220502 네오 펀치 - Room, PasswordEncoder

도메인은 뭔가

- 해결하고자 하는 문제 영역

Room은 도메인인가

- 응답값을 누가 어떻게 JSON으로 파싱해주는지 우리는 관심 없다
- 데이터가 어떤 Database에 어떻게 저장되는지 우리는 관심 없다
- 그러나 게임방의 관리, 비밀번호 암호화 저장, 비밀번호 검증 후 삭제 는 우리의 관심사다
- 그렇다면 Room은 도메인이고, 도메인 객체를 추가할 가치가 있다.

Room이 비밀번호 일치 여부에 대한 확인 책임을 갖는 것이 맞는가

- Room에게 메시지를 보내서 확인하는 구조가 객체지향적이다.
- 따라서 메시지를 Room이 받는 것이 맞다.
- 그렇게 하지 않을 경우, getter로 비밀번호값을 꺼내서 서비스 레이어에서 검증해야 한다.
- 이건 객체지향적이지 않다.

그러면 Room이 PasswordEncoder를 가져야 하는가?

- 그렇다. Room이 비밀번호 일치여부 확인 책임을 갖는 순간 그렇게 해야만 한다.
- 가진다 라는 의미 보다는, 의존한다 또는 협력한다 라고 표현하는 것이 객체지향사고에 도움이 되는 것 같다.

Room이 PasswordEncoder를 의존하는 방법 두 가지

- 필드로 갖는 방법에 대하여
    - Room이 PasswordEncoder를 필드로 가지려면 Spring의 DI를 활용해야 하므로
      Room을 Spring Bean으로 선언해야 하는데, 싱글턴으로 관리할 대상이 아닌 도메인 객체이므로
      이 방법은 제외한다.

- 매번 매개변수로 함께 전달하는 방법에 대하여
    - 필드로 직접 갖지는 않고, 비밀번호 일치여부를 물어볼 때마다 PasswordEncoder 구현체를 매개변수로 함께 전달하도록 구성해보면 어떨까?
    - 그러면 Spring DI가 도메인 객체와 어울리지 않는다는 이슈를 피해갈 수 있으면서 동시에 Room에게 메시지를 보내 비밀번호 일치여부를 확인할 수 있다.

매개변수로 협력할 다른 객체 전달하기

- 메서드 시그니처가 public boolean isMatchedPassword(String password, PasswordEncoder encoder) 와 같이 나오는데, 매 요청마다 인코더를 보내줘야 한다는 것이 처음엔 부자연스럽게 느껴졌다.
- 그러나 자동차 미션을 떠올려보면, 랜덤 움직임에 대한 의존성을 낮추기 위해 Car 객체가 자신의 이동 책임 수행 과정에서 MoveStrategy 인터페이스를 의존한 바 있다. 당시 나의 코드는 Car의 생성자에서 이동전략을 전달해서 필드로 갖고 있었지만, 이는 요구사항이 단순했기 때문이다.
- 그래서 메서드 시그니처가 public boolean move() 와 같이 단순했다. 만약 이동시마다 전략이 달라져야 했다면, setter주입으로 매번 전략을 바꾸기보다는 매번 move메서드에서 이동 전략 구현체를 전달받는 식으로 구성했을 것이다.
- 추가로, 앞으로 이런식으로 매번 다른 인터페이스나 다른 협력객체를 매개변수로 전달하는 구조를 사용할 일이 많아질 것이다. 요구사항이 복잡해질수록 객체간 의존과 협력이 더욱 적극적으로 활용되기 때문이다.
- 도메인 객체가 어떤 책임을 갖는다는 것의 의미는, 그 책임을 수행하기 위한 충분한 정보와 행위를 가지고 있다는 것과 같다. 다시말해, 자신의 책임을 무조건 혼자할 필요는 없다. 그 과정에서 다른 객체와 협력해도 된다. 그게 객체지향이다.
- 결론적으로 Room이 PasswordEncoder를 의존해서 비밀번호 일치여부 확인이라는 메시지를 가질 수 있게 구성하되, 필드로 가지지는 않고 매개변수로 매번 받도록 구성한다.
- Service에서는 DI로 필드로 갖고 있기에 매번 전달해도 같은 객체를 재사용할 수 있다.

결론

- Room은 비밀번호 일치여부 검증 책임을 갖는다.
- 검증 책임을 수행하는 과정에서 PasswordEncoder 인터페이스를 의존한다.
- PasswordEncoder는 우리가 직접 만들지 않았고, Framework가 제공하는 것을 사용한다.
- 따라서 Service 레이어에서 DI로 주입받은 객체를 재사용할 수 있다.
- 매개변수로 매번 전달하여 사용한다.
- 모든 문제가 100% 만족스럽게 해결된 느낌이 아닐 수 있다.
- 그러나 Framework가 제공하는 장점, 객체지향의 장점, 로직의 단순화, 코드 작성의 편리함 등 여러 가지 요소의 Trade off를 종합 고려하여 최선을 고안해내 선택할 줄 알아야 한다.

```
public final class Room {

    private final long id;
    private final String name;
    private final String password;

    // constructor

    public boolean matchPassword(String password, PasswordEncoder, passwordEncoder) {
        final String encoded = passwordEncoder.encode(password);
        return passwordEncoder.matches(encoded, this.password);
    }
}
```

```
public class SpringChessService {

    private final ChessRepository chessRepository;
    private final PasswordEncoder passwordEncoder; // BCryptPasswordEncoder 구현체 사용

    // constructor

    public RoomDeleteResponse deleteById(long id, RoomDeleteRequest roomDeleteRequest) {
        final Room room = chessRepository.findById(id);
        final boolean matched = room.matchPassword(roomDeleteRequest.getPassword(), passwordEncoder);

        if (matched) {
            return new RoomDeleteResponse(chessRepository.deleteById(id);
        }
        throw new PasswordNotMatchedException();
    }
}
```

사고의 확장 - 규칙과 편의성의 Trade Off

- 지금까지 객체지향 생활체조, 객체지향 프로그래밍 등 기본이 되는 규칙들에 대해 훈련해왔다.
- Spring 이라는 훌륭한 Framework를 배우고 사용하는 과정에서, 기존에 배웠던 규칙과 충돌되는 경험을 앞으로도 자주 하게 될 예정이다.
- final 키워드를 사용하지 못하게 되거나, 기본생성자를 열어야 하거나, getter를 열어야 하거나, 내부적으로 Reflection을 사용하는 등, 기존에 훈련해온 규칙과 충돌되는 경험들이 지금도 존재한다.
- 견고히 규칙을 학습하되, 많은 사람들이 오랜 세월에 걸쳐 쌓아올린 이 Java와 Spring이라는 생태계가 왜 그러한 선택을 하게 됐는지도 고민해볼 법 하고, 이미 잘 만들어져있는 Framework의 장점을 충분히 누리기 위해 장단점을 따져가며 스스로 선택할 줄도 알아야 한다.