**JDBC and Transaction Management in Spring**

**What is transaction management**

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency. The concept of transactions can be described with following four key properties described as ACID:

**Atomicity**: A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.

**Consistency**: This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.

**Isolation**: There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.

**Durability**: Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

**Normal Spring JDBC Transaction**

Complete code is given below.

package com.ddlab.spring.jdbc.txn;

import java.util.List;

public interface UserDAO {

void insertUser(User user);

User getUser(String username);

List<User> getUsers();

}

package com.ddlab.spring.jdbc.txn;

public class User {

private int id;

private String username;

private String name;

public int getId() {

return id;

}

package com.ddlab.spring.jdbc.txn;

import java.util.List;

public interface UserManager {

void insertUser(User user);

User getUser(String username);

List<User> getUsers();

}

public void setId(int id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

package com.ddlab.spring.jdbc.txn;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.List;

import javax.sql.DataSource;

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

import org.springframework.jdbc.core.RowMapper;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.stereotype.Service;

@Service

public class UserDAOImpl extends JdbcDaoSupport implements UserDAO {

@Autowired

public UserDAOImpl(DataSource dataSource) {

setDataSource(dataSource);

}

@Override

public void insertUser(User user) {

getJdbcTemplate().update(

"INSERT INTO USER1 (USERNAME, NAME) VALUES (?, ?)",

new Object[] {

user.getUsername(),

user.getName()

}

);

}

@Override

public User getUser(String username) {

User user = getJdbcTemplate().

queryForObject("SELECT \* FROM USER1 WHERE USERNAME = ?",

new Object[] { username },

new UserMapper()

);

return user;

}

@Override

public List<User> getUsers() {

List<User> users = getJdbcTemplate().

query("SELECT \* FROM USER1",

new UserMapper()

);

return users;

}

private class UserMapper implements RowMapper<User>{

@Override

public User mapRow(ResultSet rs, int rowNum)

throws SQLException {

User user = new User();

user.setId(rs.getInt("id"));

user.setUsername(rs.getString("username"));

user.setName(rs.getString("name"));

return user;

}

}

}

package com.ddlab.spring.jdbc.txn;

import java.util.List;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

@Service

public class UserManagerImpl implements UserManager {

@Autowired

private UserDAO userDAO;

@Override

@Transactional

public void insertUser(User user) {

userDAO.insertUser(user);

}

@Override

public User getUser(String username) {

return userDAO.getUser(username);

}

@Override

public List<User> getUsers() {

return userDAO.getUsers();

}

}

package com.ddlab.spring.jdbc.txn;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main

{

public static void main( String[] args )

{

ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");

UserManager userManager = (UserManager) ctx.getBean("userManagerImpl");

User user = new User();

user.setUsername("johndoe");

user.setName("johndoe");

// user.setId(11);

userManager.insertUser(user);

System.out.println("User inserted!");

user = userManager.getUser("johndoe");

System.out.println("\nUser fetched!"

+ "\nId: " + user.getId()

+ "\nUsername: " + user.getUsername()

+ "\nName: " + user.getName());

List<User> users = userManager.getUsers();

System.out.println("\nUser list fetched!"

+ "\nUser count: " + users.size());

}

}

<?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:context="http://www.springframework.org/schema/context"

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

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

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

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

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

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

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

<tx:annotation-driven />

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

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />

<property name="username" value="scott" />

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

</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

</beans>

**Local vs. Global Transactions**

Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

**Transactions in Spring : Programmatic vs. Declarative**

Spring supports two types of transaction management:

Programmatic transaction management: This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

Declarative transaction management: This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Declarative transactions can be achieved using Spring annotations and/or Spring XML.

Declarative transaction management is the most common Spring implementation as it has the least impact on application code. The XML declarative approach configures the transaction attributes in a Spring bean configuration file. Declarative transaction management in Spring has the advantage of being less invasive. There is no need for changing application code when using declarative transactions. All you have to do is to modify the application context.

Declarative transaction management approach allows you to manage the transaction with the help of configuration instead of hard coding in your source code. This means that you can separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions. The bean configuration will specify the methods to be transactional. Here are the steps associated with declarative transaction:

We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a pointcut that matches all methods we wish to make transactional and reference the transactional advice.

If a method name has been included in the transactional configuration then created advice will begin the transaction before calling the method.

Target method will be executed in a try / catch block.

If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.

Declarative transaction management

XML based

Annotations based

**Declarative Transactions with Annotaions in Spring**

We have to use the following annotation.

**@Transactional(isolation,noRollbackForClassName,propagation,readOnly,rollbackFor,rollbackForClassName,timeout,value)**

In case of JDBC the DataSourceTransactionManager manages transactions by making calls on the java.sql.Connection object retrieved from the dataSource. A successful transaction is committed by calling the commit() method on the connection. A failed transaction is rolled back by calling the rollback() method.

The important attributes of the annotation include:

**org.springframework.transaction.annotation.Propagation - The transaction propagation type**

**org.springframework.transaction.annotation.Isolation - The transaction isolation level**

**readOnly - indicates if the transaction is a read only**

**rollbackFor - indicates which exceptions should result in a transaction rollback**

**noRollbackFor - indicates which exceptions should not result in a transaction rollback**

**timeout - transaction timeout limit**

To ensure the working of annotation driven transactions we also need to add one more element in the XML file.

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

**Spring transaction propagation**

While dealing with Spring managed transactions the developer is able to specify how the transactions should behave in terms of propagation. In other words the developer has the ability to decide how the business methods should be encapsulated in both logical or physical transactions. Methods from distinct Spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions.

There are 7 types of propagation supported by Spring :

**PROPAGATION\_REQUIRED – Support a current transaction; create a new one if none exists.**

**PROPAGATION\_SUPPORTS – Support a current transaction; execute non-transactionally if none exists.**

**PROPAGATION\_MANDATORY – Support a current transaction; throw an exception if no current transaction exists.**

**PROPAGATION\_REQUIRES\_NEW – Create a new transaction, suspending the current transaction if one exists.**

**PROPAGATION\_NOT\_SUPPORTED – Do not support a current transaction; rather always execute non-transactionally.**

**PROPAGATION\_NEVER – Do not support a current transaction; throw an exception if a current transaction exists.**

**PROPAGATION\_NESTED – Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.**

**REQUIRED behavior**

Spring REQUIRED behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the Spring container will create a new one. If multiple methods configured as REQUIRED behavior are called in a nested way they will be assigned distinct logical transactions but they will all share the same physical transaction. In short this means that if an inner method causes a transaction to rollback, the outer method will fail to commit and will also rollback the transaction. Let's see an example:

**Outer bean**

@Autowired

private TestDAO testDAO;

**Inner bean**

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired() {

throw new RuntimeException("Rollback this transaction!");

}

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequired();

} catch(RuntimeException e){

// handle exception

}

}

Note that the inner method throws a RuntimeException and is annotated with REQUIRED behavior. This means that it will use the same transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

**REQUIRES\_NEW behavior**

REQUIRES\_NEW behavior means that a new physical transaction will always be created by the container. In other words the inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result: they will run in distinct physical transactions.

**Inner bean**

@Override

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void testRequiresNew() {

throw new RuntimeException("Rollback this transaction!");

}

**Outer bean**

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequiresNew(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequiresNew();

} catch(RuntimeException e){

// handle exception

}

}

The inner method is annotated with REQUIRES\_NEW and throws a RuntimeException so it will set its transaction to rollback but will not affect the outer transaction. The outer transaction is paused when the inner transaction starts and then resumes after the inner transaction is concluded. They run independently of each other so the outer transaction may commit successfully.

**NESTED behavior**

The NESTED behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions. This may be familiar to JDBC aware developers as the savepoints are achieved with JDBC savepoints, so this behavior should only be used with Spring JDBC managed transactions (Spring JDBC transactions example).

**MANDATORY behavior**

The MANDATORY behavior states that an existing opened transaction must already exist. If not an exception will be thrown by the container.

**NEVER behavior**

The NEVER behavior states that an existing opened transaction must not already exist. If a transaction exists an exception will be thrown by the container.

**NOT\_SUPPORTED behavior**

The NOT\_SUPPORTED behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.

**SUPPORTS behavior**

The SUPPORTS behavior will execute in the scope of a transaction if an opened transaction already exists. If there isn't an already opened transaction the method will execute anyway but in a non-transactional way.

When the propagation setting is PROPAGATION\_REQUIRED, a logical transaction scope is created for each method upon which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. Of course, in case of standard PROPAGATION\_REQUIRED behavior, all these scopes will be mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transaction's chance to actually commit (as you would expect it to).

If you need a laymans explanation of the use beyond that provided in the Spring Docs

Consider this code...

class Service {

@Transactional(propagation=Propagation.REQUIRED)

public void doSomething() {

// access a database using a DAO

}

}

When doSomething() is called it knows it has to start a Transaction on the database before executing. If the caller of this method has already started a Transaction then this method will use that same physical Transaction on the current database connection.

This @Transactional annotation provides a means of telling your code when it executes that it must have a Transaction. It will not run without one, so you can make this assumption in your code that you wont be left with incomplete data in your database, or have to clean something up if an exception occurs.

In Spring applications, if you enable annotation based transaction support using <tx:annotation-driven/> and annotate any class/method with @Transactional(propagation=Propagation.REQUIRED) then Spring framework will start a transaction and executes the method and commits the transaction. If any RuntimeException occurred then the transaction will be rolled back.

Actually propagation=Propagation.REQUIRED is default propagation level, you don't need to explicitly mentioned it.

Complete code example on Annotation based declarative spring transactions

package com.ddlab.rnd.spring.txn;

public interface IUserDao {

public int insertUser(User user);

public int updateUser(User user) throws Exception;

public void deleteUser(int uid);

public User selectUser(int uid);

}

package com.ddlab.rnd.spring.txn;

public class User {

private Integer id;

private String userName;

private String password;

private String enabled;

public User(Integer id, String userName, String password, String enabled) {

this.id = id;

this.userName = userName;

this.password = password;

this.enabled = enabled;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String isEnabled() {

return enabled;

}

public void setEnabled(String enabled) {

this.enabled = enabled;

}

}

package com.ddlab.rnd.spring.txn;

import java.sql.Types;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.transaction.annotation.Transactional;

@Transactional

// bydefault @Transactional has readOnly false

public class AnnotatedUserDao implements IUserDao {

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void deleteUser(int uid) {

String delQuery = "delete from users where id = ?";

jdbcTemplate.update(delQuery, new Object[] { uid });

}

// @Transactional(rollbackFor=RuntimeException.class) // It will rollback, still it will work even if you do not mention

@Transactional(noRollbackFor=RuntimeException.class) // It will not rollback even if it is throwing exception

public int insertUser(User user) {

String inserQuery = "insert into users (username, password, enabled , id) values (?, ?, ?, ?) ";

Object[] params = new Object[] { user.getUserName(),

user.getPassword(), user.isEnabled(), user.getId() };

int[] types = new int[] { Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,

Types.INTEGER };

int number = jdbcTemplate.update(inserQuery, params, types);

if(true)

throw new RuntimeException("An intentional runtime exception");

return number;

}

// override the class level transactional behaviour for select method

// @Transactional(readOnly = true,rollbackFor=RuntimeException.class)

@Transactional(readOnly = true,noRollbackFor=RuntimeException.class)

public User selectUser(int uid) {

// for all the RuntimeExceptions the transactions will be automatically

// rolled back

throw new RuntimeException("An intentional runtime exception");

}

public int updateUser(User user) throws Exception {

/\*

\* for checked exceptions, transactions are not rolled back by default.

\* The rolled back behaviour can be controlled by mentioning properties

\* in xml file. Please chk

\*/

throw new Exception("An intentional checked exception");

}

}

package com.ddlab.rnd.spring.txn;

import java.util.Random;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestAnnotationTransactions {

public static void main(String[] args) throws Exception {

ApplicationContext applicationContext=new ClassPathXmlApplicationContext("annotationTransactionContext.xml");

IUserDao dao = applicationContext.getBean("userDao", IUserDao.class);

User user= new User(generateId(), "deb"+generateId(), "passowrd", "false");

dao.insertUser(user);

dao.deleteUser(2);

/\*

\* throws a checked Exception that is not automatically rolled

\* back.Contrary to this is the RuntimeException which are automatically

\* rolled back. See next try catch block. We can controll the default

\* roll back behaviour by setting properties in xml SEE

\* txAdviceWithRollBackSettings bean and txAdvice bean definations

\*/

try{

dao.updateUser(user);

}catch (Exception e) {

e.printStackTrace();

}

try{

/\*throws a runtime exception which will be automatically rolled back

\* We can controll the default roll back behaviour by setting properties in xml.

\* SEE txAdviceWithRollBackSettings bean and txAdvice bean definations\*/

dao.selectUser(2);

}catch (Exception e) {

e.printStackTrace();

}

}

private static int generateId(){

return new Random().nextInt(500);

}

}

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

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

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

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

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

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

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

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

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

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

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

<context:annotation-config />

<!-- Add this tag to enable annotations transactions -->

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

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />

<property name="username" value="scott" />

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

</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

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

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

</bean>

<bean id="userDao" class="com.ddlab.rnd.spring.txn.AnnotatedUserDao">

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

</bean>

</beans>

**Declarative Transactions with XML in Spring**

Complete code example is given below.

package com.ddlab.rnd.spring.txn;

public class User {

private Integer id;

private String userName;

private String password;

private String enabled;

public User(Integer id, String userName, String password, String enabled) {

this.id = id;

this.userName = userName;

this.password = password;

this.enabled = enabled;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

package com.ddlab.rnd.spring.txn;

public interface IUserDao {

public int insertUser(User user);

public int updateUser(User user) throws Exception;

public void deleteUser(int uid);

public User selectUser(int uid);

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String isEnabled() {

return enabled;

}

public void setEnabled(String enabled) {

this.enabled = enabled;

}

}

package com.ddlab.rnd.spring.txn;

import java.sql.Types;

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

import org.springframework.context.ApplicationContext;

import org.springframework.jdbc.core.JdbcTemplate;

public class UserDaoImpl implements IUserDao {

/\*For reference visit:

\* http://monstersandwich.blogspot.com/2010/04/spring-transactions-sample-applications.html

\*\*/

private JdbcTemplate jdbcTemplate;

@Autowired

private ApplicationContext applicationContext;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void deleteUser(int uid) {

String delQuery = "delete from users where id = ?";

jdbcTemplate.update(delQuery, new Object[]{uid});

}

public int insertUser(User user) {

String inserQuery = "insert into users (username, password, enabled , id) values (?, ?, ?, ?) ";

Object[] params = new Object[]{user.getUserName(), user.getPassword(),user.isEnabled(),user.getId()};

int[] types = new int[]{Types.VARCHAR,Types.VARCHAR,Types.VARCHAR,Types.INTEGER};

int number = jdbcTemplate.update(inserQuery,params,types);

IUserDao dao = applicationContext.getBean("userDao", IUserDao.class);

//!!!!!!!!!!!!!!! IMPORTANT !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

/\*To test the transaction propagation behaviour call another transacted method.

\* So in this case when you call selectUser, transaction behaviour as described

\* in xml can be practically seen\*/

dao.selectUser(3);

/////////////////// IMPORTANT/////////////////////////////

if(true)

throw new RuntimeException("An intentional runtime exception for insertion");

return number;

}

public User selectUser(int uid) {

// for all the RuntimeExceptions the transactions will be automatically

// rolled back

throw new RuntimeException("An intentional runtime exception for user selection");

}

public int updateUser(User user) throws Exception {

/\*

\* for checked exceptions, transactions are not rolled back by default.

\* The rolled back behaviour can be controlled by mentioning properties

\* in xml file. Please chk

\*/

throw new Exception("An intentional checked exception for update user");

}

}

package com.ddlab.rnd.spring.txn;

import java.util.Random;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class **TestDeclarativeTransactions** {

public static void main(String[] args) throws Exception {

ApplicationContext applicationContext = new ClassPathXmlApplicationContext(

"declarativeTransactionContext.xml");

IUserDao dao = applicationContext.getBean("userDao", IUserDao.class);

User user = new User(generateId(), "DD" + generateId(), "passowrd",

"false");

dao.insertUser(user);

dao.deleteUser(2);

/\*

\* throws a checked Exception that is not automatically rolled

\* back.Contrary to this is the RuntimeException which are automatically

\* rolled back. See next try catch block. We can controll the default

\* roll back behaviour by setting properties in xml SEE

\* txAdviceWithRollBackSettings bean and txAdvice bean definations

\*/

try {

dao.updateUser(user);

} catch (Exception e) {

e.printStackTrace();

}

try {

/\*

\* throws a runtime exception which will be automatically rolled

\* back We can controll the default roll back behaviour by setting

\* properties in xml. SEE txAdviceWithRollBackSettings bean and

\* txAdvice bean definations

\*/

dao.selectUser(2);

} catch (Exception e) {

e.printStackTrace();

}

}

private static int generateId() {

return new Random().nextInt(500);

}

}

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

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

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

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

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

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

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

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

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

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

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

<context:annotation-config />

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

<!--

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

<tx:attributes>

<tx:method name="select\*" read-only="true" />

<tx:method name="\*" />

</tx:attributes>

</tx:advice>

-->

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

<tx:attributes>

<tx:method name="select\*" read-only="true" />

<tx:method name="insert\*" rollback-for="java.lang.Exception"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut id="userDaoTxPointcut" expression="execution(\* com.\*.IUserDao.\*(..))" />

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

</aop:config>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />

<property name="username" value="scott" />

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

</bean>

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

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

</bean>

<bean id="userDao" class="com.ddlab.rnd.spring.txn.UserDaoImpl">

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

</bean>

</beans>

**Programmatic Transactions in Spring using PlatformTransactionManager**

Complete code is given below.

package com.ddlab.rnd.spring.txn;

public interface IUserDao {

public int insertUser(User user);

public int updateUser(User user) throws Exception;

public void deleteUser(int uid);

public User selectUser(int uid);

}

package com.ddlab.rnd.spring.txn;

public class **User** {

private Integer id;

private String userName;

private String password;

private String enabled;

public User(Integer id, String userName, String password, String enabled) {

this.id = id;

this.userName = userName;

this.password = password;

this.enabled = enabled;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String isEnabled() {

return enabled;

}

public void setEnabled(String enabled) {

this.enabled = enabled;

}

}

package com.ddlab.rnd.spring.txn;

import java.sql.Types;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.DefaultTransactionDefinition;

/\*\*

\* This class demonstrates how to use transaction template to programmatically handle transactions.

\*/

public class **PlatformTxManagerUserDaoImpl** implements IUserDao {

private JdbcTemplate jdbcTemplate;

private PlatformTransactionManager platformTransactionManager;

public void setPlatformTransactionManager(

PlatformTransactionManager platformTransactionManager) {

this.platformTransactionManager = platformTransactionManager;

}

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void deleteUser(final int uid) {

DefaultTransactionDefinition paramTransactionDefinition = new DefaultTransactionDefinition();

// you can set propagation behaviour like this

// paramTransactionDefinition.setPropagationBehavior(DefaultTransactionDefinition.PROPAGATION\_REQUIRES\_NEW);

TransactionStatus status = platformTransactionManager

.getTransaction(paramTransactionDefinition);

try {

String delQuery = "delete from users where id = ?";

jdbcTemplate.update(delQuery, new Object[] { uid });

platformTransactionManager.commit(status);

} catch (Exception e) {

platformTransactionManager.rollback(status);

}

}

public int insertUser(final User user) {

TransactionDefinition paramTransactionDefinition = new DefaultTransactionDefinition();

TransactionStatus status = platformTransactionManager

.getTransaction(paramTransactionDefinition);

String inserQuery = "insert into users (username, password, enabled , id) values (?, ?, ?, ?) ";

Object[] params = new Object[] { user.getUserName(),

user.getPassword(), user.isEnabled(), user.getId() };

int[] types = new int[] { Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,

Types.INTEGER };

int rowsAffected = jdbcTemplate.update(inserQuery, params, types);

platformTransactionManager.commit(status);

return rowsAffected;

}

public User selectUser(int uid) {

return null; // TODO Auto-generated method stub

}

public int updateUser(User user) {

return 0; // TODO Auto-generated method stub

}

}

package com.ddlab.rnd.spring.txn;

import java.util.Random;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class **TestPlatformTransactionManager** {

public static void main(String[] args) throws ClassNotFoundException {

ApplicationContext applicationContext = new ClassPathXmlApplicationContext(

"platformTxManager.xml");

IUserDao dao = applicationContext.getBean("userDao", IUserDao.class);

User user = new User(generateId(), "Deb" + generateId(), "passowrd",

"false");

dao.insertUser(user);

dao.deleteUser(2);

/\*

\* throws a checked Exception that is not automatically rolled

\* back.Contrary to this is the RuntimeException which are automatically

\* rolled back. See next try catch block. We can controll the default

\* roll back behaviour by setting properties in xml SEE

\* txAdviceWithRollBackSettings bean and txAdvice bean definations

\*/

try {

dao.updateUser(user);

} catch (Exception e) {

e.printStackTrace();

}

try {

/\*

\* throws a runtime exception which will be automatically rolled

\* back We can controll the default roll back behaviour by setting

\* properties in xml. SEE txAdviceWithRollBackSettings bean and

\* txAdvice bean definations

\*/

dao.selectUser(2);

} catch (Exception e) {

e.printStackTrace();

}

}

private static int generateId() {

return new Random().nextInt(500);

}

}

<?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:context="http://www.springframework.org/schema/context"

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

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

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

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

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

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

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

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />

<property name="username" value="scott" />

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

</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

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

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

</bean>

<bean id="userDao" class="com.ddlab.rnd.spring.txn.PlatformTxManagerUserDaoImpl">

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

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

</bean>

</beans>

**Programmatic Transactions in Spring using Template**

Complete code is given below.

package com.ddlab.rnd.spring.txn;

public class User {

private Integer id;

private String userName;

private String password;

private String enabled;

public User(Integer id, String userName, String password, String enabled) {

this.id = id;

this.userName = userName;

this.password = password;

this.enabled = enabled;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String isEnabled() {

return enabled;

package com.ddlab.rnd.spring.txn;

public interface IUserDao {

public int insertUser(User user);

public int updateUser(User user) throws Exception;

public void deleteUser(int uid);

public User selectUser(int uid);

}

}

public void setEnabled(String enabled) {

this.enabled = enabled;

}

}

package com.ddlab.rnd.spring.txn;

import java.sql.Types;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.TransactionCallback;

import org.springframework.transaction.support.TransactionCallbackWithoutResult;

import org.springframework.transaction.support.TransactionTemplate;

/\*\*This class demonstrates how to use transaction template to programmatically handle transactions.

\*/

public class **TxTemplatedUserDaoImpl** implements IUserDao {

private JdbcTemplate jdbcTemplate;

private TransactionTemplate transactionTemplate;

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {

this.transactionTemplate = transactionTemplate;

//set transaction propagation behaviour like this

this.transactionTemplate.setPropagationBehavior(TransactionTemplate.PROPAGATION\_REQUIRES\_NEW);

}

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

public void deleteUser(final int uid) {

//use TransactionCallbackWithoutResult handler if ur query doesnt result anything

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(

TransactionStatus paramTransactionStatus) {

try{

String delQuery = "delete from users where id = ?";

jdbcTemplate.update(delQuery, new Object[]{uid});

}catch (Exception e) {

//use this to rollback exception in case of exception

paramTransactionStatus.setRollbackOnly();

}

}

});

}

public int insertUser(final User user) {

//use TransactionCallback handler if some result is returned

return transactionTemplate.execute(new TransactionCallback<Integer>() {

public Integer doInTransaction(

TransactionStatus paramTransactionStatus) {

String inserQuery = "insert into users (username, password, enabled , id) values (?, ?, ?, ?) ";

Object[] params = new Object[]{user.getUserName(), user.getPassword(),user.isEnabled(),user.getId()};

int[] types = new int[]{Types.VARCHAR,Types.VARCHAR,Types.BIT,Types.INTEGER};

return jdbcTemplate.update(inserQuery,params,types);

}

});

}

public User selectUser(int uid) {

return null; // TODO Auto-generated method stub

}

public int updateUser(User user) {

return 0; // TODO Auto-generated method stub

}

}

package com.ddlab.rnd.spring.txn;

import java.util.Random;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class **TestProgramaticTemplateTransactions** {

public static void main(String[] args) throws ClassNotFoundException {

// ApplicationContext applicationContext = new ClassPathXmlApplicationContext(

// "classpath:springjdbc/transactions/programmatic/usingtxtemplate/txTemplateContext.xml");

ApplicationContext applicationContext = new ClassPathXmlApplicationContext(

"txTemplateContext.xml");

IUserDao dao = applicationContext.getBean("userDao", IUserDao.class);

User user=new User(generateId(), "apurav"+generateId(), "passowrd", "false");

dao.insertUser(user);

dao.deleteUser(2);

/\*throws a checked Exception that is not automatically rolled back.Contrary to this is the RuntimeException

which are automatically rolled back. See next try catch block. We can controll the default roll back behaviour by setting properties in xml

SEE txAdviceWithRollBackSettings bean and txAdvice bean definations\*/

try{

dao.updateUser(user);

}catch (Exception e) {

e.printStackTrace();

}

try{

/\*throws a runtime exception which will be automatically rolled back

\* We can controll the default roll back behaviour by setting properties in xml.

\* SEE txAdviceWithRollBackSettings bean and txAdvice bean definations\*/

dao.selectUser(2);

}catch (Exception e) {

e.printStackTrace();

}

}

private static int generateId(){

return new Random().nextInt(500);

}

}

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

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

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

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

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

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

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

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

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />

<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />

<property name="username" value="scott" />

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

</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

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

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

</bean>

<!-- Create instance of transaction template for programmatic transaction manipulation -->

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

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

</bean>

<bean id="userDao" class="com.ddlab.rnd.spring.txn.TxTemplatedUserDaoImpl">

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

<property name="transactionTemplate" ref="txTemplate"></property>

</bean>

</beans>