Spring Transaction

Transaction

- 한 개 이상의 물리적인 동작으로 구성된 논리적인 작업 단위

- 트랜잭션에 포함된 물리적인 작업이 모두 성공하거나 실패하도록 관리

- ACID 특성
  - 원자성 (Atomicity)
  - 일관성 (Consistency)
  - 격리성 (Isolation)
  - 영속성 (Durability)

JDBC transaction

- Connection 객체의 autoCommit 속성과 commit(), rollback() 메서드를 사용해서 구현

```
Connection conn = null;
PreparedStatement pstmt = null;

try {
    Class.forName("com.mysql.cj.jdbc.Driver");

    conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/demo?serverTimezone=UTC", "devuser", "mariadb");

    conn.setAutoCommit(false); //executeUpdate 실행 후에 자동으로 commit 하지 마세요

    // execute query

    conn.commit(); //마지막 commit 또는 rollback 실행 후에 처리된 모든 명령을 commit
    System.out.println("계좌 이체 성공");

} catch (Exception ex) {

    try { conn.rollback(); } catch (Exception ex2) {} //마지막 commit 또는 rollback 실행 후에 처리된 모든 명령을 rollback
    System.out.println("계좌 이체 실패");

} finally {
    try { conn.setAutoCommit(true); } catch (Exception ex) {}
    try { pstmt.close(); } catch (Exception ex) {}
    try { conn.close(); } catch (Exception ex) {}
}
```

Spring 트랜잭션 지원

- 반복적인 트랜잭션 관리 코드 없이 간단한 코드로 트랜잭션 관리 가능하도록 지원

- 프로그래밍 방식
  - TransactionTemplate 클래스 사용

- AOP 방식
  - <tx:advice>와 <aop:advisor> 설정을 사용하는 선언적 트랜잭션 방식

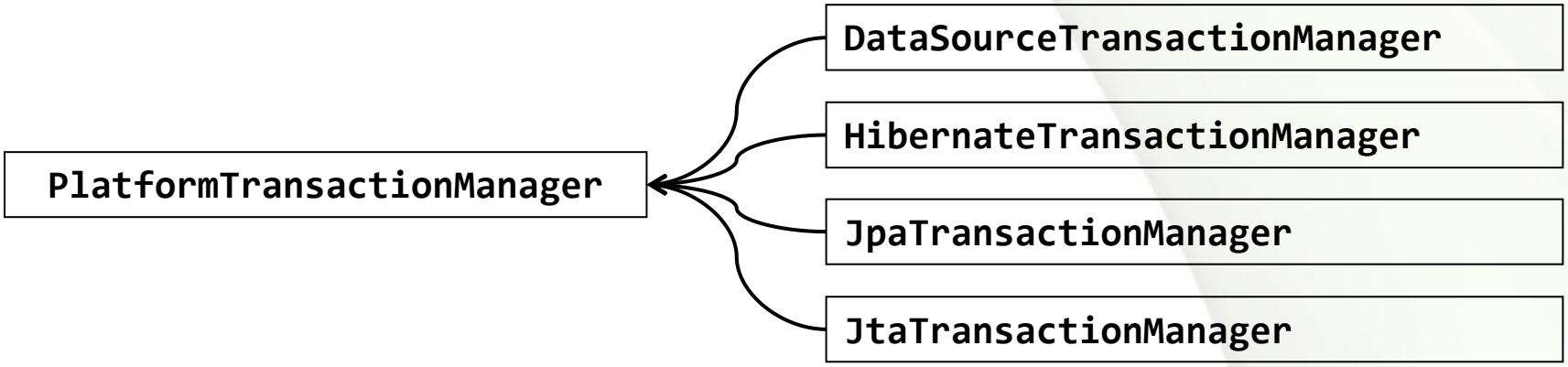
- 어노테이션 방식
  - @Transactional 어노테이션을 사용하는 선언적 트랜잭션 방식
  - 내부에서는 AOP를 사용해서 트랜잭션 관리

Spring 트랜잭션 관리자

- 스프링 프레임워크는 데이터 연동 기술에 따라 특화된 트랜잭션 관리자 제공
  - 모든 트랜잭션 관리자는 PlatformTransactionManager 인터페이스 구현

- 종류

트랜잭션 관리자	설명
DataSourceTransactionManager	MyBatis와 같은 JDBC 기반 트랜잭션 관리
HibernateTransactionManager	Hibernate 프레임워크 기반 트랜잭션 관리
JpaTranasctionManager	JPA 기반 트랜잭션 관리
JtaTransactionManager	분산 트랜잭션 지원



트랜잭션 관리자 구성

- Annotation 사용

```
@Bean
public PlatformTransactionManager transactionManager() {
    DataSourceTransactionManager tm = new DataSourceTransactionManager();
    tm.setDataSource(dataSource());
    return tm;
}
```

- XML 기반 설정

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

트랜잭션 구현 (XML 설정 기반)

- 빈 선언 및 의존성 주입

```
<bean id="txManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="txTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="txManager" />
</bean>

<bean id="memberDao" class="spring.MemberDao">
    <constructor-arg ref="dataSource" />
</bean>

<bean id="changePwdSvc" class="spring.ChangePasswordService">
    <property name="memberDao" ref="memberDao" />
    <property name="txTemplate" ref="txTemplate" />
</bean>
```

트랜잭션 구현 (XML 설정 기반)

- 트랜잭션 적용

```
public void changePassword(String email, String oldPwd, String newPwd) {
    txTemplate.execute(new TransactionCallback<Void>() {

        @Override
        public Void doInTransaction(TransactionStatus txStatus) {

            try {
                Member member = memberDao.selectByEmail(email);
                if (member == null)
                    throw new MemberNotFoundException();

                member.changePassword(oldPwd, newPwd);

                memberDao.update(member);

            } catch (Exception ex) {
                txStatus.setRollbackOnly();
            }

            return null;

        }
    });
}
```

트랜잭션 구현 (Annotation 기반)

```java
@Configuration
@EnableTransactionManagement
public class AppCtx {

    @Bean(destroyMethod = "close")
    public DataSource dataSource() {
        DataSource ds = new DataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/spring5fs?characterEncoding=utf8");
        ds.setUsername("spring5");
        ds.setPassword("spring5");
        ds.setInitialSize(2);
        ds.setMaxActive(10);
        ds.setTestWhileIdle(true);
        ds.setMinEvictableIdleTimeMillis(60000 * 3);
        ds.setTimeBetweenEvictionRunsMillis(10 * 1000);
        return ds;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager tm = new DataSourceTransactionManager();
        tm.setDataSource(dataSource());
        return tm;
    }
```

```java
@Transactional(rollbackFor = ArithmeticException.class)
public void changePassword(String email, String oldPwd, String newPwd) {
    Member member = memberDao.selectByEmail(email);
    if (member == null)
        throw new MemberNotFoundException();

    member.changePassword(oldPwd, newPwd);

    memberDao.update(member);
}
```

@Transactional과 프록시

- @Transactional 어노테이션을 이용해서 트랜잭션을 처리하면 내부적으로 AOP를 사용해서 트랜잭션 처리

- 과정
  - @EnableTransactionManagement 적용 → @Transactional이 적용된 bean을 찾아서 알맞은 프록시 객체 생성
  - @Transactional 이 적용된 메서드를 호출하면 PlatformTransactionManager를 사용해서 트랜잭션 시작
  - 정상적으로 처리되면 자동으로 commit 처리
  - 예외가 발생하면 자동으로 rollback 처리
    - » 단, RuntimeException 및 RuntimeException을 상속한 예외에 대해서만 자동으로 rollback 처리
    - » 다른 종류의 예외에 대해서도 rollback을 처리하려면 @Transactional에 rollbackFor 속성에 예외 지정

트랜잭션 속성

- 속성 종류

속성 이름	설명
전파 행위	트랜잭션 영역 설정
분리 수준	한 트랜잭션이 다른 트랜잭션에 영향 받는 정도 설정
읽기 전용	데이터 읽기 최적화 적용
타임아웃	트랜잭션 타임아웃 시간 설정
롤백 규칙	롤백이 발생하는 예외 지정

트랜잭션 속성

- 전파 행위

설정 값	설명
MANDATORY	반드시 트랜잭션 안에서 메서드 실행. 트랜잭션이 없으면 예외 발생
NESTED	트랜잭션 안에서 실행될 경우 중첩된 트랜잭션으로 실행. 트랜잭션이 없는 경우 PROPAGATION_REQUIRED와 동일
NEVER	트랜잭션 없이 실행. 트랜잭션이 있으면 예외 발생
NOT_SUPPORTED	트랜잭션 없이 실행. 기존 트랜잭션이 있는 경우 트랜잭션을 호출한 메서드가 종료될 때까지 보류
REQUIRED	기존 트랜잭션이 있는 경우 기존 트랜잭션 안에서 실행되고, 없는 경우 새 트랜잭션 생성
REQUIRED_NEW	항상 새로운 트랜잭션 생성. 기존 트랜잭션이 있는 경우 기존 트랜잭션은 보류
SUPPORTED	기존 트랜잭션이 있으면 트랜잭션 안에서 실행. 기존 트랜잭션이 없으면 트랜잭션 없이 실행

트랜잭션 속성

- 분리 수준

설정 값	설명
DEFAULT	현재는 ISOLATION_READ_COMMITED와 동일
READ_UNCOMMITTED	트랜잭션에서 변경된 데이터를 커밋하기 전에 다른 트랜잭션에서 읽을 수 있음
READ_COMMITTED	트랜잭션에서 변경된 데이터를 커밋하기 전에 다른 트랜잭션에서 읽을 수 없음
REPEATABLE_READ	트랜잭션 내에서 여러 번의 읽기 수행이 항상 동일한 값을 반환하도록 보장
SERIALIZE	모든 트랜잭션은 한 번에 하나씩만 실행

트랜잭션 설정 가이드

- 트랜잭션은 서비스 레이어에서 시작
- 삽입, 삭제, 변경 기능을 수행하는 메서드에 기본 값인 TRANSACTION_REQUIRED를 지정해서 기존 트랜잭션에 참여하거나 또는 새 트랜잭션을 시작하도록 설정
  - 읽기 메서드에는 트랜잭션이 필요 없으므로 읽기 최적화 지정
- 웹 애플리케이션의 컨트롤러에 트랜잭션 설정을 피하는 것이 권장됨
- Repository 클래스에는 트랜잭션이 필요한 메서드에 TRANSACTION_SUPPORTED를 지정해서 기존 트랜잭션에 참여하도록 설정
- RuntimeException 에 대해서만 롤백 하도록 기본 설정 유지
  - 직접 예외 처리할 경우 자동으로 Rollback 되지 않으므로 UnexpectedRollbackException 예외에서 명시적인 롤백 처리