220501 우다다다 정리하는 레이어드 아키텍처

우다다다 정리하기

우다다다

레이어드 아키텍처

객체가 역할과 책임을 구분하여 서로 협력하도록 구성하듯이

계층별로 관심사를 분리하여 서로 협력하도록 구성한다.

컨텍스트마다 다르게 사용되는 경우가 잦으나, 주로 Eric Evans 2003 DDD에서 내린 정의를 인용한다.

레이어의 구성과 역할

프레젠테이션 레이어 -> 애플리케이션 레이어 -> 도메인 레이어 -> 인프라스트럭처 레이어라는 표현이 주로 사용된다.

프레젠테이션 레이어는 컨트롤러에 해당하고, 라우팅 및 요청을 전달하고 결과를 반환하는 역할을 한다.

애플리케이션 레이어는 서비스에 해당하고, 비즈니스 로직이 담긴다. 요청을 처리하여 결과를 생성해 반환한다.

도메인 레이어는 도메인 객체들을 의미한다. 체스로 예를 들면 Board나 Piece와 같은 존재들이다. 서비스 레이어에서 이들에게 메시지를 보낸다.

인프라스트럭처 레이어는 데이터의 영속성을 위해 존재한다. 데이터베이스에 가장 가까이 위치한다. Repository와 DAO가 이에 속한다.

DAO 와 Repository

Repository는 영속 자료를 컬렉션 다루듯 다루기 위한 개념에서 출발한다.

가령 StudentRepository 라고 하면, 학생을 저장하거나 조회할 수 있는 학생 저장소가 된다.

이러한 개념이 필요한 이유는 DDD가 도메인 전문가와 개발자가 함께 일할 때 도메인 모델을 이용해 보편언어로 소통한다는 개념이기 때문이다.

비 개발자더라도 학생 저장소 라고 하면 이해할 수 있기 때문이다.

추가적으로 컬렉션처럼 영속 자료를 다루게 되면 더욱 객체지향적인 설계가 가능해지고, 서비스 레이어를 얇게 유지할 수 있게 된다.

DAO는 Repository보다 더 데이터베이스에 가까이 위치한다.

Repository는 DAO를 의존하며, DAO로부터 반환된 Entity를 이용해 도메인 객체를 만들어 서비스에게 반환한다.

때론 여러 DAO를 의존하여 여러 소스로부터 조회된 결과를 조립하여 도메인 객체를 만들어 반환하기도 한다.

즉, 데이터베이스를 조회한 뒤 이를 테이블 레벨의 데이터 즉 Entity로 반환하는 역할을 DAO가 하고,

Repository는 이러한 DAO를 의존하여 반환된 Entity중 필요한 값을 이용해 도메인 객체를 생성해 서비스에게 반환한다.

이러한 역할 분리를 통해 DAO는 도메인과의 의존 없이 Entity만 반환하도록 구성되며,

Repository는 DAO를 의존하여 Entity를 도메인으로 만드는 작업을 하여 서비스 레이어가 얇게 유지되도록 돕는다.

Entity는 무엇이고 왜 DAO는 Entity를 반환하는가

포비 펀치에서 배웠듯, 필요한 컬럼마다 메서드를 생성할 경우 테스트 코드 유지보수 비용이 급증한다.

따라서 DAO는 테이블의 컬럼을 그대로 가진 Entity 레벨로 조회한 값을 반환하도록 구성된다.

질의한 테이블의 전체 컬럼 데이터인 Entity를 반환한 뒤, 필요한 값을 꺼내는 등의 작업은 Repository에게 넘기는 것이다.

이렇게 해야 DAO가 변경에서 비교적 자유로워지고 테스트코드의 유지보수 비용이 절감된다.

그렇지 않으면 다른 컬럼이 필요해질 때마다 DAO가 비대해질 것이다.

즉 Entity는 도메인 객체는 아니며, 비즈니스 로직을 갖지 않으며, DAO가 Repository에게 테이블의 데이터를 반환하는데 사용되는 객체이다.

Entity는 테이블의 데이터를 의미하며 DAO에서 Repository로 이동하기만 하며 다른 방향이나 다른 레이어에서 사용되지 않는다는 점에서 DTO와 구분된다.

이러한 Entity는 Repository에서 도메인 객체를 만드는 데 사용된다.

응답 DTO는 누가 만드나?

컨트롤러 역시 프레젠테이션 레이어이다.

따라서 불필요한 데이터가 거기까지 전달되는 것을 권장하기 어렵다.

응답 DTO는 서비스 레이어에서 생성해 컨트롤러에게 전달하는 것이 적절하다 판단된다.

DAO와 Repository는 데이터의 영속성에 관련된 역할을 담당하는 곳이므로 응답 DTO에 대한 역할을 담당하는 것은 적절치 않다.

시나리오 적용

1. 컨트롤러로 id, password가 전달되며 게임방 삭제 요청이 도착했다. 서비스레이어로 이들을 전달한다.
2. 서비스 레이어는 Repository에게 아이디를 전달하며 게임방 조회를 요청한다.

마치 Room 도메인 객체를 컬렉션으로 감싼 일급컬렉션 Rooms 에게 findById(long id)로 방을 요청하듯이 메시지를 보낸다.

1. Repository는 DAO에게 id를 건네며 RoomEntity를 요청한다.
2. DAO가 select * from game where id = ? 를 통해 조회된 데이터를 RoomEntity로 매핑해 Repository로 반환한다.
3. Repository에서 RoomEntity중 일부 값을 꺼내 Room 도메인 객체를 생성해서 서비스에게 반환한다.
4. 서비스는 Repository로부터 반환된 도메인 객체 Room에게 password를 전달하며 이게 일치하는지 메시지를 보낸다.
5. 일치한다는 응답을 받았을 경우 Repository에게 id를 전달하며 삭제 요청을 한다. 이를 DAO가 다시 전달받아 삭제한다.
6. 정상 삭제됐을경우 서비스가 응답 DTO를 생성해 컨트롤러에게 반환한다.

그래서 Room 도메인 객체 추가해야함?

이번 미션에선 레이어드 아키텍처의 필요성과 각각의 역할에 대한 기본적인 이해 정도에서 마쳐도 되지 않을까 싶다.

이해한 내용을 모두 적용하기 위해 다시 DAO를 DAO와 Repository로 쪼갠다거나 Room 도메인 객체를 만들거나 하는 부분은 선택적이라고 생각한다.

시간적여유가 있었다면 도전해봄직 하나, 이러한 내용에 대해 이해를 한 뒤, 다음 미션에서 도전하는 게 시기적으로 맞다고 생각한다.

요구사항 추가에 의한 도메인 객체 추가의 판단 기준은?

체스를 java-chess의 콘솔버전부터 시작했기 때문에 응집도 라는 엄청난 이점을 얻을 수 있었다.

처음 DB를 연동했을 땐 매번 조회된 데이터를 Board 도메인 객체로 변환하는 과정이 대단히 비효율적으로 느껴졌으나,

컬렉션 처럼 영속 데이터를 다루는 것이 객체지향 관점에서 대단히 큰 장점이라는 것을 레이어드 아키텍처를 이해하며 배울 수 있었다.

처음부터 DB를 연결했다면 과연 지금과 같은 응집도가 있었을까?

이를 뒤집어 생각한다면, 콘솔 버전으로 요구사항을 구현한다고 했을때,

해당 요구사항 만족을 위한 도메인 객체가 추가되었을 것 같다고 판단된다면, 도메인 객체 추가가 맞는 방향이라고 본다.

즉, 콘솔 체스에서도 방을 만들고 방 목록을 보여주고, 비밀번호 검증을 통해 삭제할 수 있어야 했다면,

그때는 확실히 Room이라는 도메인 객체를 만들었고, Rooms 등의 Room을 관리하는 객체를 만들었을거라 예상할 수 있다.

그렇게 판단된다면 추가하는 게 맞다고 본다.