**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.

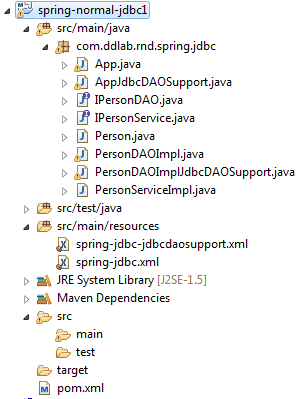
The commit mechanism has the following distinct phases, which the database performs automatically whenever a user commits a distributed transaction:

| **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. |

JDBC and Transaction Management in Spring Framework

**Normal Spring JDBC**

**Structure**



**Maven Configuration (pom.xml)**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>spring-normal-jdbc1</groupId>

<artifactId>spring-normal-jdbc1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>spring-normal-jdbc1</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>4.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<!-- For actual application, we use MYSQL -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.30</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

**Spring JDBC IOC Configuration (spring-jdbc.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.spring.txn"*/>

<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=*"personDAO"* class=*"com.ddlab.rnd.spring.jdbc.PersonDAOImpl"*>

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

</bean>

<bean id=*"personService"* class=*"com.ddlab.rnd.spring.jdbc.PersonServiceImpl"*/>

</beans>

**Entity Layer**

**Person.java**

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

**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;

}

}

**DAO or Repository layer( interface and implementation)**

**IPersonDAO.java**

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

**import** java.util.List;

**public** **interface** IPersonDAO {

**public** **void** addPerson(Person person);

**public** **void** updatePerson(Person person);

**public** **void** deletePerson(Person person);

**public** List<Person> getAllPersons();

**public** Person getPersonById(**int** id);

}

**PersonDAOImpl.java**

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

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.List;

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

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

**public** **class** PersonDAOImpl **implements** IPersonDAO {

**private** JdbcTemplate jdbcTemplate;

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

**this**.jdbcTemplate = jdbcTemplate;

}

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

**try** {

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

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

jdbcTemplate.update(insertQuery, params);

//Since it is not a transaction, it will store the data.

//Then it will throw exception

**throw** **new** NullPointerException("Some unwanted exception ...");

} **catch** (Exception e) {

e.printStackTrace();

}

}

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

**try** {

String queryString = "UPDATE person SET FIRSTNAME = ? WHERE id = ?";

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

jdbcTemplate.update(queryString, params);

} **catch** (Exception e) {

e.printStackTrace();

}

}

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

**try** {

String queryString = "DELETE from person WHERE id = ?";

Object[] params = **new** Object[] {person.getId() };

jdbcTemplate.update(queryString, params);

} **catch** (Exception e) {

e.printStackTrace();

}

}

**public** List<Person> getAllPersons() {

List<Person> personList = **null**;

**try** {

String queryString = "select \* from person";

personList = jdbcTemplate.query(queryString, **new** PersonMapper() );

} **catch** (Exception e) {

e.printStackTrace();

}

**return** personList;

}

**public** Person getPersonById(**int** id) {

String queryString = "select \* from person where id = ?";

Object[] params = **new** Object[]{id};

Person p = jdbcTemplate.queryForObject(queryString, params , **new** PersonMapper() );

**return** p;

}

//RowMapper implementation

**private** **class** PersonMapper **implements** RowMapper {

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Person p = **new** Person();

p.setId(rs.getInt("ID"));

p.setFirstName(rs.getString("FIRSTNAME"));

p.setLastName(rs.getString("LASTNAME"));

**return** p;

}

}

}

**Service Layer ( interface and implementation )**

**IPersonService.java**

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

**import** java.util.List;

**public** **interface** IPersonService {

**public** **void** createPerson( Person person );

**public** **void** updatePerson( Person person );

**public** **void** deletePerson( Person person );

**public** List<Person> getAllPersons();

**public** Person getPersonById( **int** id );

}

**PersonServiceImpl.java**

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

**import** java.util.List;

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

**public** **class** PersonServiceImpl **implements** IPersonService {

@Autowired

**private** IPersonDAO personDAO;

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

personDAO.addPerson(person);

}

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

personDAO.updatePerson(person);

}

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

personDAO.deletePerson(person);

}

**public** List<Person> getAllPersons() {

**return** personDAO.getAllPersons();

}

**public** Person getPersonById(**int** id) {

**return** personDAO.getPersonById(id);

}

}

**Test Class (App.java)**

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

**import** java.util.List;

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

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

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

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

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

//No need to set ID as in MySQL, it is auto increment

person.setFirstName("Piku");

person.setLastName("Mishra");

ApplicationContext context = **null**;

**try** {

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

IPersonService personService = (IPersonService) context.getBean("personService");

//Create a person

personService.createPerson(person);

//Update a person

Person updatedPerson = **new** Person();

updatedPerson.setId(2);

updatedPerson.setFirstName("Bibhu");

personService.updatePerson(updatedPerson);

//Get all persons

List<Person> personList = personService.getAllPersons();

**for**( Person p : personList ) {

System.***out***.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

}

//Get person by Id

Person p = personService.getPersonById(10);

System.***out***.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

System.***out***.println("Saved Successfully...");

//Delete a person

Person deletedPerson = **new** Person();

deletedPerson.setId(1);

personService.deletePerson(deletedPerson);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**Using JdbcDAOSupport**

You can extend you class to JdbcDAOSupport, there is no need to declare JdbcTemplate explicitly. Code is given below.

**PersonDAOImplJdbcDAOSupport.java**

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

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.List;

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

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

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

**public** **class** PersonDAOImplJdbcDAOSupport **extends** JdbcDaoSupport **implements** IPersonDAO {

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

**try** {

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

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

getJdbcTemplate().update(insertQuery, params);

//Since it is not a transaction, it will store the data.

//Then it will throw exception

// throw new NullPointerException("Some unwanted exception ...");

} **catch** (Exception e) {

e.printStackTrace();

}

}

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

**try** {

String queryString = "UPDATE person SET FIRSTNAME = ? WHERE id = ?";

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

getJdbcTemplate().update(queryString, params);

} **catch** (Exception e) {

e.printStackTrace();

}

}

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

**try** {

String queryString = "DELETE from person WHERE id = ?";

Object[] params = **new** Object[] {person.getId() };

getJdbcTemplate().update(queryString, params);

} **catch** (Exception e) {

e.printStackTrace();

}

}

**public** List<Person> getAllPersons() {

List<Person> personList = **null**;

**try** {

**String queryString = "select \* from person";**

**personList = getJdbcTemplate().query(queryString, new PersonMapper() );**

} **catch** (Exception e) {

e.printStackTrace();

}

**return** personList;

}

**public** Person getPersonById(**int** id) {

**String queryString = "select \* from person where id = ?";**

**Object[] params = new Object[]{id};**

**Person p = getJdbcTemplate().queryForObject(queryString, params , new PersonMapper() );**

**return** p;

}

//RowMapper implementation

**private class PersonMapper implements RowMapper** {

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Person p = **new** Person();

p.setId(rs.getInt("ID"));

p.setFirstName(rs.getString("FIRSTNAME"));

p.setLastName(rs.getString("LASTNAME"));

**return** p;

}

}

}

**Spring IOC Configuration (spring-jdbc-jdbcdaosupport.cml)**

<?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*](http://www.springframework.org/schema/tx/spring-tx-4.0.xsd)*"*>

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

<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=*"personDAO"* class=*"com.ddlab.rnd.spring.jdbc.PersonDAOImplJdbcDAOSupport"*>

<property name=*"dataSource"* ref=*"dataSource"*/> <!-- Important to note -->

</bean>

<bean id=*"personService"* class=*"com.ddlab.rnd.spring.jdbc.PersonServiceImpl"*/>

</beans>

**Test class ( AppJdbcDAOSupport.java)**

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

**import** java.util.List;

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

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

**public** **class** AppJdbcDAOSupport {

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

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

//No need to set ID as in MySQL, it is auto increment

person.setFirstName("Arzoo");

person.setLastName("Mishra");

ApplicationContext context = **null**;

**try** {

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

IPersonService personService = (IPersonService) context.getBean("personService");

//Create a person

personService.createPerson(person);

//Get person by Id

Person p = personService.getPersonById(10);

System.***out***.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

System.***out***.println("Saved Successfully...");

// //Delete a person

// Person deletedPerson = new Person();

// deletedPerson.setId(1);

// personService.deletePerson(deletedPerson);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**Spring Transaction Management**

There are two types of transactions management, one is Global and another is Local.

Global transactions

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be sourced from JNDI, meaning that you *also* need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

Previously, the preferred way to use global transactions was via EJB *CMT* (*Container Managed Transaction*): CMT is a form of *declarative transaction management* (as distinguished from *programmatic transaction management*). EJB CMT removes the need for transaction-related JNDI lookups, although of course the use of EJB itself necessitates the use of JNDI. It removes most but not all of the need to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

### Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

<http://stackoverflow.com/questions/22221741/spring-global-transaction-vs-local-transactiopn>

Global Transaction is an application server managed transaction, allowing to work with different transactional resources (this might be two different database, database and message queue, etc)

Local Transaction is resource specific transaction (for example [Oracle Transactions](http://docs.oracle.com/cd/E11882_01/server.112/e10713/transact.htm#CNCPT016)) and application server has nothing to do with them.

<http://www.geekinterview.com/question_details/21851>

The primary difference is that local transaction are single JVM level while global transaction are at multiple JVM level and hence global transactions need different design approach [which can be classified as declarative transaction management & programmatic transaction management]   
  
A global transaction is a mechanism that allows a set of programming tasks, potentially using more than one resource manager and potentially executing on multiple servers, to be treated as one logical unit.   
  
A global transaction may be composed of several local transactions, each accessing the same resource manager. The resource manager is responsible for performing concurrency control and atomicity of updates.

**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.

**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.

<http://javadecodedquestions.blogspot.in/2013/03/java-investment-bank-question.html>

ACID Property:

This property needs to be fulfil by all RDBMS.  Any database operation should fulfill these requirements.

**Atomicity**

     Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. Modification on the data in the database either fail or succeed. The beginning of such a modification starts with a transaction and ends when a transaction finishes

**Consistency**

    The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints,cascades, triggers, and any combination thereof.

**Isolation**

    The isolation property ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially, i.e. one after the other. Each transaction has to execute in total isolation i.e. if T1 and T2 are being executed concurrently then both of them should remain unaware of each other's presence.

**Durability**

    Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter).

**Transaction Isolation levels**

     Transaction isolation levels specify what data is visible to statements within a transaction. These levels directly impact the level of concurrent access by defining what interaction is possible between transactions against the same target data source.

**Database anomalies**

    Database anomalies are generated results that seem incorrect when looked at from the scope of a single transaction, but are correct when looked at from the scope of all transactions. The different types of database anomalies are described as follows:

• **Dirty reads [Read uncommitted]**occur when:

     ◦Transaction A inserts a row into a table.

     ◦Transaction B reads the new row.

     ◦Transaction A rolls back.

     ◦Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

• **Non-Repeatable reads [Read committed]**occur when:

   ◦ Transaction A reads a row.

   ◦ Transaction B changes the row.

   ◦ Transaction A reads the same row a second time and gets the new results.

• **Phantom reads**occur when:

  ◦ Transaction A reads all rows that satisfy a WHERE clause on an SQL query.

  ◦ Transaction B inserts an additional row that satisfies the WHERE clause.

  ◦ Transaction A re-evaluates the WHERE condition and picks up the additional row.

Transaction isolation level expose the application to the allowable database anomolies at the prescribed levels due to its locking strategies.

JDBC transaction isolation levels

    There are five levels of transaction isolation in the IBM Developer Kit for Java JDBC API. Listed from least to most restrictive, they are as follows:

**TRANSACTION\_NONE**

     This is a special constant indicating that the JDBC driver does not support transactions.

**TRANSACTION\_READ\_UNCOMMITTED**

    This level allows transactions to see uncommitted changes to the data. All database anomalies are possible at this level.

**TRANSACTION\_READ\_COMMITTED**

    This level means that any changes made inside a transaction are not visible outside it until the transaction is committed. This prevents dirty reads from being possible.

**TRANSACTION\_REPEATABLE\_READ**

    This level means that rows that are read retain locks so that another transaction cannot change them when the transaction is not completed. This disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

**TRANSACTION\_SERIALIZABLE**

    Tables are locked for the transaction so that WHERE conditions cannot be changed by other transactions that add values to or remove values from a table. This prevents all types of database anomalies.

Transaction Propagation

Propagation means how the service will behave in case of new or no or nested transactions. These are **Transaction Propagation** available in Spring:

[MANDATORY](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#MANDATORY)  
          Support a current transaction, throw  an exception if none exists.

[NESTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NESTED)  
          Execute within a nested transaction if a current transaction exists, behave like PROPAGATION\_REQUIRED else.

[NEVER](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NEVER)  
          Execute non-transactionally, throw an exception if a transaction exists.

[NOT\_SUPPORTED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#NOT_SUPPORTED)  
          Execute non-transactionally, suspend the current transaction if one exists.

[REQUIRED](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRED)  
          Support a current transaction, create a new one if none exists.

[REQUIRES\_NEW](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#REQUIRES_NEW)  
          Create a new transaction, suspend the current transaction if one exists.

[SUPPORTS](http://static.springsource.org/spring/docs/1.2.9/api/org/springframework/transaction/annotation/Propagation.html#SUPPORTS)  
          Support a current transaction, execute non-transactionally if none exists.

**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

**When to use programmatic and declarative transaction management ?**

   Programmatic transaction management is usually a good idea only if you have a small number of transactional operations.   
On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure.

**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.

@Transactional

| **Property** | **Type** | **Description** |
| --- | --- | --- |
| [value](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html#tx-multiple-tx-mgrs-with-attransactional) | String | Optional qualifier specifying the transaction manager to be used. |
| [propagation](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html#tx-propagation) | enum: Propagation | Optional propagation setting. |
| isolation | enum: Isolation | Optional isolation level. |
| readOnly | boolean | Read/write vs. read-only transaction |
| timeout | int (in seconds granularity) | Transaction timeout. |
| rollbackFor | Array of Class objects, which must be derived fromThrowable. | Optional array of exception classes that must cause rollback. |
| rollbackForClassName | Array of class names. Classes must be derived fromThrowable. | Optional array of names of exception classes that must cause rollback. |
| noRollbackFor | Array of Class objects, which must be derived fromThrowable. | Optional array of exception classes that must not cause rollback. |
| noRollbackForClassName | Array of String class names, which must be derived fromThrowable. | Optional array of names of exception classes that must notcause rollback. |

**What is the use of value ?**

#### Multiple Transaction Managers with @Transactional

Most Spring applications only need a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. The value attribute of the @Transactional annotation can be used to optionally specify the identity of the PlatformTransactionManager to be used. This can either be the bean name or the qualifier value of the transaction manager bean. For example, using the qualifier notation, the following Java code

**public** **class** TransactionalService {

*@Transactional("order")*

**public** **void** setSomething(String name) { ... }

*@Transactional("account")*

**public** **void** doSomething() { ... }

}

could be combined with the following transaction manager bean declarations in the application context.

<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="order"/>

</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="account"/>

</bean>

In this case, the two methods on TransactionalService will run under separate transaction managers, differentiated by the "order" and "account" qualifiers. The default <tx:annotation-driven> target bean name transactionManager will still be used if no specifically qualified PlatformTransactionManager bean is found.

**Where to use @Transactional Annotations in Spring ?**

<http://stackoverflow.com/questions/3886909/where-should-transactional-be-place-service-layer-or-dao>

Ideally, Service layer(Manager) represents your business logic and hence it should be annotated with @Transactional.

Service layer may call different DAO to perform DB operations. Lets assume a situations where you have 3 DAO operations in a service method. If your 1st DAO operation failed, other two may be still passed and you will end up inconsistent DB state. Annotating Service layer can save you from such situations.

So in practice you can put them in either place, it's up to you.

By having multiple calls in your service you need @Transactional in service. different calls to service will execute in different transactions if you put @Transactional in service.

<http://stackoverflow.com/questions/1079114/spring-transactional-annotation-best-practice>

I think transactions belong on the Service layer. It's the one that knows about units of work and use cases. It's the right answer if you have several DAOs injected into a Service that need to work together in a single transaction.

In general I agree with the others stating that transactions are usually started on the service level (depending on the granularity that you require of course).

However, in the mean time I also started adding **@Transactional(propagation = Propagation.MANDATORY)** to my DAO layer (and other layers that are not allowed to start transactions but require existing ones) because it is much easier to detect errors where you have forgotten to start a transaction in the caller (e.g. the service). If your DAO is annotated with mandatory propagation you will get an exception stating that there is no active transaction when the method is invoked.

I also have an integration test where I check all beans (bean post processor) for this annotation and fail if there is a @Transactional annotation with propagation other than Mandatory in a bean that does not belong to the services layer. This way I make sure we do not start transactions on the wrong layer.

**Declarative Transactions using Spring Annotations**

**How to use @Transaction annotations**

**Case - 1 : @Transactional at the service layer, exception throws in Service layer**

**@Transactional**

public void createPerson(Person person) {

personDAO.addPerson(person);

throw new RuntimeException("Some unwanted exception ....");

}

In this case, @Transactional, no other attributes are defined, exception throws in the Service layer.

It rolls back the transaction.

**Case - 2 : @Transactionl at the service layer, exception throws in DAO Layer**

**@Transactional**

public void createPerson(Person person) {

personDAO.addPerson(person);

}

public void addPerson(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("Transaction saved successfully");

throw new RuntimeException("Some unwanted exception ....");

}

In this case, @Transactional, no other attributes are defined, exception throws in the DAO layer.

It rollsback the transaction.

**Case - 3 : @Transactionl at the Service layer with attribute noRollbackFor=NullPointerException.class,**

**NullPointerException thrown in Service Layer**

**@Transactional(noRollbackFor=NullPointerException.class)**

public void createPerson(Person person) {

personDAO.addPerson(person);

throw new NullPointerException("Some unwanted exception ....");

}

In this case, @Transactional(noRollbackFor=NullPointerException.class) with the defined attributes, NullPointerException thrown in the Service layer.

It does not rollsback the transaction. It will persist the data.

**Case - 4 : @Transactionl at the Service layer with attribute noRollbackForClassName={"NullPointerException"},**

**NullPointerException throws in DAO Layer**

**@Transactional(noRollbackForClassName={"NullPointerException"})**

public void createPerson(Person person) {

personDAO.addPerson(person);

}

public void addPerson(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("Transaction saved successfully");

throw new NullPointerException("Some unwanted exception ....");

}

In this case, @Transactional(noRollbackFor=NullPointerException.class) with the defined attributes, NullPointerException thrown in the DAO layer. It does not rollsback the transaction. It will persist the data.

**Declarative Transaction with @Transaction Annotation example**

**Maven Configuration (pom.xml)**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>spring-declarative-annotations1</groupId>

<artifactId>spring-declarative-annotations1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>spring-declarative-annotations1</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>4.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<!-- For actual application, we use MYSQL -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.30</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

**Spring Transaction 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=*"personDAO"* class=*"com.ddlab.rnd.spring.txn.PersonDAOImpl"*>

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

</bean>

<bean id=*"personService"* class=*"com.ddlab.rnd.spring.txn.PersonServiceImpl"* />

</beans>

**Entity Layer**

**Person.java**

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

**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;

}

}

**DAO or Repository layer( interface and implementation)**

**IPersonDAO.java**

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

**import** java.util.List;

**public** **interface** IPersonDAO {

**public** **void** addPerson(Person person);

**public** **void** updatePerson(Person person);

**public** **void** deletePerson(Person person);

**public** List<Person> getAllPersons();

**public** Person getPersonById(**int** id);

}

**PersonDAOImpl.java**

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

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.List;

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

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

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

**@Repository**

**public** **class** PersonDAOImpl **implements** IPersonDAO {

**private JdbcTemplate jdbcTemplate;**

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

**this.jdbcTemplate = jdbcTemplate;**

**}**

**public** **void** addPerson(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("Transaction saved successfully");

// throw new NullPointerException("Some unwanted exception ....");

// throw new RuntimeException("Some unwanted exception ....");

}

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

String queryString = "UPDATE person SET FIRSTNAME = ? WHERE id = ?";

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

jdbcTemplate.update(queryString, params);

}

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

String queryString = "DELETE from person WHERE id = ?";

Object[] params = **new** Object[] {person.getId() };

jdbcTemplate.update(queryString, params);

}

**public** List<Person> getAllPersons() {

List<Person> personList = **null**;

**try** {

String queryString = "select \* from person";

personList = jdbcTemplate.query(queryString, **new** PersonMapper() );

} **catch** (Exception e) {

e.printStackTrace();

}

**return** personList;

}

**public** Person getPersonById(**int** id) {

String queryString = "select \* from person where id = ?";

Object[] params = **new** Object[]{id};

Person p = jdbcTemplate.queryForObject(queryString, params , **new** PersonMapper() );

**return** p;

}

//RowMapper implementation

**private** **class** PersonMapper **implements** RowMapper {

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Person p = **new** Person();

p.setId(rs.getInt("ID"));

p.setFirstName(rs.getString("FIRSTNAME"));

p.setLastName(rs.getString("LASTNAME"));

**return** p;

}

}

}

**Service Layer ( interface and implementation )**

**IPersonService.java**

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

**import** java.util.List;

**public** **interface** IPersonService {

**public** **void** createPerson( Person person );

**public** **void** updatePerson( Person person );

**public** **void** deletePerson( Person person );

**public** List<Person> getAllPersons();

**public** Person getPersonById( **int** id );

}

**PersonServiceImpl.java**

**package** com.ddlab.rnd.spring.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** PersonServiceImpl **implements** IPersonService {

@Autowired

**private** IPersonDAO personDAO;

@Transactional(noRollbackForClassName={"NullPointerException"})

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

personDAO.addPerson(person);

}

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

personDAO.updatePerson(person);

}

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

personDAO.deletePerson(person);

}

**public** List<Person> getAllPersons() {

**return** personDAO.getAllPersons();

}

**public** Person getPersonById(**int** id) {

**return** personDAO.getPersonById(id);

}

}

**Test Class (App.java)**

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

**import** java.util.List;

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

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

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

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

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

//No need to set ID as in MySQL, it is auto increment

person.setFirstName("Arzoo");

person.setLastName("Mishra");

ApplicationContext context = **null**;

**try** {

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

IPersonService personService = (IPersonService) context.getBean("personService");

//Create a person

personService.createPerson(person);

/\* //Update a person

Person updatedPerson = new Person();

updatedPerson.setId(2);

updatedPerson.setFirstName("Bibhu");

personService.updatePerson(updatedPerson);

//Get all persons

List<Person> personList = personService.getAllPersons();

for( Person p : personList ) {

System.out.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

}

//Get person by Id

Person p = personService.getPersonById(3);

System.out.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

System.out.println("Saved Successfully...");

//Delete a person

Person deletedPerson = new Person();

deletedPerson.setId(1);

personService.deletePerson(deletedPerson);\*/

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**Declarative Transaction Management with XML, AOP in Spring**

Declarative transaction can be achieved using XML also. The brief code snippet is given below.

<!-- XML based declarative transaction -->

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

**<tx:attributes>**

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

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

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

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

**</tx:attributes>**

**</tx:advice>**

**<aop:config>**

**<aop:pointcut id=*"personServicePointcut"* expression=*"execution(\* com.ddlab.rnd.spring.txn.IPersonService.\*(..))"* />**

**<aop:advisor advice-ref=*"txnAdvice"* pointcut-ref=*"personServicePointcut"*/>**

**</aop:onfig>**

**Complete code example for Declarative transaction management with XML is given below.**

**Maven Configuration( pom.xml)**

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>spring-declarative-xml1</groupId>

<artifactId>spring-declarative-xml1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>spring-declarative-xml1</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>4.0.5.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

**<dependency>**

**<groupId>cglib</groupId>**

**<artifactId>cglib</artifactId>**

**<version>3.1</version>**

**</dependency>**

**<dependency>**

**<groupId>org.aspectj</groupId>**

**<artifactId>aspectjweaver</artifactId>**

**<version>1.8.7</version>**

**</dependency>**

<!-- For actual application, we use MYSQL -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.30</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

**Spring IOC 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"* />

<!-- 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>

**<!-- XML based declarative transaction -->**

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

**<tx:attributes>**

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

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

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

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

**</tx:attributes>**

**</tx:advice>**

**<aop:config>**

**<aop:pointcut id=*"personServicePointcut"* expression=*"execution(\* com.ddlab.rnd.spring.txn.IPersonService.\*(..))"* />**

**<aop:advisor advice-ref=*"txnAdvice"* pointcut-ref=*"personServicePointcut"*/>**

**</aop:config>**

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

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

</bean>

<bean id=*"personDAO"* class=*"com.ddlab.rnd.spring.txn.PersonDAOImpl"*>

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

</bean>

<bean id=*"personService"* class=*"com.ddlab.rnd.spring.txn.PersonServiceImpl"* />

</beans>

**Entity Layer**

**Person.java**

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

**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;

}

}

**DAO or Repository layer( interface and implementation)**

**IPersonDAO.java**

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

**import** java.util.List;

**public** **interface** IPersonDAO {

**public** **void** addPerson(Person person);

**public** **void** updatePerson(Person person);

**public** **void** deletePerson(Person person);

**public** List<Person> getAllPersons();

**public** Person getPersonById(**int** id);

}

**PersonDAOImpl.java**

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

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.List;

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

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

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

**public** **class** PersonDAOImpl **implements** IPersonDAO {

**private JdbcTemplate jdbcTemplate;**

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

**this.jdbcTemplate = jdbcTemplate;**

**}**

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

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

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

jdbcTemplate.update(insertQuery, params);

}

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

String queryString = "UPDATE person SET FIRSTNAME = ? WHERE id = ?";

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

jdbcTemplate.update(queryString, params);

}

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

String queryString = "DELETE from person WHERE id = ?";

Object[] params = **new** Object[] {person.getId() };

jdbcTemplate.update(queryString, params);

}

**public** List<Person> getAllPersons() {

List<Person> personList = **null**;

**try** {

String queryString = "select \* from person";

personList = jdbcTemplate.query(queryString, **new** PersonMapper() );

} **catch** (Exception e) {

e.printStackTrace();

}

**return** personList;

}

**public** Person getPersonById(**int** id) {

String queryString = "select \* from person where id = ?";

Object[] params = **new** Object[]{id};

Person p = jdbcTemplate.queryForObject(queryString, params , **new** PersonMapper() );

**return** p;

}

//RowMapper implementation

**private** **class** PersonMapper **implements** RowMapper {

**public** Person mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

Person p = **new** Person();

p.setId(rs.getInt("ID"));

p.setFirstName(rs.getString("FIRSTNAME"));

p.setLastName(rs.getString("LASTNAME"));

**return** p;

}

}

}

**Service Layer ( interface and implementation )**

**IPersonService.java**

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

**import** java.util.List;

**public** **interface** IPersonService {

**public** **void** createPerson( Person person );

**public** **void** updatePerson( Person person );

**public** **void** deletePerson( Person person );

**public** List<Person> getAllPersons();

**public** Person getPersonById( **int** id );

}

**PersonServiceImpl.java**

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

**import** java.util.List;

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

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

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

**public** **class** PersonServiceImpl **implements** IPersonService {

@Autowired

**private** IPersonDAO personDAO;

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

personDAO.addPerson(person);

**throw** **new** NullPointerException("Unwanted exception ..."); // Transaction will be rolledback

}

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

personDAO.updatePerson(person);

}

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

personDAO.deletePerson(person);

}

**public** List<Person> getAllPersons() {

**return** personDAO.getAllPersons();

}

**public** Person getPersonById(**int** id) {

**return** personDAO.getPersonById(id);

}

}

**Test Class (App.java)**

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

**import** java.util.List;

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

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

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

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

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

//No need to set ID as in MySQL, it is auto increment

person.setFirstName("Arzoo");

person.setLastName("Mishra");

ApplicationContext context = **null**;

**try** {

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

IPersonService personService = (IPersonService) context.getBean("personService");

//Create a person

personService.createPerson(person);

/\* //Update a person

Person updatedPerson = new Person();

updatedPerson.setId(2);

updatedPerson.setFirstName("Bibhu");

personService.updatePerson(updatedPerson);

//Get all persons

List<Person> personList = personService.getAllPersons();

for( Person p : personList ) {

System.out.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

}

//Get person by Id

Person p = personService.getPersonById(3);

System.out.println(p.getId()+"----"+p.getFirstName()+"----"+p.getLastName());

System.out.println("Saved Successfully...");

//Delete a person

Person deletedPerson = new Person();

deletedPerson.setId(1);

personService.deletePerson(deletedPerson);\*/

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**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>

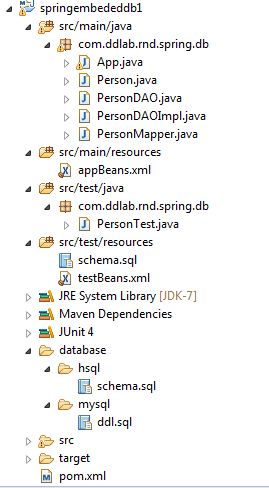
<property name="transactionTemplate" ref="txTemplate"></property>

</bean>

</beans>

Spring JDBC Embedded database for Unit Testing

## Project Structure



## Maven pom.xml

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>springembededdb1</groupId>

<artifactId>springembededdb1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>springembededdb1</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>3.2.8.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

**<dependency>**

**<groupId>commons-dbcp</groupId>**

**<artifactId>commons-dbcp</artifactId>**

**<version>1.2.2</version>**

**</dependency>**

<!-- For Unit Testing, Use HSQLDB -->

**<dependency>**

**<groupId>hsqldb</groupId>**

**<artifactId>hsqldb</artifactId>**

**<version>1.8.0.10</version>**

**</dependency>**

<!-- For actual application, we use MYSQL -->

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<version>5.1.30</version>**

**</dependency>**

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

## SQL File

## Hsql (schema.sql)

**drop** **table** Person if **exists**;

**create** **table** Person (id **int** **primary** **key**, name **varchar**(50) **not** **null**);

### mysql (ddl.sql)

**drop** **table** Person ;

**create** **table** Person (id **int** **primary** **key**, name **varchar**(50) **not** **null**);

**select** \* **from** Person;

## Configuration for unit testing (src/test/resources)

### Schema.sql

**drop** **table** Person if **exists**;

**create** **table** Person (id **int** **primary** **key**, name **varchar**(50) **not** **null**);

### testBeans.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:jdbc=*"http://www.springframework.org/schema/jdbc"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

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

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

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

*http://www.springframework.org/schema/jdbc*

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

**<jdbc:embedded-database id=*"dataSource"*>**

**<jdbc:script location=*"classpath:schema.sql"* />**

**</jdbc:embedded-database>**

<bean id=*"personDao"* class=*"com.ddlab.rnd.spring.db.PersonDAOImpl"*>

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

</bean>

</beans>

## Java code for Junit (src/test/java)

#### PersonTest

package com.ddlab.rnd.spring.db;

import static org.junit.Assert.assertEquals;

import java.util.List;

import org.junit.BeforeClass;

import org.junit.FixMethodOrder;

import org.junit.Test;

import org.junit.runners.MethodSorters;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

/\*\*

\* The Class PersonTest

\* Running test cases in order of method names in ascending order

\* **Use @FixMethodOrder(MethodSorters.NAME\_ASCENDING).**

**\* But it s always discouraged because all test cases should be independent.**

\*/

@FixMethodOrder(MethodSorters.NAME\_ASCENDING)

public class PersonTest {

private static ApplicationContext context = null;

private static PersonDAO personDao = null;

private static int id = 0;

@BeforeClass

public static void init() {

new PersonTest().doSetup();

}

public void doSetup() {

context = new ClassPathXmlApplicationContext("testBeans.xml");

personDao = (PersonDAO) context.getBean("personDao");

}

@Test

public void testCreatePerson() {

createPerson("Deb");

Person p1 = personDao.getPersonById(1);

assertEquals(1,p1.getId());

}

@Test

public void testGetAllPersons() {

List<Person> personList = personDao.getAllPersons();

if(personList.size() == 0 ) {

createPerson("Deb");

personList = personDao.getAllPersons();

}

assertEquals(1, personList.size());

}

@Test

public void testGetPersonById() {

createPerson("Deb3");

Person p1 = personDao.getPersonById(1);

assertEquals(1,p1.getId());

}

private void createPerson(String personName) {

Person p1 = new Person();

p1.setId((++id));

p1.setName(personName);

personDao.createPerson(p1);

}

}

## Main java code (src/main/java)

#### App.java

package com.ddlab.rnd.spring.db;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

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

ApplicationContext context = new ClassPathXmlApplicationContext("appBeans.xml");

PersonDAO personDao = (PersonDAO) context.getBean("personDao");

for( int i = 1 ; i < 6 ; i++ ) {

Person person = new Person();

person.setId(i);

person.setName("Deb"+i);

personDao.createPerson(person);

}

//Get all persons from DB

List<Person> personList = personDao.getAllPersons();

for( Person person : personList ) {

System.out.println("Person Id:::"+person.getId());

System.out.println("Person Name :::"+person.getName());

}

//Get person by Id

Person personById = personDao.getPersonById(3);

System.out.println("Person Id:::"+personById.getId());

System.out.println("Person Name :::"+personById.getName());

}

}

#### Person.java

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

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

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

**private** String name;

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

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

#### PersonDAO.java

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

**import** java.util.List;

**public** **interface** PersonDAO {

**public** List<Person> getAllPersons();

**public** Person getPersonById(**int** id);

**public** **void** createPerson(Person person);

}

#### PersonDAOImpl.java

package com.ddlab.rnd.spring.db;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class PersonDAOImpl implements PersonDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplate;

public List<Person> getAllPersons() {

String sqlQuery = "select \* from Person";

List<Person> persons = jdbcTemplate.query(sqlQuery, new PersonMapper());

return persons;

}

public Person getPersonById(int id) {

String sqlQuery = "select \* from Person where id = ?";

Person person = jdbcTemplate.queryForObject(sqlQuery,

new Object[] { id }, new PersonMapper());

return person;

}

public void createPerson(Person person) {

String sqlQuery = "insert into Person values (?,?)";

jdbcTemplate.update(sqlQuery,

new Object[] { person.getId(), person.getName() });

}

public DataSource getDataSource() {

return dataSource;

}

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplate = new JdbcTemplate(this.dataSource);

}

}

#### PersonMapper.java

package com.ddlab.rnd.spring.db;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class PersonMapper implements RowMapper<Person> {

public Person mapRow(ResultSet rs, int rowNum) throws SQLException {

Person person = new Person();

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

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

return person;

}

}

### Spring configuration (src/main/resources)

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

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

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

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

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

*http://www.springframework.org/schema/jdbc*

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

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

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

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

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

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

</bean>

<bean id=*"personDao"* class=*"com.ddlab.rnd.spring.db.PersonDAOImpl"*>

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

</bean>

</beans>