6장. 영속성 어댑터 구현하기

💡 데이터베이스 주도 설계로 이루어지는 의존성을 역전시키기 위해 영속성 계층을 애플리케이션 계층의 플러그인으로 만들어보자 !

영속성 어댑터

`주도되는` / `아웃고잉` 어댑터

🔌 포트

- `애플리케이션 서비스` - `영속성 코드` 간 간접적인 계층
- 영속성 계층에 대한 코드 의존성을 없애기 위해 사용

영속성 어댑터의 책임

1. 입력 받음
2. 입력을 데이터베이스 포맷으로 매핑
3. 입력을 데이터베이스로 전송
4. 데이터베이스 출력을 애플리케이션 포맷으로 매핑
5. 출력 반환

> 핵심은 ...
> 영속성 어댑터의 입력 모델이 영속성 어댑터 내부에 있는 것이 아니라
>
> **애플리케이션 코어**에 있기 때문에
>
> **영속성 어댑터 내부를 변경하는 것이 코어에 영향을 미치지 않는다**는 것!

포트 인터페이스 나누기

☝️ 특정 엔티티가 필요로 하는 모든 연산을 하나의 인터페이스에 넣기

- 모든 서비스가 실제로 필요하지 않은 메서드에 의존하게 됨
- 코드 이해와 테스트가 어려워짐

→ `인터페이스 분리 원칙`을 적용하여 특화된 인터페이스를 만들자.

✌️ 인터페이스 분리 원칙을 적용하여 특화된 인터페이스 만들기

- 각 서비스는 실제로 필요한 메서드에만 의존
- 포트의 이름이 포트에 역할을 잘 표현
- 테스트에서 mocking이 쉬워짐

영속성 어댑터 나누기

- 영속성 연산이 필요한 도메인 클래스(애그리거트) 하나 당 하나의 영속성 어댑터를 구현할 수 있다.
- 영속성 어댑터들은 각 영속성 기능을 이용하는 도메인 경계를 따라 자동으로 나눠진다.
- 도메인 코드는 영속성 포트에 의해 정의된 명세를 어떤 클래스가 충족시키는지 관심 없다.
- `애그리거트 : 영속성 어댑터 = 1 : 1` 은 여러 개의 바운디드 컨텍스트의 영속성 요구사항을 분리하기 위한 좋은 토대가 된다.
- 각 바운디드 컨텍스트는 영속성 어댑터를 하나씩 가지고 있다.
- 어떤 맥락이 다른 맥락에 있는 무엇인가를 필요로 한다면 전용 인커밍 포트를 통해 접근해야 한다.

스프링 데이터 JPA 예제

⌨️ Account

```
package buckpal.domain;

@AllArgsConstructor(access = AccessLevel.PRIVATE)
public class Account {

    @Getter private final AccountId id;
    @Getter private final ActivityWindow activityWindow;
    private final Money baselineBalance;

    public static Account withoutId(Money baselineBalance,

        return new Account(null, baselineBalance, activityWin
    }

    public static Account withId(AccountId accountId,
                                                          M
                                                          A
```

```
        return new Account(accountId, baselineBalance, activi
    }

    public Money calculateBalance() {
        ...
    }

    public boolean withdraw(Money money, AccountId targetAcco
        ...
    }

    public boolean deposit(Money money, AccountId sourceAccou
        ...
    }
```

- 최대한 불변성을 유지한다.
- 유효한 상태의 Account 엔티티만 생성할 수 있는 팩토리 메서드를 제공한다.
- 모든 상태 변경 메서드에서 유효성 검증을 수행한다.

⌨️ AccountJpaEntity

```
package buckpal.adapter.persistence;

@Entity
@Table(name = "account")
@Data
@AllArgsConstructor
@NoArgsConstructor
class AccountJpaEntity {

    @Id
    @GeneratedValue
    private Long id;
}
```

⌨️ ActivityJpaEntity

```
package buckpal.adapter.persistence;

@Entity
@Table(name = "activity")
@Data
@AllArgsConstructor
@NoArgsConstructor
class ActivityJpaEntity {

    @Id
    @GeneratedValue
    private Long id;

    @Column private LocalDateTime timestamp;
    @Column private Long ownerAccountId;
    @Column private Long sourceAccountId;
    @Column private Long targetAccountId;
    @Column private Long amount;
}
```

- 특정 계좌의 모든 활동을 들고 있는 엔티티

⌨️ ActivityRepository

```
interface ActivityRepository extends JpaRepository<ActivityJp
    @Query("select a from ActivityJpaENtity a " +
        "where a.ownerAccountId = :ownerAccountId " +
        "and a.timestamp >= :since")
    List<ActivityJpaEntity> findByOnwerSince(
        @Param("ownerAccountId") Long ownerAccountId,
        @Param("since") LocalDateTime since);

    @Query("select sum(a.amount) from ActivityJpaEntity a " +
        "where a.targetAccountId = :accountId " +
        "and a.ownerAccountId = :accountId " +
        "and a.timestamp < :until")
    Long getDepositBalanceUntil(
```

```
        @Param("accountId") Long accountId,
        @Param("until") LocalDateTime until);

    @Query("select sum(a.amount) from ActivityJpaEntity a " +
        "where a.sourceAccountId = :accountId " +
        "and a.ownerAccountId :=accountId " +
        "and a.timestamp < :until")
    Long getWithdrawalBalanceUntil(
        @Param("accountId") Long accountId,
        @Param("until") LocalDateTIme until);
}
```

⌨️ AccountPersistenceAdapter

```
@RequiredArgsConstructor
@Component
class AccountPersistenceAdapter implements
    LoadAccountPort,
    UpdateAccountStatePort {

    private final AccountRepository accountRepository;
    private final ActivityRepository activityRepository;
    private final AccountMapper accountMapper;

    @Override
    public Acccount loadAccount(AccountId accountId, LocalDat
        AccountJpaEntity account = accountRepository.findById


        List<ActivityJpaEntity> activities = activityReposito



        Long withdrawalBalance = orZero(activityRepository
                                        .getWithdrawalBalanceUnti
```

```
        Long withdrawalBalance = orZero(activityRepository
                                        .getDepositBalanceUntil(a


        return accountMapper.mapToDomainEntity(account,



    }

    private Long orZero(Long value) {
        return value == null ? 0L : value;
    }

    @Override
    public void updateActivities(Account account) {
        for (Activity activity : account.getActivityWindow().
            if (activity.getId() == null) {
                activityRepository.save(accountMapper.mapToJp
            }
        }
    }
```

- 영속성 어댑터
- `매핑하지 않기` 전략도 유효하다 !
  - JPA 엔티티는 기본 생성자를 필요로 한다.
  - 영속성 계층에서는 성능 측면에서 `@ManyToOne` 관계 설정이 적절할 수 있지만, 예제에서는 항상 일부 데이터만 가져오기를 바란다.
  - **영속성 측면과의 타협 없이 풍부한 도메인 모델을 생성하고 싶다면 도메인 모델과 영속성 모델을 매핑하자.**

데이터베이스 트랜잭션

트랜잭션 경계는 어디에 위치시켜야 할까?

- 트랜잭션은 하나의 유스케이스에 대해 일어나는 모든 쓰기 작업에 대해 걸쳐 있어야 한다.
- 영속성 어댑터는 어떤 연산이 같은 유스케이스에 포함되는지 알지 못한다.
- 영속성 어댑터 호출을 관장하는 서비스에 위임해야 한다.
- `@Transactional` 어노테이션을 애플리케이션 서비스 클래스에 붙이자.
  - 모든 public 메서드를 트랜잭션으로 감싼다.
- 어노테이션 없이 깔끔하게 유지되기를 바란다면 AspectJ 등으로 트랜잭션 경계를 코드에 위빙할 수 있다.

결론

- 도메인 코드에 플러그인처럼 동작하는 영속성 어댑터를 만들자.
  - 도메인 코드가 영속성과 분리되어 풍부한 도메인 모델을 만들 수 있다.
- 좁은 포트 인터페이스를 사용하면 포트마다 다른 방식으로 구현할 수 있는 유연함이 생긴다.
- 포트 뒤에서 애플리케이션이 모르게 다른 영속성 기술을 사용할 수 있다.
- 포트의 명세만 지켜지면 영속성 계층 전체를 교체할 수 있다.