@Transactional

생성 일시	@2024년 7월 28일 오후 3:36
카테고리	가이드

1. 개요

Spring 트랜잭션을 올바르게 구성하는 방법과 *@Transactional* 을 잘 사용하는 방법 및 잘못된 사용 방법에 대해 알아보자.

2. 트랜잭션 설정

@Configuration 클래스에 ***@EnableTransactionManagement*** 를 사용하여 트랜잭션을 활성화 할 수 있다.

```
@Configuration
@EnableTransactionManagement
class PersistenceJPAConfig {

    @Bean
    fun entityManagerFactory(): LocalContainerEntityManagerFactoryBean {
        // ...
    }

    @Bean
    fun transactionManager(): PlatformTransactionManager {
        val transactionManager = JpaTransactionManager()
        transactionManager.setEntityManager(entityManagerFactory().getObject())
        return transactionManager
    }
}
```

- spring-data-* 또는 spring-tx 의존이 있는 경우에는 기본적으로 트랜잭션이 활성화된다.

3. *@Transactional* 어노테이션

클래스나 메서드에 *@Transactional* 어노테이션을 정의할 수 있다.

```
@Service
@Transactional
class FooService {
    // ...
}
```

- 다음과 같은 설정도 지원한다.
    - *Propagation Type :* 전파 유형

```
@Transactional(propagation = Propagation.REQUIRED)
```

- *REQUIRED* : 기본 값. 현재 트랜잭션이 존재하면 그 트랜잭션에 참여, 존재하지 않으면 새로운 트랜잭션을 시작
- *REQUIRES_NEW* : 항상 새로운 트랜잭션을 시작. 현재 트랜잭션이 존재하면 일시 중단.
- *SUPPORTS* : 현재 트랜잭션이 존재하면 그 트랜잭션에 참여, 존재하지 않으면 트랜잭션 없이 실행
- *NOT_SUPPORTS* : 트랜잭션이 존재하면 일시 중단하고 트랜잭션 없이 실행
- *MANDATORY* : 반드시 트랜잭션 내에서 실행되어야 함. 트랜잭션이 없으면 예외
- *NEVER* : 트랜잭션이 없어야 실행. 트랜잭션이 존재하면 예외
- *NESTED* : 현재 트랜잭션이 존재하면 중첩된 트랜잭션을 시작, 존재하지 않으면 새로운 트랜잭션을 시작.

- *Isolation Level :* 격리 수준

```
@Transactional(propagation = Propagation.REQUIRED)
```

- *DEFAULT* : 데이터베이스의 격리 수준을 따름
- *READ_UNCOMMITTED* : 다른 트랜잭션에서 커밋되지 않은 변경 내용도 읽음
- *READ_COMMITTED* : 다른 트랜잭션에서 커밋된 데이터만 읽음
- *REPEATABLE_READ* : 트랜잭션 동안 읽은 데이터가 변경되지 않도록 보장
- *SERIALIZABLE* : 가능 높은 격리 수준으로, 완벽한 읽기 일관성 보장

- *Timeout* : 시간 초과

```
@Transactional(timeout = 5) // 5초
```

- *readOnly* : 읽기 전용

```
@Transactional(readOnly = true)
```

- *Rollback* : 롤백 규칙

```
// 롤백할 예외를 지정할 때 사용
@Transactional(rollbackFor = Exception.class)

// 롤백하지 않을 예외를 지정할 때 사용
@Transactional(noRollbackFor = IllegalArgumentException.class)
```

- 특정 예외가 발생할 때 트랜잭션을 롤백하도록 설정

4. 잘못된 사용

4.1. 트랜잭션과 프록시

Spring은 *@Transactional* 이 붙은 모든 클래스에 대한 프록시를 생성한다. Spring은 프록시를 활용해 트랜잭션을 시작하고 커밋하기 위해 메서드 실행 전후에 트랜잭션 로직을 실행한다.

위와 같은 특성으로 인해 @Transactional 을 사용할 때 두 가지의 주의할 점이 있다.

첫 번째는 클래스 내에서 자기 자신의 메서드를 호출하는 경우에는 *@Transactional* 이 정상 동작하지 않는 것이다. *@Transactional* 이 붙은 빈은 트랜잭션 관리를 위해 Spring이 자동으로 생성하는 프록시 객체를 통해 호출되는데 내부 호출은 프록시를 거치지 않기 때문에 트랜잭션이 동작하지 않는 것이다.

즉, 다음과 같이 호출할 경우 @Transactional 이 동작하지 않게 된다.

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class TransactionalService {

    @Transactional
    public void methodA() {
        // Do something that requires a transaction
        System.out.println("methodA - Transaction started");
        methodB(); // Self-invocation
    }

    @Transactional
    public void methodB() {
        // Do something else that requires a transaction
        System.out.println("methodB - Transaction started");
    }
}
```

두 번째로 주의할 점은 *public* 메서드에만 *@Transactional* 을 붙어야 한다는 것이다. 다른 가시성에 대해서는 프록시가 동작하지 않으므로 *@Transactional* 이 무시된다.

4.2. Isolation Level 변경

다음과 같이 트랜잭션의 isolation level을 변경할 수 있다.

```
@Transactional(isolation = Isolation.SERIALIZABLE)
```

- 이는 Spring 4.1에서 도입되어 이전 버전에서는 예외가 발생한다.

4.3. Read-Only 트랜잭션

다음과 같이 트랜잭션에 Read-Only를 설정할 수 있다.

```
@Transactional(readOnly = true)
```

- 주의할 점은 *readOnly*가 true로 설정되어 있어도 삽입이나 업데이터가 발생하지 않을 것이라고 확신할 수 없다는 것이다. 이는 구현체에 따라 다르다.

readOnly 는 트랜잭션 내부에서만 동작하므로 트랜잭션 컨텍스트 외부에서 작업이 발생하면 *readOnly*가 무시된다.

```
@Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
```

- 이처럼 트랜잭션 컨텍스트가 아닐 경우에는 readOnly가 무시될 수 있다.

4.4. 트랜잭션 로깅

트랜잭션 관련 문제를 디버깅 할 때 유용한 방법은 트랜잭션 패키지에 대한 로깅을 수정하는 것이다.

패키지는 “*org.springframework.transaction*” 이며, *TRACE* 로 로깅 레벨을 설정하면 된다.

5. 동작 원리

https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth

Reference

https://www.baeldung.com/transaction-configuration-with-jpa-and-spring