JDBC 에서 Transaction 처리

- JDBC 를 이용한 Transaction 의 시작은 Connection.setAutoCommit(false) 로 시작합니다.
- COMMIT 과 ROLLBACK 은 Connection 객체의 commit(), rollback() 메소드로 처리할 수 있습니다.
- 하나의 트랜잭션은 하나의 Connection 객체로 이루어져야 합니다.
- 주로 다음과 같이 구현할 수 있습니다.

```
try {
   con.setAutoCommit(false);
   Statement stmt = con.createStatement();

   String SQL = "INSERT INTO JdbcStudents  " +
               "VALUES (3, '학생3', now())";
   stmt.executeUpdate(SQL);

   //같은 ID 로 INSERT 하여 DUPLICATION 에러가 발생합니다.
   String SQL = "INSERTED IN JdbcStudents  " +
               "VALUES (3, '학생4', now())";
   stmt.executeUpdate(SQL);

   //만약 에러가 없다면 COMMIT 하게 됩니다.
   con.commit();
}catch(Exception e){
   // Exception 이 발생하면 ROLLBACK 됩니다.
   con.rollback();
}finally{
   con.close();
}
```

JDBC 에서 ISOLATION LEVEL

- JDBC 에서는 Connection.setTransactionIsolation(int level) 로 설정할 수 있습니다.

Spring Framework 의 Transaction

- Spring Framework 는 @Transactional 를 메소드에 설정하여 Transaction 지원 여부를 결정할 수 있습니다.

Transaction 환경 설정

- DatabaseConfig.java
- @EnableTransactionManagement 를 Configruation 클래스에 설정하고 PlatformTransactionManager 를 스프링 빈으로 등록합니다.

```
package com.nhnacademy.edu.jdbc1.config;

import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@EnableTransactionManagement
public class DatabaseConfig {
    @Bean
    public DataSource dataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();

basicDataSource.setUrl("jdbc:mysql://133.186.211.156:3306/nhn_academy_5
0");
        basicDataSource.setUsername("nhn_academy_50");
        basicDataSource.setPassword("/B7owD_6TtCRJl6C");
        basicDataSource.setInitialSize(10);
        basicDataSource.setMaxTotal(20);

        return basicDataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dataSource());
    }
}
```

- @Service 의 클래스 또는 메소드에 @Transactional 을 설정합니다.

```
    @Override
    @Transactional
    public void insertAndDelete(Student student) {
```

```
        studentRepository.insert(student);
        studentRepository.deleteById(student.getId());
    }
```

@Transactional Propagation

옵션	설명
REQUIRED	기본 옵션 부모 트랜잭션이 존재한다면 부모 트랜잭션에 합류, 그렇지 않다면 새로운 트랜잭션을 만든다. 중간에 자식/부모에서 rollback 이 발생된다면 자식과 부모 모두 rollback 한다.
REQUIRES_NEW	무조건 새로운 트랜잭션을 만든다. nested 한 방식으로 메소드 호출이 이루어지더라도 rollback 은 각각 이루어진다.
MANDATORY	무조건 부모 트랜잭션에 합류시킨다. 부모 트랜잭션이 존재하지 않는다면 예외를 발생시킨다.
SUPPORTS	메소드가 트랜잭션을 필요로 하지는 않지만, 진행 중인 트랜잭션이 존재하면 트랜잭션을 사용한다는 것을 의미한다. 진행 중인 트랜잭션이 존재하지 않더라도 메소드는 정상적으로 동작한다.
NESTED	부모 트랜잭션이 존재하면 부모 트랜잭션에 중첩시키고, 부모 트랜잭션이 존재하지 않는다면 새로운 트랜잭션을 생성한다. 부모 트랜잭션에 예외가 발생하면 자식 트랜잭션도 rollback 한다. 자식 트랜잭션에 예외가 발생하더라도 부모 트랜잭션은 rollback 하지 않는다. 이때 롤백은 부모 트랜잭션에서 자식 트랜잭션을 호출하는 지점까지만 롤백된다. 이후 부모 트랜잭션에서 문제가 없으면 부모 트랜잭션은 끝까지 commit 된다. 현재 트랜잭션이 있으면 중첩 트랜잭션 내에서 실행하고, 그렇지 않으면 REQUIRED 처럼 동작합니다.
NEVER	메소드가 트랜잭션을 필요로 하지 않는다. 만약 진행 중인 트랜잭션이 존재하면 익셉션이 발생한다.

REQUIRED

- Transaction 이 없는 경우에는 새롭게 생성을 한다.
- Method 2 가 실행이 되는 경우에는 Transaction 이 생성된 것을 사용한다.

REQUIRES_NEW

- Method 2 에서 새로운 Transaction 을 생성을 한다.

MANDATORY

- 반드시 Transaction 이 있어야 된다, 없으면 에러가 발생이 된다.

SUPPORTS

- 기존 Transaction 을 유지를 하고, 없으면 Non-Transaction 상태로 실행한다.

NESTED

- Transaction 있는 경우 해당하는 Transaction 을 활용한다. (= required)

NEVER

- Transaction 없이 실행한다. 있는 경우에는 에러가 발생이 된다.

NOT_SUPPORTED

- Transaction 없이 실행한다. 있는 경우에 해당하는 Transaction 을 중지하고 실행한다.