Chapter 09 SPRING의 Transaction 관리

: Database 프로그래밍을 하는데 있어서 Transaction은 매우 중요하다. Transaction이 보장되지 않으면 서비스를 올바르게 제공할 수 없기 때문이다.

예를 들어, 항공 예약, 호텔 예약, 렌터카 예약을 제공하는 여행 상품을 생각해 보자. 상품을 구매할 경우 항공과 호텔은 예약이 성공적으로 되었는데

렌터카 예약을 실패한 경우 상품 구매가 정상적으로 이루어지면 안 될 것이다. 이런 경우 세 가지 예약이 모두 성공적으로 이루어진 경우에만 상품 구매가

정상적으로 처리되어야 할 것이다.

Transaction은 이렇게 성공적으로 처리되거나 또는 하나라도 실패하면 완전히 실패 처리를 해야 하는 경우에 사용된다. SPRING은 Database 연동뿐만 아니라

Transaction 관리 기능을 지원하고 있기 때문에, SPRING을 사용하면 간단한 설정만으로 단일 Resource과 다중 Resource에 대한 Transaction을 처리할 수 있다. 본 장에서는

Transaction이 무엇인지 간단하게 살펴보고, SPRING이 제공하는 Transaction 관리 방법-Code 기반 처리, 선언적 Transaction, Annotation 기반-을 살펴볼 것이다.

1. Transaction이란

: 인터넷 서점 Site에서 도서를 구매할 경우 다음과 같은 순서로 작업이 진행될 것이다.

- 결재를 수행한다.

- 결재 내역을 저장한다.

- 구매 내역을 저장한다.

위의 과정은 도서 구매시 반드시 성공적으로 이루어져야 한다. 한 가지 과정이라도 실패할 경우 반드시 모든 과정이 취소되어야 한다. 예를 들어, 결재 내역 저장까지는

성공했는데, 구매 내역을 저장하는 과정이 실패했다고 하자. 이때, 전 과정이 취소되지 않는다면 구매자는 결재만 하고 구매는 하지 않은 것처럼 될 것이다.

Transaction은 여러 과정을 하나의 행위로 묶을 때 사용된다. Transaction은 Transaction 범위내에 있는 처리 과정 중 한 가지라도 실패할 경우 전체 과정을 취소시킴으로써

데이터의 무결성을 보장한다.

1.1 ACID

: 트랜 잭션을 설명할 때에는 보통 네 가지의 특징인 ACID를 이용한다. ACID는 다음과 같다.

- Atomic : Transaction은 한 개 이상의 동작을 논리적으로 한 개의 작업 단위(unit of work)로 묶는다. 원자성은 Transaction 범위에 있는 모든 동작이 모두 실행되거나

또는 모두 실행이 취소됨을 보장한다. 모든 동작이 성공적으로 실행되면 Transaction은 성공한다. 만 약 하나라도 실패하면 Transaction은 실패하고 모든과정은

롤백한다.

- Consistent : Transaction이 종료되면, 시스템은 비즈니스에서 기대하는 상태가 된다. 예를 들어, 서적 구매 Transaction이 성공적으로 실행되면 결재 내역, 구매 내역,

잔고 정보가 비즈니스에 맞게 저장되고 변경된다.

- Isolated : Transaction은 다른 Transaction과 독립적으로 실행되어야 하며, 서로 다른 Transaction이 동일한 데이터에 동시에 접근할 경우 알맞게 동시 접근을 제어해야

한다. (동시 접근 제어는 설정한 격리 레벨에 따라 달라진다.)

- Durable : Transaction이 완료되면, 그 결과는 지속적으로 유지되어야 한다. 현재의 Application이 변경되거나 없어지더라도 데이터는 유지된다.

일반적으로 물리적인 저장소를 통해서 Transaction 결과가 저장된다.

예를 들어, 도서 구매의 경우 결재 처리, 구매 내역 처리 등이 하나의 작업으로 처리되어야 한다. 만약, 이 중 하나라도 실패하면 전체 구매 과정이 취소된다. 또한,

Transaction이 성공적으로 실행되면 결제 정보와 구매 내역 정보가 반드시 시스템에 기록되어야 한다. 또한, 동일한 서적에 대한 구매 Transaction이 동시에 한 개 이상

실행될 경우, 잔고 정보가 알맞게 처리되어야 하므로 각 Transaction이 알맞은 순서로 실행되어야 한다. 또한, 시스템이 일시적으로 정지되더라도 구매 정보는 사라져서는

안되며, 다시 시스템이 구동될 때 사용 가능해야 한다.

2. SPRING의 Transaction 지원

: SPRING은 Code 기반의 Transaction 처리(Programmatic Transaction)뿐만 아니라 선언적 Transaction(Declarative Transaction)을 지원하고 있다. 따라서, 개발자가 직접적으로

Transaction의 범위를 Code 수준에서 정의하고 싶은 경우에는 SPRING이 제공하는 Transaction 템플릿 Class를 이용해서 손쉽게 Transaction 범위를 지정할 수 있다.

또한, 설정 파일이나 Annotation을 이용해서 Transaction의 범위 및 규칙을 정의할 수 있기 때문에 Transaction을 매우 쉽게 관리할 수 있다.

SPRING은 Database 연동 기술에 상관 없이 동일한 방식으로 Transaction을 처리 할 수 있도록 하고 있다. 예를 들어, Hibernate를 사용하거나 iBatis를 사용하거나

JPA를 사용하거나 또는 JTA를 이용해서 Transaction을 처리하는 지의 여부에 상관없이 SPRING은 동일한 Code를 이용해서 Transaction을 관리할 수 있도록 지원한다.

2.1 SPRING의 PlatformTransactionManager 설정

: SPRING은 PlatformTransactionManager Interface를 이용해서 Transaction 처리를 추상화 하였고, 데이터 베이스 연동 기술에 따라 알맞은 PlatformTransactionManager

구현 Class를 제공하고 있다. 다음은 주요 구현 Class 간의 관계를 보여 주고 있다.

<<Interface>>

PlatformTransactionManager

||

AbstractPlatformTransactionManager

|| || || ||

DataSourceTransactionManager || || JpaTransactionManager

|| ||

JtaTransactionManager HibernateTransactionManager

|| ||

WeblogicJtaTransactionManager WebSphereUowTransactionManager

각각의 Transaction 관리자 구현 Class는 관련된 Database 기술에 따라 알맞은 Transaction 처리를 수행한다. 예를 들어, HibernateTransactionManager의 경우

내부적으로 Hibernate의 Transaction을 이용해서 Transaction을 처리한다. 아래와 같이 Transaction 관련 처리를 HibernateTransactionManager에 요청하면,

HibernateTransactionManager는 그 요청을 Hibernate의 Transaction에 전파함으로써 Transaction을 처리하게 된다.

Client ==1:commit()==> :HibernateTransactionManager ==2:commit()==> :Transaction

Code에서 직접적으로 PlatformTransactionManager를 사용해서 Transaction을 처리 할 수도 있지만, 그것 보다는 TransactionTemplate Class나 선언적 방식을 이용해서

Transaction을 처리하게 된다.

Transaction을 처리하려면 Database 연동을 위한 구현 기술에 알맞은 Transaction 관리자를 등록해 주어야 하는데, 본 절에서는 주요 PlatformTransactionManager의

구현 Class를 설정하는 방법에 대해서 살펴보도록 하겠다.

2.2 JDBC 기반 Transaction 관리자 설정

: JDBC나 iBatis와 같이 JDBC를 이용해서 Database 연동을 처리하는 경우, 다음과 같이 DataSourceTransactionManager를 Transaction 관리자로 사용한다.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"

p:driverClassName="oracle.jdbc.driver.OracleDriver"

p:url="jdbc:oracle:thin:@localhost:1521:ORCL"

p:username="nals"

p:password="dkagh1234."/>

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager"

p:dataSource-ref="dataSource" />

DataSourceTransactionManager는 dataSource 프로퍼티를 통해서 전달받은 DataSource로 부터 Connection을 가져온 뒤, Connection의 commit(), rollback()등의

메서드를 사용해서 Transaction을 관리한다.

2.3 Hibernate Transaction 관리자 설정

: Hibernate를 사용하는 경우에는 HibernateTransactionManager를 Transaction 관리자로 사용한다. 아래 Code는 설정 예이다.

<bean id="transactionManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager"

p:sessionFactory-ref="sessionFactory" />

<bean id="sessionFactory"

class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"/>

<property name="mappingResources">

...

</property>

<property name="hibernateProperties">

...

</property>

</bean>

HibernateTransactionManager는 sessionFactory 프로퍼티를 통해서 전달받은 Hibernate Transaction을 생성한 뒤, Transaction을 이용해서 Transaction을 관리한다.

2.4 JPA Transaction 관리자 설정

: JPA를 사용할 경우에는 JpaTransactionManager를 Transaction 관리자로 사용하면 된다. 아래 Code는 설정 예이다.

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

...

</bean>

<bean id="transactionManager"

class="org.springframework.orm.jpa.JpaTransactionManager"

p:entityManagerFactory="entityManagerFactory" />

JpaTransactionManager는 entityManagerFactory 프로퍼티를 통해 전달받은 EntityManagerFactory를 이용해서 Transaction을 관리한다.

2.5 JTA Transaction 관리자 설정

: 다중 Resource에 접근하는 경우 JTA(Java Transaction API)를 이용해서 Transaction을 처리하게 되는데, 이 경우에는 JtaTransactionManager를 사용한다.

JtaTransactionManager는 아래 Code와 같이 transactionManagerName 프로퍼티를 이용해서 JtaTransactionManager를 구할 수 있는 JNDI 이름을 설정한다.

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"

p:transactionManagerName="java:comp/TransactionManager"/>

Container가 제공하는 TransactionManager를 사용하지 않고, TransactionEssentials과 같은 오픈 Source Library를 이용해서 로컬 JTA를 사용할 경우에는,

userTransaction 프로퍼티를 이용해서 UserTransaction을 직접 설정할 수도 있다.

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"

depends-on="userTransactionService">

<property name="transactionManager" ref="atomikosTransactionManager" />

<property name="userTransaction" ref="atomikosUserTransaction" />

</bean>

<bean id="userTransactionService" class="com.atomikos.icatch.config.UserTransactionServiceImpl"

init-method="init" destroy-method="shutdownForce">

...

</bean>

<bean id="atomikosTransactionManager" class="com.atomikos.icatch.jta.UserTransactionManager"

init-method="init" destroy-method="close"

depends-on="userTransactionService">

<property name="forceShutdown" value="false"/>

</bean>

<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImpl"

depends-on="userTransactionService">

<property name="transactionTimeout" value="300"/>

</bean>

JtaTransactionManager는 javax.transaction.UserTransaction의 commit(), rollback()등을 이용해서 Transaction을 처리한다.

3. TransactionTemplate을 이용한 Transaction 처리

: 상품 구매를 처리하는 Code가 아래와 같다고 하자.

public class PlaceOrderServiceImpl implements PlaceOrderService {

@Override

public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)

throws ItemNotFoundException {

Item item = itemDao.findById(orderRequest.getItemId());

if (item == null)

throw new ItemNotFoundException(orderRequest.getItemId());

PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());

paymentInfoDao.insert(paymentInfo);

PurchaseOrder order = new PurchaseOrder(item.getId(), orderRequest

.getAddress(), paymentInfo.getId());

purchaseOrderDao.insert(order);

return new PurchaseOrderResult(item, paymentInfo, order);

}

}

위 Code는 다음과 같은 순서로 상품 구매를 처리한다.

- 첫째, itemDao로부터 구매하려는 상품 정보를 구한다. 존재하지 않을 경우 예외를 발생시킨다.

- 둘째, paymentInfoDao를 이용해서 결재 정보(PaymentInfo)를 Database에 삽입한다.

- 셋째, purchaseOrderDao를 이용해서 구매 정보(PurchaseOrder)를 Database에 삽입한다.

위 과정은 하나의 Transaction에서 처리되어야 한다. 만약, 세 번째 과정이 실패하면 이전에 실행된 모든 과정이 롤백 되어야 할 것이다. 그렇지 않을 경우,

구매 주문은 올바르게 되지 않았는데 결재 처리는 되 버리고 말 것이다.

SPRING에서 하나의 Transaction에서 처리하는 첫 번째 방법은 TransactionTemplate Class를 이용하는 것이다. DAO를 위한 템플릿 Class와 마찬가지로

TransactionTemplate Class는 Transaction과 관련된 작업(Transaction 시작, 커밋, 롤백)을 처리해 주는 템플릿 Class이다.

TransactionTemplate을 사용하려면 먼저 TransactionTemplate을 빈으로 설정해 주어야 한다. TransactionTemplate을 설정할 때는 다음과 같이

transactionManager 프로퍼티에 SPRING이 PlatformTransactionManager 빈을 지정하면 된다.

<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate"

p:transactionManager-ref="transactionManager"/>

<bean id="placeOrderService" class="madvirus.spring.store.domain.PlaceOrderServiceTransactionTemplateImpl"

p:transactionTemplate-ref="transactionTemplate"/>

TransactionTemplate 빈을 설정했다면, TransactionTemplate Class를 이용해서 Transaction을 처리할 수 있다. 아래 Code는 TransactionTemplate Class의

설정 예이다.

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.TransactionCallback;

import org.springframework.transaction.support.TransactionTemplate;

public class PlaceOrderServiceTxTemplateImpl implements PlaceOrderService {

private ItemDao itemDao;

private PaymentInfoDao paymentInfoDao;

private PurchaseOrderDao purchaseOrderDao;

private TransactionTemplate transactionTemplate;

...

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {

this.transactionTemplate = transactionTemplate;

}

@Override

public PurchaseOrderResult order(final PurchaseOrderRequest orderRequest)

throws ItemNotFoundException {

return transactionTemplate

.execute(new TransactionCallback<PurchaseOrderResult>() {

@Override

public PurchaseOrderResult doInTransaction(

TransactionStatus status) {

Item item = itemDao.findById(orderRequest.getItemId());

if (item == null)

throw new ItemNotFoundException(orderRequest

.getItemId());

PaymentInfo paymentInfo = new PaymentInfo(item

.getPrice());

paymentInfoDao.insert(paymentInfo);

PurchaseOrder order = new PurchaseOrder(item.getId(),

orderRequest.getAddress(), paymentInfo.getId());

purchaseOrderDao.insert(order);

return new PurchaseOrderResult(item, paymentInfo, order);

}

});

}

}

TransactionTemplate.execute() 메서드는 Transaction을 시작한 뒤에 파라미터로 전달받은 TransactionCallback 구현 객체의 doInTransaction() 메서드를 호출한다.

doInTransaction() 메서드의 실행이 종료되면 Transaction을 커밋한다. 만약 doInTransaction() 메서드에서 RuntimeException이나 Error를 발생시키면

Transaction을 롤백한다.

일반적으로 TransactionTemplate Class를 사용할 때에는 TransactionCallback Interface를 구현한 Class를 작성하기 보다는 위 Code와 같이 임의 Class를

사용해서 TransactionCallback 객체를 TransactionTemplate.execute() 메서드에 전달한다.

TransactionCallback.doInTransaction() 메서드는 다음과 같이 정의되어 있다.

public interface TransactionCallback<T>{

T doInTransaction(TransactionStatus status);

}

TransactionTemplate.execute() 메서드는 TransactionCallback.doInTransaction() 메서드가 리턴하는 객체를 리턴한다.

doInTransaction() 메서드는 throws를 통해서 발생시킬 수 있는 예외를 설정하고 있지 않기 때문에, doInTransaction() 메서드 내부에서는 RuntimeException이나

Error 타입의 예외만 발생시킬 수 있다. 만약 doInTransaction 매부에서 반드시 catch로 처리해 주어야 하는 예외를 발생시킨다면, 다음과 같이 doInTransaction()

메서드 내부에서 try~catch 블럭을 사용해서 예외를 처리한 뒤 롤백 여부를 설정해 주어야 한다.

@Override

public PurchaseOrderResult order(final PurchaseOrderRequest orderRequest)

throws ItemNotFoundException {

return transactionTemplate

.execute(new TransactionCallback<PurchaseOrderResult>() {

@Override

public PurchaseOrderResult doInTransaction(TransactionStatus status) {

try{

Item item = itemDao.findById(orderRequest.getItemId());

if (item == null)

throw new ItemNotFoundException(orderRequest

.getItemId());

PaymentInfo paymentInfo = new PaymentInfo(item

.getPrice());

paymentInfoDao.insert(paymentInfo);

PurchaseOrder order = new PurchaseOrder(item.getId(),

orderRequest.getAddress(), paymentInfo.getId());

purchaseOrderDao.insert(order);

return new PurchaseOrderResult(item, paymentInfo, order);

}catch(Exception e){

status.setRollbackOnly();

return ex;

}

}

});

}

TransactionStatus.setRollbackOnly() 메서드를 호출하면, TransactionTemplate은 예외 발생 여부에 상관없이 Transaction을 롤백한다.

4. 선언적 Transaction 처리

: 선언적 Transaction(Declarative Transaction)은 TransactionTemplate과 달리 Transaction 처리를 Code에서 직접적으로 수행하지 않는다. 대신, 선언적 Transaction은

설정 파일이나 Annotation을 이용해서 Transaction의 범위, 롤백 규칙등을 정의하게 된다. 선언적 Transaction은 다음과 같은 두 가지 방식으로 정의할 수 있다.

- <tx:advice> 태그를 이용한 Transaction 처리

- @Transactional Annotation을 이용한 Transaction 설정

이 두 가지 Transaction 설정 방법에 대해 차례대로 살펴보도록 하자.

4.1 <tx:advice> 태그를 이용한 Transaction 처리

: <tx:advice> 태그는 SPRING 2.0 버전부터 지원하는 확장 태그이다. <tx:advice> 태그를 이용해서 Transaction 속성을 정의하기 위해서는 먼저 tx 네임스페이스를

추가해 주어야 한다.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:p="http://www.springframework.org/schema/p" xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager"

p:dataSource-ref="dataSource" />

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="order" propagation="REQUIRED" />

<tx:method name="get" read-only="true" />

</tx:attributes>

</tx:advice>

...

</beans>

tx 네임스페이스를 <beans> 태그에 추가해 주었다면, <tx:advice> 태그, <tx:attributes> 태그, 그리고 <tx:method> 태그를 이용해서 Transaction

속성을 정의할 수 있다.

먼저, <tx:advice> 태그는 Transaction을 적용할 때 사용될 Advisor를 생성한다. id 속성은 생성될 Transaction Advisor의 식별 값을 입력하며,

transaction-manager 속성에는 SPRING의 PlatformTransactionManager 빈을 설정한다.

<tx:method> 태그는 <tx:attributes> 태그의 자식 태그로 설정한다. <tx:method> 태그는 Transaction을 설정할 메서드 및 Transaction 속성을 설정한다.

<tx:method>의 속성 및 설정은 아래 표와 같다.

속성 이름 설 명

=================================================================================

name Transaction이 적용될 메서드 이름을 명시한다. '\*'을 사용한 설정이 가능하다. 예를 들어, "get\*"으로

설정할 경우 이름이 get으로 시작하는 메서드를 의미한다.

=================================================================================

propagation Transaction 전파 규칙을 설정한다.

=================================================================================

isolation Transaction 격리 레벨을 설정한다.

=================================================================================

read-only 읽기 전용 여부를 설정한다.

=================================================================================

no-rollabck-for Transaction을 롤백하지 않을 예외 타입을 설정한다.

=================================================================================

rollback-for Transaction을 롤백할 예외 타입을 설정한다.

=================================================================================

timeout Transaction의 타임 아웃 시간을 초 단위로 설정한다.

=================================================================================

<tx:advice> 태그는 Advisor만 생성하는 것이지 실제로 Transaction을 적용하는 것은 아니다. 실제로 Transaction을 적용하는 것은 AOP를 통해서 이루어진다.

아래 Code는 설정 예이다.

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="order" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="execution(public \* madvirus.spring..\*Service.\*(..))"

id="servicePublicMethod" />

<aop:advisor advice-ref="txAdvice" pointcut-ref="servicePublicMethod" />

</aop:config>

위 Code에서 <aop:config> 태그를 이용해서 madvirus.spring 패키지의 하위 패키지에 있는 \*Service의 public 메서드에 <tx:advice>로 설정한 Transaction

Advisor를 적용하도록 설정하였다. <tx:method> 태그에서 이름이 "order"인 메서드에 대해 Transaction을 적용하도록 설정했으므로, 실제로는 \*Service의

public 메서드 중에서 order 메서드를 호출할 때에 Transaction이 적용된다.

(1) <tx:method> 태그의 propagation 속성과 전파 규칙

: <tx:method> 태그에서 중요한 속성중의 하나가 propagation 속성이다. propagation 속성은 Transaction의 전파 규칙을 정의할 때 사용되며,

propagation 속성에 올 수 있는 값은 다음 표와 같다.

속성 값 설 명

==========================================================================================

REQUIRED 메서드를 수행하는데 Transaction이 필요하다는 것을 의미한다. 현재 진행 중인 Transaction이 존재하면, 해당 Transaction을

(기본 값) 사용한다. 존재하지 않는다면 새로운 Transaction을 생성한다.

==========================================================================================

MANDATORY 메서드를 수행하는데 Transaction이 필요하다는 것을 의미한다. 하지만, REQUIRED와 달리, 진행중인 Transaction이

존재하지 않을 경우 예외를 발생시킨다.

==========================================================================================

REQUIRES\_NEW 항상 새로운 Transaction을 시작한다. 기존 Transaction이 존재하면 기존 Transaction을 일시 중지하고 새로운 Transaction을

시작한다. 새로 시작된 Transaction이 종료된 뒤에 기존 Transaction이 계속된다.

==========================================================================================

SUPPORTS 메서드가 Transaction을 필요로 하지는 않지만, 기존 Transaction이 존재할 경우 Transaction을 사용한다는 것을 의미한다.

진행중인 Transaction이 존재하지 않더라도 메서드는 정상적으로 동작한다.

==========================================================================================

NOT\_SUPPORTED 메서드가 Transaction을 필요로 하지 않음을 의미한다. SUPPORTS와 달리 진행중인 Transaction이 존재할 경우 메서드가

실행되는 동안 Transaction은 일시 중지되며, 메서드 실행이 종료된 후에 Transaction을 계속 진행한다.

==========================================================================================

NEVER 메서드가 Transaction을 필요로 하지 않으며, 만약 진행중인 Transaction이 존재하면 예외를 발생시킨다.

==========================================================================================

NESTED 기존 Transaction이 존재하면, 기존 Transaction에 중첩된 Transaction에서 메서드를 실행한다. 기존 Transaction이 존재하지

않으면 REQUIRED와 동일하게 동작한다. 이 기능은 JDBC 3.0 드라이버를 사용할 때에만 적용된다.

(JTA Provider가 이 기능을 지원할 경우에도 사용 가능하다.)

==========================================================================================

선언적 Transaction을 하는 경우 별도 처리가 필요하지 않다면 기본값인 "REQUIRED"를 Transaction 전파 규칙으로 사용한다.

(2) <tx:method> 태그의 rollback-for 속성과 no-rollback-for 속성을 통한 롤백 처리

: SPRING Transaction은 기본적으로 RuntimeException 및 Error에 대해서만 롤백 처리를 수행한다. 떠라서, Throwable이나 Exception 타입의 예외가 발생하더라도

롤백되지 않고, 예외가 발생되기 전까지의 작업이 커밋된다.

예외 발생시 Transaction의 롤백 규칙을 좀 더 정교하게 정의하고 싶다면 rollback-for 속성과 no-rollback-for 속성을 사용하면 된다. rollback-for 속성은

예외 발생시 롤백 작업을 수행할 예외 타입을 설정하며, no-rollback-for 속성은 예외가 발생하더라도 롤백하지 않을 예외 타입을 설정한다.

명시할 예외 타입이 한 개 이상인 경우 각각의 예외는 콤마로 구분한다. 예외 Class는 완전한 이름을 입력하거나 또는 패키지 이름을 제외한 Class

이름만을 입력해도 된다.

Exception 및 하위 타입의 예외가 발생할 경우 롤백 작업을 수행하고, MemberNotFoundException이 발생하는 경우에는 롤백 작업을 수행하지 않는다고 하자.

이 경우 다음과 같이 설정할 수 있다.

<tx:method name="regist" rollback-for="Exception" no-rollback-for="MemberNotFoundException"/>

SPRING은 예외 발생시 좀더 정확하게 메칭되는 타입을 기준으로 롤백 규칙을 적용한다. 따라서, MemberNotFoundException이 Exception Class를 상속

받았다고 하더라도 MemberNotFoundException이 발생하면 롤백을 수행하지 않게 된다.

(3) <tx:method> 태그의 isolation 속성과 Transaction 격리 레벨

: Transaction 격리 레벨은 isolation 속성을 사용해서 설정하며, isolation 속성에서 사용할 수 있는 값은 아래 표와 같다.

속성 값 설 명

======================================================================

DEFAULT 기본 설정을 사용한다.

======================================================================

READ\_UNCOMMITED 다른 Transaction에서 커밋하지 않은 데이터를 읽을 수 있다.

======================================================================

READ\_COMMITED 다른 Transaction에 의해 커밋된 데이터를 읽을 수 있다.

======================================================================

REPEATABLE\_READ 처음에 읽어온 데이터와 두 번째 읽어온 데이터가 동일한 값을 갖는다.

======================================================================

SERIALIZABLE 동일한 데이터에 대해서 동시에 두 개 이상의 Transaction이 수행될 수 없다.

======================================================================

4.2 Annotation 기잔 Transaction 설정

: @Transactional Annotation을 사용해서 Transaction을 설정할 수도 있다. @Transactional Annotation은 다음 Code와 같이 메서드나 Class에 적용되며

관련 Transaction 속성을 설정한다.

import org.springframework.transaction.annotation.Transactional;

public class PlaceOrderServiceAnnotImpl implements PlaceOrderService {

...

@Override

@Transactional

public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)

throws ItemNotFoundException {

Item item = itemDao.findById(orderRequest.getItemId());

if (item == null)

throw new ItemNotFoundException(orderRequest.getItemId());

PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());

paymentInfoDao.insert(paymentInfo);

PurchaseOrder order = new PurchaseOrder(item.getId(), orderRequest

.getAddress(), paymentInfo.getId());

purchaseOrderDao.insert(order);

return new PurchaseOrderResult(item, paymentInfo, order);

}

}

@Transactional Annotation은 propagation 속성을 비롯하여 아래 표에 표시한 속성을 이용해서 Transaction 속성을 정의한다.

속 성 설 명

===============================================================================

propagation Transaction 전파 규칙을 설정한다. Propagation 열거형 타입에 값이 정의되어 있다.

기본 값은 Propagation.REQUIRED이다.

===============================================================================

isolation Transaction 격리 레벨을 설정한다. Isolation 열거형 타입에 값이 정의되어 있다.

===============================================================================

readOnly 읽기 전용 여부를 설정한다. boolean 값을 설정하며, 기본 값은 false이다.

===============================================================================

rollbackFor Transaction을 롤백할 예외 타입을 설정한다. 예, rollback-For={Exception.class}

===============================================================================

noRollbackFor Transaction을 롤백하지 않을 예외 타입을 설정한다.

예, noRollbackFor={ItemNotFoundException.class}

===============================================================================

timeout Transaction의 타임아웃 시간을 초 단위로 설정한다.

===============================================================================

@Transactional Annotation이 적용된 SPRING 빈에 실제로 Transaction을 적용하려면 다음과 같이 <tx:transaction-driven> 태그를 설정하면 된다.

<tx:transaction-driven transaction-manager="transactionManager"/>

<bean id="placeOrderService" class="madvirus.spring.chap09.store.domain.PlaceOrderServiceAnnotImpl"

..../>

<tx:annotation-driven> 태그가 제공하는 속성은 다음과 같다.

속 성 설 명 기 본 값

===========================================================================

transaction-manager 사용할 PlatformTransactionManager 빈의 이름 transactionManager

===========================================================================

proxy-target-class Class에 대해서 프록시를 생성할지의 여부, true일 경우 false

CGLIB를 이용해서 프록시를 생성하며, false인 경우 Java

다이나믹 프록시를 이용해서 프록시를 생성한다.

===========================================================================

order Advice 적용 순서. int의 최대 값.

(가장 낮은 순위)

===========================================================================

<tx:annotation-driven> 태그 대신 PersistenceAnnotationBeanPostProcessor Class를 빈으로 등록해도 된다.

<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

5. TransactionsEssentials을 이용한 분산 Transaction 처리

: 두 개 이상의 Resource에 동시에 접근하는데 Transaction이 필요한 경우가 있다. 예를 들어, 결재 정보를 저장하는 Database와 구매 내역을 저장하는

Database가 다르다고 하자. 이 경우 두 Database에 접근하기 위한 DataSource는 서로 다르지만, 두 Database에 접근하는 Code는

단일 Transaction으로 처리되어야 한다.

Java에서 분산 Transaction을 처리하기 위해서는 분산 Transaction 서비스를 제공해주는 Transaction 관리자가 필요하다. Weblogic이나 JBoss 같은 Container는

자체적으로 분산 Transaction 서비스를 지원하고 있지만, 톰캣과 같은 Servlet Container는 분산 Transaction을 지원하고 있지 않다.

단위 Test를 수행할 때 Container 없이 분산 Transaction을 Test해야 한다거나 톰캣과 같이 분산 Transaction 서비스를 지원하지 않는 Container에서

분산 Transaction을 구현해야 한다면, TransactionEssentials나 Jencks와 같은 Transaction 매니저를 이용하면 된다. TransactionEssentials나 Jencks를

사용하면 JBoss와 같은 Container를 사용하지 않고도 분산 Transaction을 구현할 수 있는데, 본 장에서는 TransactionEssentials을 이용해서 분산 Transaction을

설정하는 방법을 살펴보도록 하자.

5.1 TransactionEssentials Download/설정

: TransactionEssentials은 Atomikos에서 개발한 ExtremeTransaction의 오픈Source 버전으로서 http://www.atomikos.com Site에서 Download 받을 수 있다.

Maven을 사용하고 있다면, 다음과 같이 Atomikos의 Maven 리포지토리 및 transaction-essentials-all에 대한 의존을 추가함으로써 TransactionEssentials를

사용할 수 있다. (참고로 아래 Code는 3.6.1 버전을 기준으로 작성된 것으로서 버전에 따라 차이가 있을 수 있다.)

<repositories>

<repository>

<id>atomikos</id>

<name> Atomikos Repository</name>

<url>http://repo.atomikos.com</url>

</repository>

</repositories>

<dependencies>

...

<dependency>

<groupId>com.atomikos</groupId>

<artifactId>transactions-essentials-all</artifactId>

<version>3.6.1</version>

<type>pom</type>

<exclusions>

<exclusion>

<groupId>com.atomikos</groupId>

<artifactId>transactions-jdbc-deprecated</artifactId>

</exclusion>

<exclusion>

<groupId>com.atomikos</groupId>

<artifactId>transactions-jms-deprecated</artifactId>

</exclusion>

<exclusion>

<groupId>com.atomikos</groupId>

<artifactId>transactions-hibernate3</artifactId>

</exclusion>

</exclusions>

<dependency>

</dependencies>

5.2 TransactionEssentials와 SPRING 연동

: TransactionEssentials에 대한 설정을 완료했다면, SPRING 설정 파일에 다음과 같은 정보를 추가해주면 된다.

- TransactionEssentials를 이용한 JtaTransactionManager 설정

- TransactionEssentials가 제공하는 Class를 이용한 XADataSource 설정

- XADataSource를 사용하도록 설정

(1) TransactionEssentials를 이용한 JtaTransactionManager 설정

: 먼저 TransactionEssentials를 이용하는 JtaTransactionManager를 설정하려면 아래와 같은 설정을 추가해 주어야 한다.

<bean id="userTransactionService" class="com.atomikos.icatch.config.UserTransactionServiceImp"

init-method="init" destroy-method="shutdownForce">

<constructor-arg>

<!-- 여기에 Atomikos 프로퍼티 위치 -->

<props>

<prop key="com.atomikos.icatch.service">

com.atomikos.icatch.standalone.UserTransactionServiceFactory

</prop>

</props>

</constructor-arg>

</bean>

<!-- Atomikos UserTransactionManager 생성. SPRING 설정에 필요함 -->

<bean id="atomikosTransactionManager" class="com.atomikos.icatch.jta.UserTransactionManager"

init-method="init" destroy-method="close" depends-on="userTransactionService">

<property name="forceShutdown" value="false" />

</bean>

<!-- Atomikos UserTransactionImp 생성. SPRING 설정에 필요함 -->

<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp"

depends-on="userTransactionService">

<property name="transactionTimeout" value="300" />

</bean>

<!-- Atomikos를 사용하는 SPRING JtaTransactionManager 설정 -->

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"

depends-on="userTransactionService">

<property name="transactionManager" ref="atomikosTransactionManager" />

<property name="userTransaction" ref="atomikosUserTransaction" />

</bean>

(2) TransactionEssentials가 제공하는 Class를 이용한 XADataSource 설정

: JtaTransactionManager 설정이 완료되었다면, 그 다음으로 할 작업은 TransactionEssentials가 제공하는 Class를 이용해서 XADataSource를

생성하는 것이다. TransactionEssentials는 다음의 두 가지 XADataSource를 제공하고 있다.

- com.atomikos.jdbc.AtomikosDataSourceBean : XA를 지원하는 JDBC 드라이버를 위한 DataSource 설정.

- com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean : XA를 지원하지 않는 JDBC 드라이버를 위한 DataSource 설정. 이 Class는 XA에

호환되지 않기 때문에 Transaction의 원자성(atomic)을 보장할 수 없다.

XA를 지원하는 JDBC 드라이버를 갖고 있다면, AtomikosDataSourceBean을 이용해서 DataSource를 설정해주면 된다. 아래 Code는 설정 예를 보여주고 있다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<property name="xaDataSourceClassName"

value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">root</prop>

<prop key="password"></prop>

<prop key="url">jdbc:mysql://localhost/z\_billing?useUnicode=true&amp;characterEncoding=euckr

</prop>

</props>

</property>

<property name="poolSize" value="1" />

</bean>

<bean id="storeDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="STOREDB" />

<property name="xaDataSourceClassName"

value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">root</prop>

<prop key="password"></prop>

<prop key="url">jdbc:mysql://localhost/z\_store?useUnicode=true&amp;characterEncoding=euckr

</prop>

</props>

</property>

<property name="poolSize" value="1" />

</bean>

AtomikosDataSourceBean Class를 사용할 때 기본 프로퍼티는 다음과 같다.

- uniqueResourceName : DataSource를 식별하는 고유 Resource 이름.

- xaDataSourceClassName : XADataSource 구현 Class의 완전한 이름.

- xaProperties : XADataSource를 설정할 때 필요한 <이름,값> 쌍.

사용하려는 XADataSource 구현 Class에 따라 xaProperties에 전달할 설정 값이 달라진다. 이에 대한 내용은 본 장의 '각 DBMS 별 AtomikosDataSourceBean

설정'에서 살펴보도록 하겠다.

JDBC 드라이버가 XA를 지원하지 않는 경우에는 AtomikosNonXADataSourceBean Class를 사용해서 DataSource를 설정하면 된다. 아래 Code는 설정 예이다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosNonXADataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<property name="user" value="sa"/>

<property name="password" value=""/>

<property name="url" value="jdbc:HypersonicSQL:BILLDB"/>

<property name="driverClassName" value="org.hsql.jdbcDriver"/>

<property name="poolSize" value="10" />

</bean>

AtomikosNonXADataSourceBean Class의 주요 프로퍼티는 다음과 같다.

- user : DB에 접근할 때 사용할 사용자 계정.

- password : DB에 접근할 때 사용할 암호.

- url : JDBC URL

- driverClassName : JDBC 드라이버 Class 이름.

AtomikosNonXADataSourceBean Class의 경우 완전한 XA를 지원하지 않기 때문에, 원자성(atomic)을 100% 보장하지 못한다는 점에 유의해야 한다.

(3) TransactionEssentials가 제공한 XADataSource를 이용.

: TransactionEssentials가 제공하는 Class를 이용해서 XADataSource를 설정했다면, 남은 작업은 하나의 Transaction 범위에서 두 개 이상의 DataSource를 사용하도록

설정하는 것뿐이다. 아래 Code는 설정 예를 보여주고 있다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<!--설정-->

</bean>

<bean id="storeDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="STOREDB" />

<!--설정-->

</bean>

<bean id="itemDao" class="madvirus.spring.chap09.store.dao.jdbc.JdbcItemDao">

<constructor-arg>

<ref bean="storeDataSource" />

</constructor-arg>

</bean>

<bean id="paymentInfoDao"

class="madvirus.spring.chap09.store.dao.jdbc.JdbcPaymentInfoDao">

<constructor-arg>

<ref bean="billingDataSource" />

</constructor-arg>

</bean>

<bean id="purchaseOrderDao"

class="madvirus.spring.chap09.store.dao.jdbc.JdbcPurchaseOrderDao">

<constructor-arg>

<ref bean="storeDataSource" />

</constructor-arg>

</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="placeOrderService"

class="madvirus.spring.chap09.store.domain.PlaceOrderServiceAnnotImpl"

p:itemDao-ref="itemDao" p:paymentInfoDao-ref="paymentInfoDao"

p:purchaseOrderDao-ref="purchaseOrderDao" />

위 Code에서 itemDao, paymentInfoDao, purchaseOrderDao는 각각 storeDataSource와 billingDataSource를 사용하고 있으며, placeOrderService는

itemDao와 paymentInfoDao, purchaseOrderDao를 이용해서 두 개의 DataSource에 접근하게 된다. 위 Code에서 <tx:annotation-drive> 태그에서

참조하는 transactionManager는 TransactionEssentials를 이용해서 설정한 JtaTransactionManager 빈이다.

placeOrderService는 다음과 같이 Transaction 범위 내에서 두 개 이상의 DataSource에 접근하게 된다.

@Transactional

public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)

throws ItemNotFoundException {

Item item = itemDao.findById(orderRequest.getItemId());

if (item == null)

throw new ItemNotFoundException(orderRequest.getItemId());

PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());

paymentInfoDao.insert(paymentInfo);

PurchaseOrder order = new PurchaseOrder(item.getId(), orderRequest

.getAddress(), paymentInfo.getId());

purchaseOrderDao.insert(order);

return new PurchaseOrderResult(item, paymentInfo, order);

}

위 Code에서 order() 메서드는 TransactionEssentials가 제공하는 글로벌 Transaction을 통해서 관리되기 때문에, order() 메서드에서 사용하는 두 개 이상의

DataSource는 하나의 Transaction으로 처리된다.

5.3 커넥션 풀 관련 프로퍼티

: AtomikosDataSourceBean Class와 AtomikosNonXADataSourceBean Class의 커넥션 풀 관련 프로퍼티는 다음과 같다.

프 로 퍼 티 설 명 기본 값

=====================================================================

maxPoolSize 최대 풀 개수. 1

=====================================================================

minPoolSize 최소 풀 개수. 1

=====================================================================

poolSize 최대/최소 풀 개수를 한번에 설정한다. 없음

maxPoolSize를 설정하지 않은 경우 필요하다.

=====================================================================

borrowConnectionTimeout 풀에서 커넥션을 가져오기까지 대기 시간. 30

시간 단위는 초이다.

=====================================================================

maintenanceInterval 커넥션 풀의 검사 주기를 초 단위로 설정한다. 60

=====================================================================

maxIdleTime 사용되지 않는 커넥션의 최대 유효 시간. 단위는 초. 60

=====================================================================

testQuery 사용된 커넥션을 풀에 되돌리기 전에 커넥션을 검사할 때 사용할 쿼리.

=====================================================================

5.4 각 DBMS 별 AtomikosDataSourceBean 설정

: AtomikosDataSourceBean을 이용해서 XADataSource를 설정할 때에는 다음의 두 프로퍼티를 사용한다.

- xaDataSourceClassName : JDBC 드라이버의 XADataSource 구현 Class.

- xaProperties : xaDataSource 구현체를 설정할 때 사용될 프로퍼티. XADataSource 구현 Class의 프로퍼티 설정 메서드(set 메서드)를 이용한다.

오라클을 사용하는 경우의 예제 설정은 다음과 같다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<property name="xaDataSourceClassName"

value="oracle.jdbc.xa.client.OracleXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">bill</prop>

<prop key="password">billpw</prop>

<prop key="url">jdbc:oracle:thin:@localhost-xe:1521:XE</prop>

</props>

</property>

<property name="poolSize" value="10" />

</bean>

AtomikosDataSourceBean의 오라클 설정과 XA를 위한 오라클 설정에 대한 내용은 각각 아래 Site를 참고하기 바란다.

- http://www.atomikos.com/Documentation/ConfiguringOracle

- http://www.atomikos.com/Documentation/ConfiguringOracleForXA

MySQL의 설정 예는 다음과 같다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<property name="xaDataSourceClassName"

value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">bill</prop>

<prop key="password">billpw</prop>

<prop key="serverName">localhost</prop>

<prop key="portNumber">1433</prop>

<prop key="databaseName">BILLDB</prop>

</props>

</property>

<property name="poolSize" value="10" />

</bean>

MySQL에 대한 설정 시 아래 Site를 참고하기 바란다.

- http://www.atomikos.com/Documentation/ConfiguringSQLServer

MYSQL에 대한 설정 예는 다음과 같다.

<bean id="billingDataSource" class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="BILLDB" />

<property name="xaDataSourceClassName"

value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource" />

<property name="xaProperties">

<props>

<prop key="user">root</prop>

<prop key="password"></prop>

<prop key="url">jdbc:mysql://localhost/z\_billing?useUnicode=true&amp;characterEncoding=euckr

</prop>

</props>

</property>

<property name="poolSize" value="1" />

</bean>

오라클, MySQL, MS SQL 이외의 다른 DBMS에 대한 설정 방법은 아래 Site를 참고하기 바란다.

- http://www.atomikos.com/Documentation/ConfiguringTransactionEssentials