chapter 00

Spring Transaction

1. **Transaction**
2. Spring | Transaction
3. Spring | Transaction Setting
4. Spring | How to use transaction

01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) 이란

트랜잭션은 데이터베이스의 상태를 변환시키는 쪼갤 수 없는 업무 처리의 최소 단위를 말합니다.

트랜잭션이란 개념은 작업의 완전성(integrity)을 보장해주는데, 일련의 논리적인 작업들을 모두 완벽하게 처리하거나 그렇지 못할 경우에는 작업 이전의 상태로 복구하여 오류(일부 쿼리만 수행된 상태)를 만들어 내지 않도록 해주는 기능입니다.

예를 들어, A라는 사람이 B라는 사람에게 1,000원을 지급하고 B가 그 돈을 받은 경우, 이 거래 기록은 더 이상 작게 쪼갤 수가 없는 하나의 트랜잭션을 구성합니다. 만약 A는 돈을 지불했으나 B는 돈을 받지 못했다면 그 거래는 성립되지 않아야 합니다. 이처럼 A가 돈을 지불하는 행위와 B가 돈을 받는 행위는 별개로 분리될 수 없으며 하나의 거래내역으로 처리되어야 하는 단일 거래입니다. 이런 거래의 최소 단위를 트랜잭션이라고 합니다. 트랜잭션 처리가 정상적으로 완료된 경우 커밋(commit)을 하고, 오류가 발생할 경우 원래 상태대로 롤백(rollback)을 합니다.

01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) 의 조건

트랜잭션이 안전하게 수행되기 위해서는 ACID 조건을 충족해야합니다. ACID란

- Atomicity(원자성) : 하나의 트랜잭션이 더 이상 작게 쪼갤 수 없는 최소한의 업무 단위입니다.
- Consistency(일관성) : 트랜잭션이 완료된 결과값이 일관적인 DB 상태를 유지해야합니다.
- Isolation(격리성) : 하나의 트랜잭션 수행 시 다른 트랜잭션의 작업이 끼어들지 못하도록 보장해야 합니다.
- Durability(지속성) : 트랜잭션이 정상적으로 종료된 다음에는 영구적으로 데이터베이스에 작업 결과가 저장되어야 합니다.

의 약자로서, 데이터베이스의 트랜잭션이 안전하게 수행되기 위한 4가지 필수적인 성질을 말합니다.

01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) 의 특징

트랜잭션 기능의 대표적인 이점 중 하나는 **무정지성의 향상**입니다.

운영체제 장애와 같은 서버 장애가 발생하여 그로부터 데이터베이스를 재가동한 때에 장애 직전까지의 커밋결과를 손실하지 않고 마치는 것이 가능합니다. 트랜잭션을 지원하지 않는 데이터베이스의 경우 OS 장애나 프로세스의 비정상적 종료에도 데이터가 손상될 수 있습니다.

01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) 의 **Locking**

로킹 기법이란 하나의 트랜잭션이 데이터를 액세스하는 동안 다른 트랜잭션이 그데이터 항목을 액세스할 수 없도록 하는 병행 제어 기법을 말합니다. 잠금 기법에서 데이터베이스에서의 로킹(Locking) 단위란 말 그대로 "잠그는 단위"를 말 합니다. 즉, 한번에 한명만 사용할 수 있게 하는 단위를 로킹 단위, 사용하는 데이터 객체를 로크 라고 합니다.

로킹 단위	로크의 수	병행 제어	로킹 오버헤드	병행성 수준
커짐	적어짐	단순해짐	감소	낮아짐
작아짐	많아짐	복잡해짐	증가	높아짐

잠금은 동시성 제어를 위한 기능이고, 트랜잭션은 데이터의 정합성을 보장하기 위한 기능입니다. 트랜잭션의 조건 중, 격리성을 보장하게 하는 것이 이 로킹기법입니다. 데이터의 정합성을 위해 동시성 제어를 한다라고 이해할 수 있습니다.

잠금 기법에서 트랜잭션들은 어떤 데이터를 접근하기 전에 잠금을 요청하여 잠금을 허락받아야 하며, 데이터를 갱신할 때는 반드시 잠금(Lock) -> 실행(Execute) -> 해제(Unlock)의 규칙을 따라야 합니다.

01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) 의 Locking 에 의한 문제점

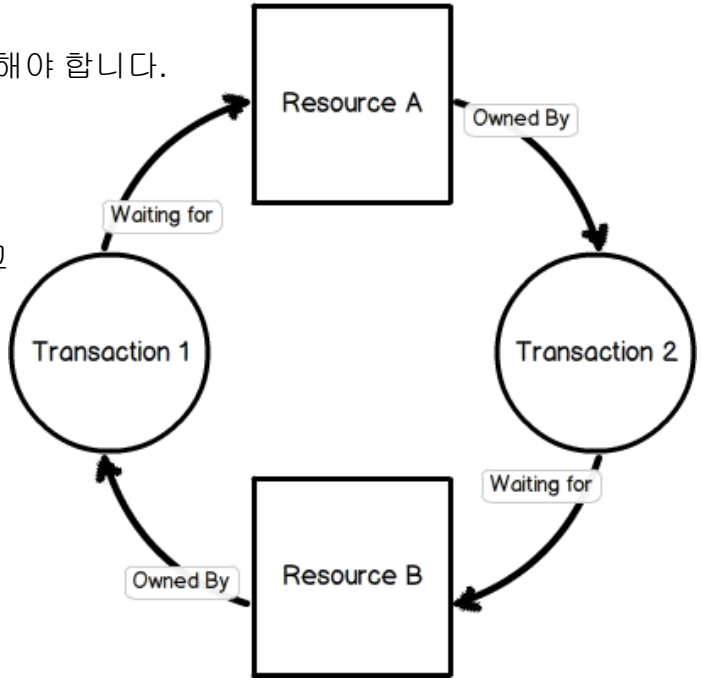
블로킹(Blocking)

블로킹은 로킹으로 인해 특정 세션이 작업을 진행하지 못하는 상태를 말합니다. 로킹 중인 트랜잭션이 commit 또는 rollback 됨으로써 해제될 수 있습니다.

잦은 블로킹으로 인한 호율 하락을 막기 위해서는 트랜잭션을 최대한 짧게 설계해야 합니다.

교착상태(Deadlock)

교착 상태란 어떤 한 트랜잭션이 사용하기 위해 잠구어 놓은 자원을 사용하기 위해 기다리므로 모든 트랜잭션들이 실행을 전혀 진행시키지 못하고 무한대기상태를 말합니다.

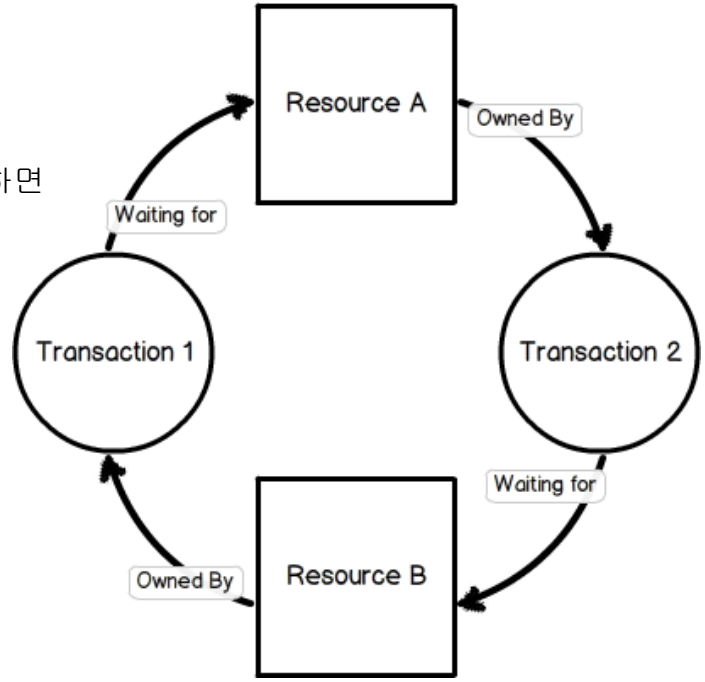


01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction) DeadLock 발생 빈도를 낮추는 방법

- 트랜잭션을 자주 커밋 하고
- 정해진 순서로 테이블에 접근 하도록 한다
- 트랜잭션 1 이 자원 B -> A 의 순으로 접근했고, 트랜잭션 2 는 자원 A -> B의 순으로 접근했다 (문제발생)
- 트랜잭션들이 동일한 자원 순으로 접근하게 한다
- 읽기 잠금 획득 (SELECT ~ FOR UPDATE)의 사용을 피한다
- 한 테이블의 복수 행을 복수의 연결에서 순서 없이 갱신하면 교착상태가 발생하기 쉽다, 이 경우에는 테이블 단위의 잠금을 획득해 갱신을 직렬화 하면 동시성을 떨어지지만 교착상태를 회피할 수 있다.



01 Transaction

■ Transaction(트랜잭션)에 대한 이해

- 트랜잭션(Transaction)의 필요성

만약 데이터베이스의 데이터를 수정하는 도중에 예외가 발생된다면 어떻게 해야 할까? DB의 데이터들은 수정이 되기 전의 상태로 다시 되돌아가져야 하고, 다시 수정 작업이 진행되어야 할 것이다.
이렇듯 여러 작업을 진행하다가 문제가 생겼을 경우 이전 상태로 롤백하기 위해 사용되는 것이 트랜잭션(Transaction) 이다.

트랜잭션은 더 이상 쪼갤 수 없는 최소 작업 단위를 의미한다. 그래서 트랜잭션은 commit으로 성공 하거나 rollback으로 실패 이후 취소되어야 한다.

하지만 모든 트랜잭션이 동일한 것은 아니고 속성에 따라 동작 방식을 다르게 해줄 수 있다.
예를 들어 1개의 새로운 데이터를 추가하는 와중에 에러가 발생하면 해당 추가 작업은 없었던 것처럼 되돌려진다. 하지만 만약 여러 개의 작업에 대해 롤백을 하려면 어떻게 해야 될까? 여러 개의 작업을 1개의 트랜잭션으로 관리해야 할 것이다.

위에서 설명한 것과 마찬가지로 트랜잭션의 마무리 작업으로는 크게 2가지가 있다.
•트랜잭션 커밋: 작업이 마무리 됨
•트랜잭션 롤백: 작업을 취소하고 이전의 상태로 돌림

만약 여러 작업이 모두 마무리 되었다면 트랜잭션 커밋을 통해 작업이 마무리되었음을 알려주어 반영해야 하며, 만약 문제가 생겼다면 작업 취소를 위해 트랜잭션 롤백 처리를 해주어야 한다.

chapter 00

Spring Transaction

1. Transaction
2. **Spring | Transaction**
3. Spring | Transaction Setting
4. Spring | How to use transaction

■ Spring이 제공하는 Transaction(트랜잭션) 핵심 기술

Spring은 트랜잭션과 관련된 3가지 핵심 기술을 제공하고 있다.

- Transaction **Synchronization(동기화)**
- Transaction **abstraction(추상화)**
- **AOP**를 이용한 Transaction **isolation(분리)**

■ Transaction Synchronization(동기화)

JDBC를 이용하는 개발자가 직접 여러 개의 작업을 하나의 트랜잭션으로 관리하려면 Connection 객체를 공유하는 등 상당히 불필요한 작업들이 많이 생길 것이다.

Spring은 이러한 문제를 해결하고자 트랜잭션 동기화(Transaction Synchronization) 기술을 제공하고 있다.
트랜잭션 동기화는 **트랜잭션을 시작하기 위한 Connection 객체를 특별한 저장소에 보관해두고 필요할 때 꺼내쓸 수 있도록 하는 기술**이다.

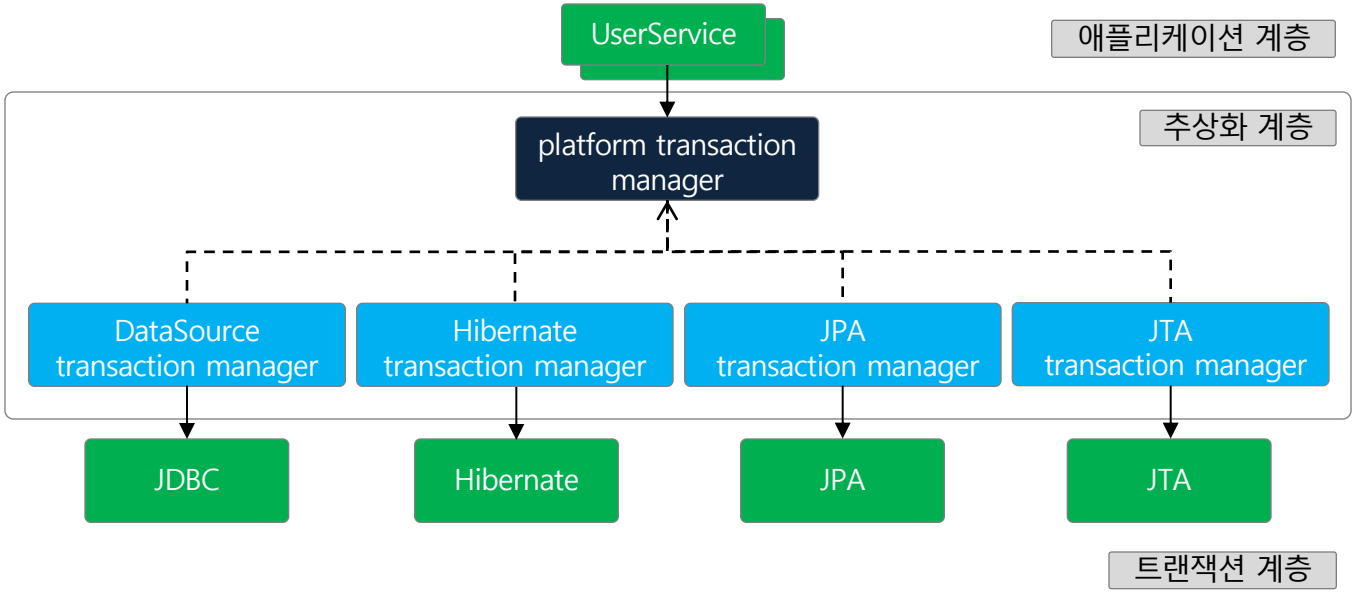
트랜잭션 동기화 저장소는 작업 쓰레드마다 Connection 객체를 독립적으로 관리하기 때문에, 멀티쓰레드 환경에서도 충돌이 발생할 여지가 없다. 그래서 다음과 같이 트랜잭션 동기화를 적용하게 된다.

```
// 동기화 시작
TransactionSynchronizeManager.initSynchronization();
Connection c = DataSourceUtils.getConnection(dataSource);

// 작업 진행

// 동기화 종료
DataSourceUtils.releaseConnection(c, dataSource);
TransactionSynchronizeManager.unbindResource(dataSource);
TransactionSynchronizeManager.clearSynchronization();
```

하지만 개발자가 **JDBC**가 아닌 **Hibernate**와 같은 기술을 쓴다면 위의 JDBC 종속적인 트랜잭션 동기화 코드들은 문제를 유발하게 된다. 대표적으로 Hibernate에서는 Connection이 아닌 Session이라는 객체를 사용하기 때문이다. 이러한 기술 종속적인 문제를 해결하기 위해 Spring은 트랜잭션 관리 부분을 추상화한 기술을 제공하고 있다.

■ Transaction abstraction(추상화)

Spring은 트랜잭션 기술의 공통점을 담은 트랜잭션 추상화 기술을 제공하고 있다.
이를 이용함으로써 애플리케이션에 각 기술 (JDBC, JPA, Hibernate 등)마다 종속적인 코드를 이용하지 않고도 일관되게 트랜잭션을 처리할 수 있도록 해주고 있다.



■ Transaction abstraction(추상화)

Spring이 제공하는 트랜잭션 경계 설정을 위한 추상 인터페이스는 PlatformTransactionManager 이다. 예를 들어 만약 JDBC의 로컬 트랜잭션을 이용한다면 DataSourceTransactionManager 를 이용하면 된다.

이제 우리는 사용하는 기술과 무관하게 PlatformTransactionManager 를 통해 다음의 코드와 같이 트랜잭션을 공유하고, 커밋하고, 롤백할 수 있게 되었다.

```
public Object invoke(MethodInvocation invocation) throws Throwable {
   TransactionStatus
status= this.transactionManager.getTransaction(new DefaultTransactionDefinition());
   try {
      Object ret = invocation.proceed();
      this.transactionManager.commit(status);
      return ret;
   } catch (Exception e) {
      this.transactionManager.rollback(status);
      throw e;
   }
}
```

하지만 위와 같은 트랜잭션 관리 코드들이 비지니스 로직 코드와 결합되어 2가지 책임을 갖고 있다. Spring에서는 AOP를 이용해 이러한 트랜잭션 부분을 핵심 비지니스 로직과 분리하였다.

■ AOP를 이용한 Transaction isolation(분리)

예를 들어 다음과 같이 트랜잭션 코드와 비지니스 로직 코드가 복잡하게 얽혀있는 코드가 있다고 하자.

```
public void addUsers(List<User> userList) {
    TransactionStatus status= this.transactionManager.getTransaction( new DefaultTransactionDefinition() );
    try {
        for (User user: userList) {
            if( isEmailNotDuplicated( user.getEmail() ) ){ userRepository.save(user); }
        }
        this.transactionManager.commit(status);
    } catch (Exception e) {
        this.transactionManager.rollback(status); throw e
    }
}
```

위의 코드는 여러 책임을 가질 뿐만 아니라 서로 성격도 다르고 주고받는 것도 없으므로 분리하는 것이 적합하다.
하지만 위의 코드를 어떻게 분리할 것인지에 대한 고민을 해야 한다. 흔히 떠올릴 수 있는 방법으로는 내부 메소드로 추출하거나 DI로 합성을 이용해 해결하거나 상속을 이용할 수 있을 것이다.
하지만 위의 어떠한 방법을 이용하여도 **트랜잭션을 담당하는 기술 코드를 완전히 분리시키는 것이 불가능** 하였다.
그래서 Spring에서는 마치 트랜잭션 코드와 같은 부가 기능 코드가 존재하지 않는 것 처럼 보이기 위해 해당 로직을 클래스 밖으로 빼내서 별도의 모듈로 만드는 AOP(Aspect Oriented Programming, 관점 지향 프로그래밍)를 고안 및 적용하게 되었고, 이를 적용한 트랜잭션 어노테이션(@Transactional)을 지원하게 되었다.

■ AOP를 이용한 Transaction isolation(분리)

트랜잭션 코드와 비지니스 로직 코드가 얽혀있는 코드가 트랜잭션 어노테이션(@Transactional)을 지원하면서 코드가 간결해 진다

```
@Service
@RequiredArgsConstructor
@Transactional
public class UserService {

    private final UserRepository userRepository;

    public void addUsers(List<User> userList) {
        for (User user : userList) {
            if ( isEmailNotDuplicated( user.getEmail() ) ) {
                userRepository.save(user);
            }
        }
    }
}
```

■ Spring 트랜잭션의 세부 설정

Spring의 DefaultTransactionDefinition이 구현하고 있는 TransactionDefinition 인터페이스는 트랜잭션의 동작방식에 영향을 줄 수 있는 네 가지 속성을 정의하고 있다. 해당 4가지 속성은 트랜잭션을 세부적으로 이용할 수 있게 도와주며, @Transactional 어노테이션에도 공통적으로 적용할 수 있다.

- Transaction **Propagation** (트랜잭션 전파)
- Transaction **Isolation Level** (트랜잭션 격리 수준)
- Transaction **Timeout** (트랜잭션 제한시간)
- Transaction **Readonly** (트랜잭션 읽기전용)

■ Transaction Propagation (트랜잭션 전파)

트랜잭션 전파란 트랜잭션의 경계에서 이미 진행중인 트랜잭션이 있거나 없을 때 어떻게 동작할 것인가를 결정하는 방식을 의미한다. 예를 들어 어떤 A 작업에 대한 트랜잭션이 진행중이고 B 작업이 시작될 때 B 작업에 대한 트랜잭션을 어떻게 처리할까에 대한 부분이다.

1. A의 트랜잭션에 참여(PROPAGATION_REQUIRED)
B의 코드는 새로운 트랜잭션을 만들지 않고 A에서 진행중이 트랜잭션에 참여할 수 있다. 이 경우 B의 작업이 마무리 되고 나서, 남은 A의 작업(2)을 처리할 때 예외가 발생하면 A와 B의 작업이 모두 취소된다. 왜냐하면 A와 B의 트랜잭션이 하나로 묶여있기 때문이다.

2. 독립적인 트랜잭션 생성(PROPAGATION_REQUIRES_NEW)
반대로 B의 트랜잭션은 A의 트랜잭션과 무관하게 만들 수 있다. 이 경우 B의 트랜잭션 경계를 빠져나오는 순간 B의 트랜잭션은 독자적으로 커밋 또는 롤백되고, 이것은 A에 어떠한 영향도 주지 않는다. 즉, 이후 A가 (2)번 작업을 하면서 예외가 발생해 롤백되어도 B의 작업에는 영향을 주지 못한다.

3. 트랜잭션 없이 동작(PROPAGATION_NOT_SUPPORTED)
B의 작업에 대해 트랜잭션을 걸지 않을 수 있다. 만약 B의 작업이 단순 데이터 조회라면 굳이 트랜잭션이 필요 없을 것이다.

이렇듯 이미 진행중인 트랜잭션이 어떻게 영향을 미칠 수 있는가에 대한 부분이 트랜잭션 전파 속성이다. 트랜잭션을 시작하는 것처럼 보이는 getTransaction() 코드가 begin이 아닌 get으로 이름이 지어진 이유도 여기에 있다. 해당 트랜잭션은 다른 트랜잭션에 묶여있을 수 있기 때문이다. 위에서 설명한 대표적인 처리 방법 외에도 다양한 처리 방법을 지원하고 있다.

■ Transaction Isolation Level (트랜잭션 격리 수준)

모든 DB 트랜잭션은 격리수준을 가지고 있어야 한다.

서버에서는 여러 개의 트랜잭션이 동시에 진행될 수 있는데, 모든 트랜잭션을 독립적으로 만들고 순차 진행 한다면 안전하겠지만 성능이 크게 떨어질 수 밖에 없다. 따라서 적절하게 격리수준을 조정해서 가능한 많은 트랜잭션을 동시에 진행시키면서 문제가 발생하지 않도록 제어해야 한다. 이는 JDBC 드라이버나 DataSource 등에서 재설정할 수 있고, 트랜잭션 단위로 격리 수준을 조정할 수도 있다.

DefaultTransactionDefinition에 설정된 격리수준은 **ISOLATION_DEFAULT**로 DataSource에 정의된 격리 수준을 따른다는 것이다.

기본적으로는 DB나 DataSource에 설정된 기본 격리 수준을 따르는 것이 좋지만, 특별한 작업을 수행하는 메소드라면 독자적으로 지정해줄 필요가 있다.

■ Transaction Timeout (트랜잭션 제한시간)

트랜잭션을 수행하는 제한시간을 설정할 수 있다.

제한시간의 설정은 트랜잭션을 직접 시작하는 **PROPAGATION_REQUIRED**나 **PROPAGATION_REQUIRES_NEW**의 경우에 사용해야만 의미가 있다.

■ Transaction Readonly (트랜잭션 읽기전용)

읽기전용으로 설정해두면 트랜잭션 내에서 데이터를 조작하는 시도를 막아줄 수 있을 뿐만 아니라 데이터 액세스 기술에 따라 성능이 향상될 수 있다.

chapter 00

Spring Transaction

1. Transaction
2. Spring | Transaction
3. **Spring | Transaction Setting**
4. Spring | How to use transaction

■ 전파 속성(Propagation)

Spring이 제공하는 선언적 트랜잭션(트랜잭션 어노테이션, @Transactional)의 장점 중 하나는 여러 트랜잭션 적용 범위를 묶어서 커다란 하나의 트랜잭션 경계를 만들 수 있다는 점이다. 우리는 Spring이 트랜잭션을 어떻게 진행시킬지 결정하도록 전파 속성을 전달해야 하는데, 이를 통해 새로운 트랜잭션을 시작할지 또는 기존의 트랜잭션에 참여할지 등을 결정하게 된다.

Spring이 지원하는 전파 속성은 다음의 7가지가 있다.

- REQUIRED
- SUPPORTS
- MANDATORY
- REQUIRES_NEW
- NOT_SUPPORTED
- NEVER
- NESTED

■ 전파 속성(Propagation)

- REQUIRED
  - 디폴트 속성으로써 모든 트랜잭션 매니저가 지원함
  - 이미 시작된 트랜잭션이 있으면 참여하고 없으면 새로 시작
  - 하나의 트랜잭션이 시작된 후 다른 트랜잭션 경계가 설정된 메소드를 호출하면 같은 트랜잭션으로 묶임

  REQUIRED는 기본 속성으로써 모든 트랜잭션 매니저가 지원하며, 대부분의 경우 이 속성이면 충분하다. REQUIRED는 미리 시작된 트랜잭션이 있으면 참여하고 없으면 새로 시작한다. 가장 간단하고 자연스러운 트랜잭션 전파 방식이지만 매우 강력하며 유용하다. REQUIRED 속성일 때 하나의 트랜잭션이 시작된 후 다른 트랜잭션 경계가 설정된 메소드를 호출하면 자연스럽게 같은 트랜잭션으로 묶인다.

- SUPPORTS
  - 이미 시작된 트랜잭션이 있으면 참여하고, 그렇지 않으면 트랜잭션 없이 진행함
  - 트랜잭션이 없어도 해당 경계 안에서 Connection 객체나 하이버네이트의 Session 등은 공유 할 수 있음

  SUPPORTS는 이미 시작 트랜잭션이 있으면 참여하고, 그렇지 않으면 트랜잭션 없이 진행하게 만든다. 트랜잭션이 없기는 하지만 해당 경계 안에서 Connection 객체나 하이버네이트의 Session 등은 공유를 할 수 있다.

■ 전파 속성(Propagation)

- MANDATORY
  - 이미 시작된 트랜잭션이 있으면 참여하고, 없으면 새로 시작하는 대신 없으면 예외를 발생시킴
  - MANDATORY는 혼자서 독립적으로 트랜잭션을 진행하면 안되는 경우에 사용

  MANDATORY 역시 이미 시작된 트랜잭션이 있으면 참여한다. 하지만 트랜잭션이 시작된 것이 없으면 새로 시작하는 대신 예외를 발생시킨다. 즉, MANDATORY는 혼자서 독립적으로 트랜잭션을 진행하면 안되는 경우에 사용할 수 있다.

- REQUIRES_NEW
  - 항상 새로운 트랜잭션을 시작해야 하는 경우에 사용
  - 이미 진행중인 트랜잭션이 있으면 이를 보류시키고 새로운 트랜잭션을 만들어 시작함

  REQUIRES_NEW는 항상 새로운 트랜잭션을 시작해야 하는 경우에 사용할 수 있다. 만약 이미 시작된 트랜잭션이 있으면 트랜잭션을 잠시 보류시키고 새로운 트랜잭션을 만들어 시작하게 된다. 만약 JTA 트랜잭션 매니저를 사용한다면 서버의 트랜잭션 매니저에 트랜잭션 보류가 가능하도록 설정되어 있어야 한다.

■ 전파 속성(Propagation)

- NOT_SUPPORTED
  - 이미 진행중인 트랜잭션이 있으면 이를 보류시키고, 트랜잭션을 사용하지 않도록 함

- NEVER
  - 이미 진행중인 트랜잭션이 있으면 예외를 발생시키며, 트랜잭션을 사용하지 않도록 강제함

■ 전파 속성(Propagation)

- NESTED
  - 이미 진행중인 트랜잭션이 있으면 중첩(자식) 트랜잭션을 시작함
  - 중첩 트랜잭션은 트랜잭션 안에 다시 트랜잭션을 만드는 것으로, 독립적인 트랜잭션을 만드는 REQUIRES_NEW와는 다름
  - NESTED에 의한 중첩 트랜잭션은 먼저 시작된 부모 트랜잭션의 커밋과 롤백에는 영향을 받지만, 자신의 커밋과 롤백은 부모 트랜잭션에게 영향을 주지 않음

예를 들어 어떤 중요한 작업을 진행하면서 작업 로그를 DB에 저장해야 한다고 해보자. 그런데 로그를 저장하는 작업은 실패를 하더라도 메인 작업의 트랜잭션까지는 롤백하지 말아야 하는 경우가 있다. 왜냐하면 힘들게 처리한 중요한 작업을 로그를 남기지 못해서 모두 실패로 만들 수 없기 때문이다. 반면에 핵심 작업에서 예외가 발생한다면 이때는 저장된 로그도 제거해야 한다.

이렇듯 **부모의 트랜잭션은 자식의 작업에 영향을 줘야하지만 자식의 트랜잭션은 부모에 영향을 주지 않아야 할 때** NESTED 전파 속성을 이용할 수 있다.

그리고 이러한 중첩 트랜잭션은 JDBC 3.0 스펙의 저장포인트(SavePoint)를 지원하는 드라이버와 DataSourceTransactionManager를 이용할 경우에 적용할 수 있다. 또는 중첩 트랜잭션을 지원하는 일부 WAS의 JTA 트랜잭션 매니저를 이용할 때도 적용할 수 있다. 즉, NESTED는 모든 트랜잭션 매니저에 적용 가능하지 않으므로 사용하는 트랜잭션 매니저와 드라이버 또는 WAS 등을 확인해야 한다.

선언적 트랜젝션에서는 @Transactional 어노테이션의 propagation 엘리먼트로 원하는 전파 속성을 지정할 수 있으며, 기본은 REQUIRED로 설정되어 있다. 또한 Spring은 6가지 전파 속성 방법을 지원하고 있지만 해당 속성을 지원하지 않는 트랜잭션 매니저와 데이터 액세스 기술이 있을 수 있다. 그러므로 변경이 필요한 경우 별도의 설정이 필요한지 확인을 해봐야 한다.

■ 격리 수준(Isolation)

트랜잭션 격리수준은 동시에 여러 트랜잭션이 진행될 때 트랜잭션의 작업 결과를 여타 트랜잭션에게 어떻게 노출할 것인지를 결정한다.

스프링은 다음의 5가지 격리수준 속성을 지원한다.

- DEFAULT
- READ_UNCOMMITTED
- READ_COMMITTED
- REPEATABLE_READ
- SERIALIZABLE

■ 격리 수준(Isolation)

- DEFAULT
  - 사용하는 데이터 액세스 기술 또는 DB 드라이버의 디폴트 설정을 따름
  - 일반적으로 드라이버의 격리 수준은 DB의 격리 수준을 따르며, 대부분의 DB는 READ_COMMITED를 기본 격리수준으로 가짐
  - 일부 DB는 디폴트 값이 다른 경우도 있으므로 DEFAULT를 사용할 때는 드라이버와 DB의 문서를 참고해서 기본 격리수준을 확인해야 함

  DEFAULT는 사용하는 데이터 액세스 기술 또는 DB 드라이버의 디폴트 설정을 따른다. 물론 일반적으로 드라이버의 격리 수준은 DB의 격리 수준을 따르며, 대부분의 DB는 READ_COMMITED를 기본 격리수준으로 갖는다. 하지만 일부 DB는 디폴트 값이 다른 경우도 있으므로 DEFAULT를 사용할 경우에는 드라이버와 DB의 문서를 참고해서 기본 격리수준을 확인해야 한다.

- READ_UNCOMMITTED
  - 가장 낮은 격리수준으로써 하나의 트랜잭션이 커밋되기 전에 그 변화가 다른 트랜잭션에 그대로 노출되는 문제가 있음
  - 하지만 가장 빠르기 때문에 데이터의 일관성이 조금 떨어지더라도 성능을 극대화할 때 의도적으로 사용함

  가장 낮은 격리수준으로써 하나의 트랜잭션이 커밋되기 전에 그 변화가 다른 트랜잭션에 그대로 노출되는 문제가 있다. 하지만 가장 빠르기 때문에 데이터의 일관성이 조금 떨어지더라도 성능을 극대화할 때 의도적으로 사용한다.

■ 격리 수준(Isolation)

- READ_COMMITTED
  - Spring은 기본 속성이 DEFAULT이며, DB는 일반적으로 READ_COMMITED가 기본 속성이므로 가장 많이 사용됨
  - READ_UNCOMMITTED와 달리 다른 트랜잭션이 커밋하지 않은 정보는 읽을 수 없음
  - 대신 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정할 수 있어서 처음 트랜잭션이 같은 로우를 다시 읽을 때 다른 내용이 발견될 수 있음

  READ_COMMITTED는 가장 많이 사용되는 격리수준이다. Spring은 기본적으로 DEFAULT로 지정되어 있고, DB는 일반적으로 READ_COMMITED로 되어있기 때문이다. READ_COMMITTED는 READ_UNCOMMITTED와 달리 다른 트랜잭션이 커밋하지 않은 정보는 읽을 수 없다. 대신 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정할 수 있다. 이 때문에 처음 트랜잭션이 같은 로우를 다시 읽을 때 다른 내용이 발견될 수 있다.

- REPEATABLE_READ
  - 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정할 수 없도록 막아주지만 새로운 로우를 추가하는 것은 막지 않음
  - 따라서 SELECT로 조건에 맞는 로우를 전부 가져오는 경우 트랜잭션이 끝나기 전에 추가된 로우가 발견될 수 있음

  REPEATABLE_READ는 하나의 트랜잭션이 읽은 로우를 다른 트랜잭션이 수정할 수 없도록 막아준다. 하지만 새로운 로우를 추가하는 것은 막지 않는다. 따라서 SELECT로 조건에 맞는 로우를 전부 가져오는 경우 트랜잭션이 끝나기 전에 추가된 로우가 발견될 수 있다.

■ 격리 수준(Isolation)

- SERIALIZABLE
  - 가장 강력한 트랜잭션 격리 수준으로, 이름 그대로 트랜잭션을 순차적으로 진행시켜줌
  - 그러므로 SERIALIZABLE은 여러 트랜잭션이 동시에 같은 테이블의 정보를 액세스할 수 없음
  - SERIALIZABLE은 가장 안전하지만 가장 성능이 떨어지므로 극단적으로 안전한 작업이 필요한 경우가 아니라면 사용해서는 안됨

SERIALIZABLE은 가장 강력한 트랜잭션 격리 수준으로, 이름 그대로 트랜잭션을 순차적으로 진행시켜준다. 그렇기 때문에 SERIALIZABLE은 여러 트랜잭션이 동시에 같은 테이블의 정보를 액세스할 수 없다.
SERIALIZABLE은 가장 안전하지만 가장 성능이 떨어지는 격리수준이기 때문에 극단적으로 안전한 작업이 필요한 경우가 아니라면 사용해서는 안된다.

선언적 트랜젝션에서는 @Transactional 어노테이션의 isolation 엘리먼트로 원하는 전파 속성을 지정할 수 있으며, 기본은 **DEFAULT**로 설정되어 있다.

03 Spring | Transaction Setting

■ 읽기 전용(readOnly)

Spring에서는 트랜잭션을 다음의 2가지 목적(성능 최적화와 쓰기 방지)으로 읽기 전용(readOnly)으로 설정할 수 있다.

- 읽기 전용으로 설정함으로써 성능을 최적화함
- 쓰기 작업이 일어나는 것을 의도적으로 방지함

읽기 전용으로 설정이 되어 있으면 트랜잭션을 준비하면서 트랜잭션 매니저에게 이러한 정보가 전달되며 그에 따라 트랜잭션 매니저가 적절한 작업을 수행한다. 그런데 일부 트랜잭션 매니저의 경우 읽기전용 속성을 무시하고 쓰기 작업을 하용할 수도 있으므로 주의해야 한다. 물론 일반적으로 읽기전용 트랜잭션이 시작된 이후 INSERT, UPDATE, DELETE의 작업이 진행되면 예외가 발생한다.

■ 롤백/커밋(rollback/commit) 예외

선언적 트랜잭션에서는 런타임 예외가 발생하면 롤백하고, 예외가 발생하지 않았거나 체크 예외가 발생하였다면 커밋한다. 여기서 체크 예외를 커밋 대상으로 삼은 이유는 체크 예외가 예외적인 상황에서 사용되기 보다는 반환값을 대신해 비지니스적인 의미를 담은 결과로 많이 사용되기 때문이다. 스프링에서는 데이터 액세스 기술의 예외는 런타임 예외로 전환하여 던지므로 런타임 예외만 롤백 대상으로 삼은 것이다.
하지만 롤백/커밋의 동작 방식의 변경을 원한다면 설정을 통해 동작 방식을 바꿀 수 있는데, 커밋 대상이지만 롤백을 발생시킬 예외나 클래스 이름은 각각 rollbackFor 또는 rollbackForClassName으로 지정할 수 있으며, 반대로 롤백 대상인 런타임 예외를 트랜잭션 커밋 대상으로 지정하기 위해서는 noRollbackFor 또는 noRollbackForClassName을 이용할 수 있다.

■ 제한 시간(timeout)

timeout 속성을 이용하면 트랜잭션에 제한시간을 지정할 수 있다. 값은 int 타입의 초 단위로 지정할 수 있는데 문자열로 지정하기를 원한다면 timeoutString을 사용하면 된다.
만약 별도로 값을 설정해주지 않는다면 트랜잭션 시스템의 제한시간을 따르며, 제한 시간을 직접 지정했는데 트랜잭션 매니저에서 이 기능을 지원하지 못한다면 예외가 발생할 수 있다.

모든 트랜잭션 속성들은 각각의 트랜잭션 경계설정 속성에 지정할 수 있다. 하지만 보통은 readOnly 속성 정도만 활용하고, 나머지는 디폴트로 사용하는 경우가 많다. 세밀한 속성은 DB나 WAS의 트랜잭션 매니저 설정을 이용해도 되기 때문이다.

chapter 00

Spring Transaction

1. Transaction
2. Spring | Transaction
3. Spring | Transaction Setting
4. **Spring | How to use transaction**

04 Spring | How to use transaction

■ 1. @Transactional (declarative transaction)

- **@Transactional 어노테이션**

Spring에서는 클래스나 인터페이스 또는 메소드에 부여할 수 있는 @Transactional이라는 어노테이션을 제공하고 있다. 이 어노테이션이 붙으면 스프링은 해당 타깃을 포인트 컷의 대상으로 자동 등록하며 트랜잭션 관리 대상이 된다. 즉, 이 어노테이션을 통해 포인트 컷에 등록하고 트랜잭션 속성을 부여하는 것이다.

이렇듯 AOP를 이용해 코드 외부에서 트랜잭션의 기능을 부여해주고 속성을 지정할 수 있게 해주는 것을 **선언적 트랜잭션(declarative transaction)**이라고 한다.

반대로 TrnasactionTemplate이나 개별 데이터 기술의 트랜잭션 API를 이용해 직접 코드안에서 사용하는 방법을 **프로그램에 의한 트랜잭션(programmatic transaction)**이라고 한다.

스프링에서는 두 가지 방식을 모두 지원하지만, 특별한 경우가 아니라면 선언적 트랜잭션을 사용하는 것이 좋다.

왜냐하면 @Transactional 어노테이션을 이용하면 트랜잭션 속성을 **메소드 단위로** 다르게 지정할 수 있어 매우 세밀한 트랜잭션 속성 제어가 가능할 뿐만 아니라 **직관적**이므로 이해하기도 좋다.

이 어노테이션을 이용하려면 @EnableTransactionManagement을 추가 해주어야 한다.

■ 1. @Transactional (declarative transaction)

- **@Transactional의 롤백 처리**

Java에는 체크 예외(Checked Exception)와 언체크 예외(Unchecked Exception)가 있다. 두 가지 예외 종류를 구분하는 것이 중요한 이유는 트랜잭션 롤백 범위가 다르기 때문이다. 체크 예외란 Exception 클래스 하위 클래스이며, 언체크 예외란 Exception 하위의 RuntimException 하위의 예외이다.

스프링의 선언적 트랜잭션(@Transactional) 안에서 예외가 발생했을 때, 해당 예외가 언체크 예외(런타임 예외)라면 자동적으로 롤백이 발생한다. 하지만 체크 예외라면 롤백이 되지 않는다. 체크 예외를 롤백시키기 위해서는 @Transactional의 rollbackFor 속성으로 해당 체크 예외를 적어주어야 한다.

스프링의 트랜잭션이 언체크 예외(런타임 예외)나 에러(Error) 만을 롤백 대상으로 보는 이유는 해당 예외들이 복구 가능성이 없는 예외들이므로 별도의 try-catch나 throw를 통해 처리를 강제하지 않기 때문이다. 스프링의 트랜잭션은 내부적으로 언체크 예외(런타임 예외)이거나 에러(Error) 인지 검사한 후에 맞으면 롤백 여부를 결정하는 rollback-only를 True로 변경하는 로직이 있다. (정확히는 TransactionInfo의 transactionAttribute의 rollbackOn에 의해 검사된다)

```
@Override
public boolean rollbackOn(Throwable ex) {
    return (ex instanceof RuntimeException || ex instanceof Error);
}
```

■ 1. @Transactional (declarative transaction)

- **@Transactional의 롤백 처리**

그리고 만약 언체크 예외라면 rollback 처리를 진행한다.

```
protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txInfo, Throwable ex) {
    if (logger.isTraceEnabled()) {
        logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() + "] after except
    }
    if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.rollbackOn(ex)) {
        try {
            txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
        }
    ...

}
```

■ 1. @Transactional (declarative transaction)

- **@Transactional의 대체 정책(Fallback Policy)**

만약 모든 메소드에 @Transactional이 붙어있으면 메소드가 상당히 더러워진다. 그래서 스프링은 메소드 외에도 클래스와 인터페이스에 어노테이션을 붙일 수 있도록 하고 있다. 그리고 트랜잭션 어노테이션을 적용할 때 타깃 메소드, 타깃 클래스, 선언 메소드, 선언 타입(클래스 or 인터페이스) 순으로 @Transactional이 적용되었는지 차례로 확인하고, 가장 먼저 발견되는 속성 정보를 사용한다. 이를 4단계의 대체 정책(fallback policy)라고 부르며, 이를 통해 어노테이션을 최소화하는 동시에 세밀한 제어를 해줄 수 있다.

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 비지니스 로직과의 결합**

트랜잭션을 중구난방으로 적용하는 것은 좋지 않다. 대신 특정 계층의 경계를 트랜잭션 경계와 일치시키는 것이 좋은데, 일반적으로 비지니스 로직을 담고 있는 서비스 계층의 메소드와 결합시키는 것이 좋다. 왜냐하면 데이터 저장 계층으로부터 읽어온 데이터를 사용하고 변경하는 등의 작업을 하는 곳이 서비스 계층이기 때문이다. 위와 같이 클래스 레벨에 트랜잭션 어노테이션을 붙여주면 메소드까지 적용이 된다

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 비지니스 로직과의 결합**

```
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class UserService {

    private final UserRepository userRepository;
    private final BCryptPasswordEncoder passwordEncoder;

    public List<User> getUserList() {
        return userRepository.findAll();
    }

}
```

서비스 계층을 트랜잭션의 시작과 종료 경계로 정했다면, 테스트와 같은 특별한 이유가 아니고는 다른 계층이나 모듈에서 DAO에 직접 접근하는 것은 차단해야 한다. 트랜잭션은 보통 서비스 계층의 메소드 조합을 통해 만들어지기 때문에 DAO가 제공하는 주요 기능은 서비스 계층에 위임 메소드를 만들어둘 필요가 있다. 그리고 가능하면 다른 모듈의 DAO에 접근할 때는 서비스 계층을 거치도록 하는 것이 바람직하다.

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 읽기 전용 트랜잭션의 공통화**

```
@RequiredArgsConstructor
@Transactional(readOnly = true)
public class UserService {

    private final UserRepository userRepository;
    private final BCryptPasswordEncoder passwordEncoder;

    public List<User> getUserList() {
        return userRepository.findAll();
    }

    @Transactional
    public User signUp(final SignUpDTO signUpDTO) {
        final User user = User.builder()
                .email(signUpDTO.getEmail())
                .pw(passwordEncoder.encode(signUpDTO.getPw()))
                .role(UserRole.ROLE_USER)
                .build();

        return userRepository.save(user);
    }

}
```

클래스 레벨에는 공통적으로 적용되는 읽기전용 트랜잭션 어노테이션을 선언하고, 추가나 삭제 또는 수정이 있는 작업에는 쓰기가 가능하도록 별도로 @Transacional 어노테이션을 메소드에 선언하는 것이 좋다. 이를 체감하기는 힘들겠지만 약간의 성능적인 이점을 얻을 수 있다.

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 테스트의 롤백**

트랜잭션 어노테이션을 테스트에 붙이면 테스트의 DB 커밋을 롤백해주는 기능이 있다.
DB와 연동되는 테스트를 할 때에는 DB의 상태와 데이터가 상당히 중요하다. 하지만 문제는 테스트에서 DB에 쓰기 작업을 하면 DB의 데이터가 바뀌는 것인데, 트랜잭션 어노테이션을 테스트에 활용하면 테스트를 진행하는 동안에 조작한 데이터를 모두 롤백하고 테스트를 진행하기 전의 상태로 만들어준다. 어떠한 경우에도 커밋을 하지 않기 때문에 테스트가 성공하거나 실패해도 상관이 없으며 심지어 예외가 발생해도 어떠한 문제가 발생하지 않는다. 강제로 롤백시키도록 설정되어 있기 때문이다.

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 테스트의 롤백**

```
@Transactional
@ExtendWith(SpringExtension.class)
@DataJpaTest
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void findByEmailAndPw() {
        final User user = User.builder()
                .email("email")
                .pw("pw")
                .role(UserRole.ROLE_USER).build();
        userRepository.save(user);

        assertThat(userRepository.findAll().size()).isEqualTo(1);
    }

}
```

■ 2. Spring에서 트랜잭션의 사용법

- **Spring에서 트랜잭션의 활용법 - 테스트의 롤백**

하지만 테스트 메소드 안에서 진행되는 작업을 하나의 트랜잭션으로 묶고는 싶지만 강제 롤백을 원하지 않을 수 있다. 테스트의 작업을 그대로 DB에 반영하고 싶다면 @Rollback(false)를 이용해주면 된다. @Rollback은 메소드에만 적용가능하므로, 클래스 레벨에 부여하기를 원한다면 @TransactionConfiguration(defaultRollback=false) 를 이용하고, 롤백을 원하는 메소드에 @Rollback(true)를 이용하면 된다.

물론 여기서 auto_increment나 sequence 등에 의해 증가된 값은 롤백이 되지 않는다. 그렇기 때문에 테스트를 위해서는 별도의 데이터베이스로 연결을 하거나 또는 H2와 같은 휘발성(인메모리) 데이터베이스를 사용하는 것이 좋다.