07

트랜잭션

01. 트랜잭션

- 데이터의 일관성과 무결성을 유지하기 위해 사용
- @Transactional 사용
  - 클래스 수준
    - 클래스에 @Transactional을 추가하면 해당 클래스의 모든 메서드가 트랜잭션에서 실행
  - 메서드 수준
    - 특정 메서드에서만 트랜잭션을 적용하려면 메서드에 @Transactional을 추가

```
@Service
@Transactional
public class OrderService {
    @Transactional
    public void placeOrder(Order order) {
        //트랜잭션 내에서 실행
        }
}
```

02. 트랜잭션 전파(Propagation) 설정

- @Transactional의 propagation 속성으로 제어
- 전파 속성 종류
  - REQUIRED (기본값): 이미 트랜잭션이 존재하면 해당 트랜잭션에 참여하고, 없으면 새 트랜잭션을 생성.
  - REQUIRES_NEW: 항상 새 트랜잭션을 생성, 기존 트랜잭션은 일시 중단.
  - MANDATORY: 트랜잭션이 반드시 존재해야 하며, 없으면 예외를 발생.
  - SUPPORTS: 트랜잭션이 있으면 참여하고, 없으면 트랜잭션 없이 실행.
  - NOT_SUPPORTED: 항상 트랜잭션 없이 실행.
  - NEVER: 트랜잭션 없이 실행되며, 트랜잭션이 존재하면 예외를 발생.
  - NESTED: 기존 트랜잭션 내에서 중첩된 트랜잭션을 생성

02. 트랜잭션 전파(Propagation) 설정

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void saveOrder(Order order) {
    // 항상 새 트랜잭션에서 실행
}
```

03. 트랜잭션 격리 수준(Isolation Level)

- 트랜잭션 간의 간섭을 제어
- 격리 수준 종류
  - DEFAULT: 데이터베이스의 기본 격리 수준 사용
    - Mysql : REPEATABLE_READ
  - READ_UNCOMMITTED: 다른 트랜잭션이 커밋하지 않은 데이터를 읽을 수 있음
  - READ_COMMITTED: 커밋된 데이터만 읽을 수 있음
  - REPEATABLE_READ: 동일 트랜잭션에서 같은 데이터를 읽으면 항상 동일한 결과 반환
  - SERIALIZABLE: 완전한 격리, 가장 높은 일관성 보장

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
public void processOrder() {
    // REPEATABLE_READ 격리 수준에서 실행
}
```

04. 트랜잭션 롤백 설정

- Spring은 기본적으로 트랜잭션 내에서 발생한 RuntimeException 또는 Error에 대해 롤백
- 특정 예외에 대해 롤백을 설정할 수 있음

```
@Transactional(rollbackFor = Exception.class)  // CheckedException도 롤백
public void processOrder() throws Exception {
    // 비즈니스 로직
}
```

- 롤백 제외

```
@Transactional(noRollbackFor = CustomException.class)  // CustomException 발생 시 롤백하지 않음
public void saveOrder(Order order) {
    // 비즈니스 로직
}
```

05. 트랜잭션 읽기 전용 설정

- 읽기 작업에만 사용하는 메서드에서는 readOnly = true를 설정해 성능 최적화

```
@Transactional(readOnly = true)
public List<Order> findAllOrders() {
    return orderRepository.findAll();
}
```

06. 주의사항

동일 클래스의 메서드 간 호출 시 트랜잭션이 @Transactional 설정 적용 안됨

해결방법

- 별도의 서비스 클래스를 분리
- self-injection을 사용

```
@Service
public class OrderService {

    @Transactional
    public void placeOrder() {
        // 트랜잭션 시작
            validateOrder(); // 프록시를 거치지 않고 직접 호출
            // 트랜잭션 종료
    }
    @Transactional (…..)
    public void validateOrder() {
        // 이 메서드에 @Transactional이 있어도 트랜잭션이 적용되지 않음
            System.out.println("Validating order...");
    }
}
```

06. 주의사항

```
@Service
public class OrderService {

    @Autowired
    private ApplicationContext applicationContext;

    @Transactional
    public void placeOrder() {
        //트랜잭션이 적용됨
            applicationContext.getBean(OrderService.class).validateOrder();
    }

    @Transactional(…..)
    public void validateOrder() {
        //트랜잭션이 적용됨
            System.out.println("Validating order...");
    }
}
```

07. 메소드보다 작은 단위의 트랜잭션 적용

TransactionTemplate 사용

```
@Service
public class MyService {
    private final TransactionTemplate transactionTemplate;

    public MyService(TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }
    public void process() {
        //트랜잭션 블록 시작
            transactionTemplate.execute(status -> {
            //트랜잭션 안에서 실행되는 로직
                    saveData1();
            return null;
        });
        //트랜잭션이 없는 작업
            nonTransactionalOperation();
        //또 다른 트랜잭션 블록
            transactionTemplate.execute(status -> {
            //다른 트랜잭션에서 실행되는 로직
                    saveData2();
            return null;
        });
    }
```

07. 메소드보다 작은 단위의 트랜잭션 적용

```
private void saveData1() {
        // 데이터베이스 저장 로직 1
    }

    private void saveData2() {
        // 데이터베이스 저장 로직 2
    }

    private void nonTransactionalOperation() {
        // 트랜잭션이 필요 없는 작업
        }
}
```