**Spring Transactions Tips**

|  |
| --- |
| **Tip** |
| You can omit the transaction-manager attribute in the <tx:annotation-driven/> tag if the bean name of the PlatformTransactionManager that you want to wire in has the name transactionManager. If the PlatformTransactionManager bean that you want to dependency-inject has any other name, then you have to use the transaction-manager attribute explicitly, as in the preceding example. |

**Method visibility and @Transactional**

When using proxies, you should apply the@Transactional annotation only to methods with *public*visibility. If you do annotate protected, private or package-visible methods with the @Transactional annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods.

You can place the @Transactional annotation before an interface definition, a method on an interface, a class definition, or a *public* method on a class. However, the mere presence of the@Transactional annotation is not enough to activate the transactional behavior. The @Transactionalannotation is simply metadata that can be consumed by some runtime infrastructure that is@Transactional-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the preceding example, the <tx:annotation-driven/> element *switches on* the transactional behavior.

|  |  |  |
| --- | --- | --- |
| [Tip] | **Tip** | |
| Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Transactional annotation, as opposed to annotating interfaces. You certainly can place the @Transactional annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are *not inherited from interfaces* means that if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a transactional proxy, which would be decidedly *bad*. | |
| [Note] | | **Note** | |
| In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual transaction at runtime even if the invoked method is marked with @Transactional. | |

Consider the use of AspectJ mode (see mode attribute in table below) if you expect self-invocations to be wrapped with transactions as well. In this case, there will not be a proxy in the first place; instead, the target class will be weaved (that is, its byte code will be modified) in order to turn @Transactional into runtime behavior on any kind of method.

**Spring transaction with a special case**

Problem Statement : A person wants to buy a product from Flipkart, when he tries to buy the item, his entire transaction except the auditLog entry is rolled back due to insufficient available balance in his account. Audit Log is required because for monitoring and to check who is that person who wanted to buy the item. We want to achieve it using Spring transaction.

The entire code is given below.

**Database table design**

**create** **table** person

(

ID **INT** **PRIMARY** **KEY** AUTO\_INCREMENT,

FIRSTNAME **VARCHAR**(26),

LASTNAME **VARCHAR**(26)

);

**create** **table** ShoppingCart

(

id **int** **primary** **key** AUTO\_INCREMENT,

name **varchar**(50),

noOfItems **int**

);

**create** **table** Account

(

id **int** **primary** **key** AUTO\_INCREMENT,

name **varchar**(50),

actNo **varchar**(50)

);

**create** **table** Product

(

id **int** **primary** **key** AUTO\_INCREMENT,

name **varchar**(50),

status **varchar**(50)

);

/\*Delete the tables\*/

**delete** **from** person;

**delete** **from** ShoppingCart;

**delete** **from** Account;

**delete** **from** Product;

**COMMIT**;

/\*All select queries\*/

**select** \* **from** person;

**select** \* **from** ShoppingCart;

**select** \* **from** Account;

**select** \* **from** Product;

**Maven Configuration (pom.xml)**

**Put the pom.xml details here.**

**Entity Layer**

**Account.java**

**package** com.ddlab.rnd.spring.txn;

**public** **class** Account {

**private** **int** id;

**private** String name;

**private** String actNo;

**public** Account(String name, String actNo) {

**super**();

**this**.name = name;

**this**.actNo = actNo;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** String getActNo() {

**return** actNo;

}

}

**Person.java**

**package** com.ddlab.rnd.spring.txn;

**public** **class** Person {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

}

**Product.java**

**package** com.ddlab.rnd.spring.txn;

**public** **class** Product {

**private** **int** id;

**private** String name;

**private** String status;

**public** Product(String name, String status) {

**super**();

**this**.name = name;

**this**.status = status;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** String getStatus() {

**return** status;

}

}

**ShoppingCart.java**

**package** com.ddlab.rnd.spring.txn;

**public** **class** ShoppingCart {

**private** **int** id;

**private** String name;

**private** **int** noOfItems;

**public** ShoppingCart(String name, **int** noOfItems) {

**super**();

**this**.name = name;

**this**.noOfItems = noOfItems;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **int** getNoOfItems() {

**return** noOfItems;

}

}

**DAO Layer**

**IFlipkartDAO.java**

**package** com.ddlab.rnd.spring.txn;

**public** **interface** IFlipkartDAO {

**void** auditLog( Person p );

**void** addToCart( ShoppingCart cart );

**void** buyProduct( Product p );

**void** debitAccount( Account act );

}

**FlipkartDAOImpl.java**

**package** com.ddlab.rnd.spring.txn;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.stereotype.Repository;

@Repository

**public** **class** FlipkartDAOImpl **implements** IFlipkartDAO {

**private JdbcTemplate jdbcTemplate;**

**public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {**

**this.jdbcTemplate = jdbcTemplate;**

**}**

**public** **void** auditLog(Person person) {

String insertQuery = "insert into person (FIRSTNAME,LASTNAME) values(?,?)";

Object[] params = **new** Object[] {person.getFirstName(), person.getLastName() };

jdbcTemplate.update(insertQuery, params);

System.***out***.println("Person saved successfully");

}

**public** **void** addToCart(ShoppingCart cart) {

String insertQuery = "insert into ShoppingCart (name,noOfItems) values(?,?)";

Object[] params = **new** Object[] {cart.getName(), cart.getNoOfItems() };

jdbcTemplate.update(insertQuery, params);

}

**public** **void** buyProduct(Product p) {

String insertQuery = "insert into Product (name,status) values(?,?)";

Object[] params = **new** Object[] {p.getName(), p.getStatus() };

jdbcTemplate.update(insertQuery, params);

}

**public** **void** debitAccount(Account act) {

String insertQuery = "insert into Account (name,actNo) values(?,?)";

Object[] params = **new** Object[] {act.getName(), act.getActNo() };

jdbcTemplate.update(insertQuery, params);

}

}

**Service Layer**

**IFlipkartService.java**

**package** com.ddlab.rnd.spring.txn;

**public** **interface** IFlipkartService {

**void buyProduct(Person p, ShoppingCart cart , Product prod , Account act);**

**void auditLog( Person p );**

**void addToCart( ShoppingCart cart );**

**void buyProduct( Product p );**

**void debitAccount( Account act );**

}

**FlipkartServiceImpl.java**

**package** com.ddlab.rnd.spring.txn;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

@Service

**public** **class** FlipkartServiceImpl **implements** IFlipkartService {

@Autowired

**private** FlipkartDAOImpl flipkartDAO;

@Autowired

**private** FlipkartXtendedService xtendedService;

**@Transactional**

**public** **void** buyProduct(Person p, ShoppingCart cart, Product prod, Account act) {

**handleAuditLog(p);//It works**

**addToCart(cart);**

**buyProduct(prod);**

**debitAccount(act);**

}

**private** **void** handleAuditLog(Person p) {

xtendedService.auditLog(p);

}

@Transactional

**public** **void** auditLog(Person p) {

flipkartDAO.auditLog(p);

}

@Transactional

**public** **void** addToCart(ShoppingCart cart) {

flipkartDAO.addToCart(cart);

}

@Transactional

**public** **void** buyProduct(Product prod) {

flipkartDAO.buyProduct(prod);

}

@Transactional

**public** **void** debitAccount(Account act) {

flipkartDAO.debitAccount(act);

**throw** **new** NullPointerException("There is not enough money to buy");

}

}

**FlipkartXtendedService.java**

**package** com.ddlab.rnd.spring.txn;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Propagation;

**import** org.springframework.transaction.annotation.Transactional;

@Service

**public** **class** FlipkartXtendedService {

@Autowired

**private** FlipkartDAOImpl flipkartDAO;

**@Transactional(propagation = Propagation.*REQUIRES\_NEW*)**

**public** **void** auditLog(Person p) {

flipkartDAO.auditLog(p);

}

}

*In the above class “FlipkartXtendedService.java”, if the method “auditLog” has the following annotation,*

*@Transactional instead of @Transactional(propagation = Propagation.REQUIRES\_NEW), the whole transaction including audit log will be rolled back.*

***To remember***

*@Transactional – Rolls back the whole transaction including audit log.*

*@Transactional(propagation = Propagation.REQUIRES\_NEW) – Rolls back everything except audit log because the method is designed inside a private method and actual implementation is defined in a separate class called “FlipkartXtendedService.java”.*

**Test Class**

**App.java**

**package** com.ddlab.rnd.spring.txn;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** **void** main(String[] args) {

Person person = **new** Person();

person.setFirstName("Piku");

person.setLastName("Mishra");

ShoppingCart cart = **new** ShoppingCart(person.getFirstName(), 1);

Product prod = **new** Product(person.getFirstName(), "bought");

Account act = **new** Account(person.getFirstName(), "11111111111");

ApplicationContext context = **null**;

**try** {

context = **new** ClassPathXmlApplicationContext("spring-txn.xml");

IFlipkartService service = (IFlipkartService) context.getBean("flipkartService");

service.buyProduct(person, cart, prod, act);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**Spring configuration(spring-txn.xml)**

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

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

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

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:jdbc=*"http://www.springframework.org/schema/jdbc"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"*>

<context:component-scan base-package=*"com.ddlab.rnd.spring.txn"* />

<tx:annotation-driven transaction-manager=*"transactionManager"* /> <!-- Mark It -->

<!-- Initialization for TransactionManager -->

<bean id=*"transactionManager"*

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

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

</bean>

<bean id=*"dataSource"*

class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test"*></property>

<property name=*"username"* value=*"deba"*></property>

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

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

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

</bean>

<bean id=*"flipkartDAO"* class=*"com.ddlab.rnd.spring.txn.FlipkartDAOImpl"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"* />

</bean>

**<bean id=*"flipkartService"* class=*"com.ddlab.rnd.spring.txn.FlipkartServiceImpl"* />**

**<bean id=*"flipkartXtendedService"* class=*"com.ddlab.rnd.spring.txn.FlipkartXtendedService"* />**

</beans>

**After running this above test class, it is observed that the table “person” which is called as audit table is only populated with data and other table remain empty.**

**Miscellaneous Use Cases**

**Case -1 : Use of RuntimeException and explicit handling of exception**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

try {

createAddress(adrs);

} catch (Exception e) {

e.printStackTrace();

}

}

In both the tables, data will be saved. It does not matter whether you have written **Propagation.*REQUIRES\_NEW*** or **Propagation.Required** or simply **@Transactional. The default is Propagation.Required.** The reason behind is we are handling the exception so data will be saved.

**Case -2 : No handling of exception**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional

public void createAddress(Address adrs) {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

}

In this case data will not be saved as createAddress throws a RumtimeException. So entire transaction is rolled back.

**Case -3 : Use of Propagation.*REQUIRES\_NEW***

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

}

In this case data will not be saved , no new transaction will be started as it is inside the public method, same class and same proxy. For more details, refer to the first page for Spring transaction with propagation.

**Case -4: Handling of exception with Propagation.REQUIRES\_NEW**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

createAddress(adrs);

}

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

try {

personAdrsDAO.createAddress(adrs);

throw new RuntimeException("Unwanted exception ...");

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

Data will be saved in both tables, as we are handling exception. Propagation setting has no impact.

**Case -5 : Use a private method and create a separate class to handle**

@Transactional

public void createPersonAddress(Person person, Address adrs) {

createPerson(person);

try {

updateAddress(adrs);

} catch (Exception e) {

e.printStackTrace();

}

}

@Autowired

ApplicationContext context;

private void updateAddress(Address adrs) {

AnotherServiceImpl another = (AnotherServiceImpl) context.getBean("anotherService");

another.createAddress(adrs);

}

@Service

public class AnotherServiceImpl {

@Autowired

private JdbcTemplate jdbcTemplate;

@Transactional(propagation = Propagation.REQUIRES\_NEW)

public void createAddress(Address adrs) {

String insertQuery = "insert into address (city) values(?)";

Object[] params = new Object[] {adrs.getCity() };

jdbcTemplate.update(insertQuery, params);

System.out.println("Address saved successfully");

throw new RuntimeException("Unwanted exception ...");

}

}

In this case, person table will be populated with the data and address will not be populated.

If we change to only @Transactional in the AnotherServiceImpl class, then both the tables will not be populated with data. This is a special case, refer to Flipkart transaction use case above.

**Miscellaneous Transaction Related Concepts**

Two-Phase Commit Mechanism

Unlike a transaction on a local database, a distributed transaction involves altering data on multiple databases. Consequently, distributed transaction processing is more complicated, because the database must coordinate the committing or rolling back of the changes in a transaction as a self-contained unit. In other words, the entire transaction commits, or the entire transaction rolls back.

The database ensures the integrity of data in a distributed transaction using the **two-phase commit mechanism**. In the **prepare phase**, the initiating node in the transaction asks the other participating nodes to promise to commit or roll back the transaction. During the **commit phase**, the initiating node asks all participating nodes to commit the transaction. If this outcome is not possible, then all nodes are asked to roll back.

All participating nodes in a distributed transaction should perform the same action: they should either all commit or all perform a rollback of the transaction. The database automatically controls and monitors the commit or rollback of a distributed transaction and maintains the integrity of the **global database** (the collection of databases participating in the transaction) using the two-phase commit mechanism. This mechanism is completely transparent, requiring no programming on the part of the user or application developer.

The commit mechanism has the following distinct phases, which the database performs automatically whenever a user commits a distributed transaction:

| **Phase** | **Description** |
| --- | --- |
| Prepare phase | The initiating node, called the **global coordinator**, asks participating nodes other than the commit point site to promise to commit or roll back the transaction, even if there is a failure. If any node cannot prepare, the transaction is rolled back. |
| Commit phase | If all participants respond to the coordinator that they are prepared, then the coordinator asks the commit point site to commit. After it commits, the coordinator asks all other nodes to commit the transaction. |
| Forget phase | The global coordinator forgets about the transaction. |