        return coRepo.delete(code);
    }
}
```

아직은 트랜젝션과 관련된 코드를 작성하지 않았다.

클래스 선언부에 @Service를 선언해서 빈으로 등록될 수 있게 한다.

delete를 재정의 할 때는 CityRepo와 CountryRepo의 delete를 사용하는데 이를 위해 CityRepo와 CountryRepo를 주입 받아서 사용한다.

```java
@Override
public int delete(String code) {
    // 먼저 도시 삭제 후
    ciRepo.delete(code);
    // 문제 발생
    int i = 1 / 0;
    // 국가 삭제
    return coRepo.delete(code);
}
```

이런식으로하면 ArithmeticException이 발생하는데 Country 정보를 삭제하기 전에 문제가 발생한다.

```java
@Override
@Transactional
public int delete(String code) {
    // 먼저 도시 삭제 후
    ciRepo.delete(code);
    int i = 1 / 0;
    // 국가 삭제
    return coRepo.delete(code);
}
```

이런 식으로 하면 에러가 발생했을때 City의 정보가 삭제가 된것을 롤백할 수 있다.

애너테이션하나로 DML을 사용할 때에 번거로움을 덜어준다.