Spring Declarative Transaction Management Example

[**Previous**](https://www.dineshonjava.com/programmatic-transaction-management/)

[**Next**](https://www.dineshonjava.com/using-hibernate-with-spring-chapter-37/)

**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.

**Popular Tutorials**

* [***Spring Tutorial***](https://www.dineshonjava.com/spring-tutorial/)
* [***Spring MVC Web Tutorial***](https://www.dineshonjava.com/spring-web-mvc-framework-chapter-38/)
* [**Spring Boot Tutorial**](https://www.dineshonjava.com/introduction-to-spring-boot-a-spring-boot-complete-guide/)
* [***Spring Security Tutorial***](https://www.dineshonjava.com/spring-security-take-baby-step-to-secure/)
* [***Spring AOP Tutorial***](https://www.dineshonjava.com/spring-aop-tutorial-with-example-aspect-advice-pointcut-joinpoint/)
* [***Spring JDBC Tutorial***](https://www.dineshonjava.com/using-spring-jdbc-framework-chapter-32/)
* [***Spring HATEOAS***](https://www.dineshonjava.com/spring-hateoas-hypermedia-driven-restful-web-service/)
* [***Microservices with Spring Boot***](https://www.dineshonjava.com/microservices-with-spring-boot/)
* [***REST Webservice***](https://www.dineshonjava.com/jax-rs-web-service-tutorial/)
* [***Core Java***](https://www.dineshonjava.com/core-java-baby-step-to-be-best-java-ian/)
* [***Hibernate Tutorial***](https://www.dineshonjava.com/hibernate-3-on-baby-steps/)
* [***Spring Batch***](https://www.dineshonjava.com/spring-batch-process-with-example/)

**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**](https://www.dineshonjava.com/2012/07/introduction-to-aop-in-spring.html) advice commits the transaction successfully otherwise it performs a rollback.

Declarative transaction management

* XML based
* Annotations based

**Resource managers** such as relation databases provide a transaction manager and an API to control transactions. Those familiar with JDBC will know that by default a transaction is started because of the setting **autocommit= true**. Every statement that changes the database is automatically committed. This behavior can be changed by setting autocommit to false. Now the programmer must explicitly begin a transaction and then commit or rollback the transaction.

In rest of the article, we discuss with an example, declarative transaction management with Spring.

Let us add transactions support to ***EmpDAOImpl.java***. This class has the ***createEmployee()*** method that create an employeeto the database. Let us modify the method a little bit to throw a **RuntimeException** after the insert into the database. The runtime exception is added to pretend that an error occured in business logic while updating the database.

public int createEmployee(String name, int age, Long salary) {

String insert\_sql = "INSERT INTO EMPLOYEE ('name', 'age', 'salary') VALUES(?,?,?)";

JdbcTemplate jt = getJdbcTemplate() ;

Object[] params = new Object[]{name, age, salary};

int ret = jt.update(insert\_sql, params) ;

throw new RuntimeException("simulate Error condition') ;

return ret ;

}

In this method, would you expect the insert to be committed to the database ? The answer is Yes, though that is not the desirable behavior. The default behaviour of JDBC is **autocommit = true** , which means, each insert or update is committed immediately. You could set **autocommit = false** and explicitly commit or rollback at the end of the method. But it is much easier to let your container handle this.

To add declarative transaction management to the above method use the following steps:  
**Step 1:** Define a transaction manager in **spring.xml**

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

**Step 2:** Turn on support for transaction annotations  
Add to **spring.xml**

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

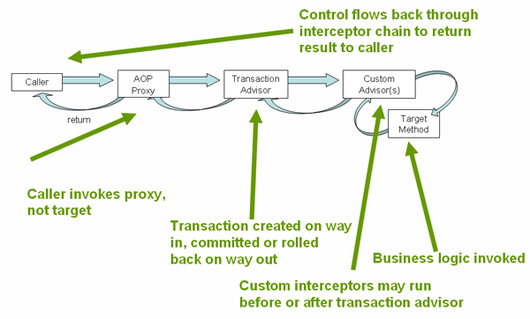
**Step 3:** Add the ***@Transactional*** annotation to the ***createEmployee*** Method

@Transactional

public int createEmployee(Employee employee) {

...

Conceptually, calling a method on a transactional proxy looks like this…



Let us see how above mentioned steps work but before we begin, it is important to have at least one database tables on which we can perform various CRUD operations with the help of transactions. Let us take **Employee** table, which can be created in MySQL **DAVDB** database with the following DDL:

CREATE TABLE Employee(

EMPID INT NOT NULL AUTO\_INCREMENT,

NAME VARCHAR(26) NOT NULL,

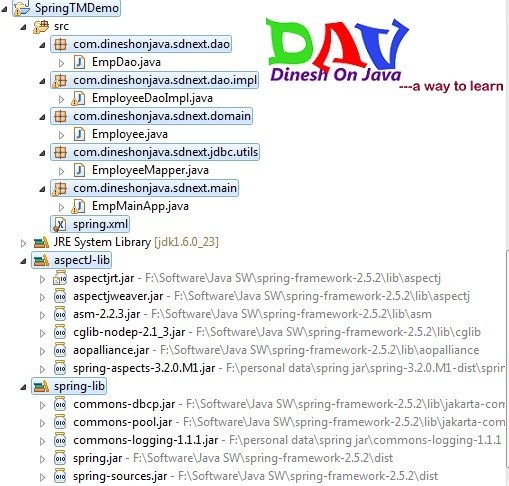
AGE INT NOT NULL,

SALARY BIGINT NOT NULL,

PRIMARY KEY (EMPID)

);

1. Create a project with a name **SpringTMDemo** and create a package ***com.dineshonjava.sdnext*** under the **src** folder in the created project.



**2.** Add required Spring libraries using ***Add User Libs*** option as explained in the *S****pring Hello World Example*** chapter.  
3***.*** Add other required libraries ***mysql-connector-java.jar***  
4.Create DAO interface ***EmpDAO*** and list down all the required methods. Though it is not required and you can directly write ***EmployeeDaoImpl*** class, but as a good practice, let’s do it.  
**EmpDAO.java**

package com.dineshonjava.sdnext.dao;

import java.util.List;

import com.dineshonjava.sdnext.domain.Employee;

/\*\*

\* @author Dinesh Rajput

\*

\*/

public interface EmpDao {

/\*\*

\* This is the method to be used to create

\* a record in the Employee table.

\*/

void create(String name, Integer age, Long salary);

/\*\*

\* This is the method to be used to list down

\* a record from the Employee table corresponding

\* to a passed Employee id.

\*/

Employee getEmployee(Integer empid);

/\*\*

\* This is the method to be used to list down

\* all the records from the Employee table.

\*/

List listEmployees();

/\*\*

\* This is the method to be used to delete

\* a record from the Employee table corresponding

\* to a passed Employee id.

\*/

void delete(Integer empid);

/\*\*

\* This is the method to be used to update

\* a record into the Employee table.

\*/

void update(Integer empid, Integer age);

}

**EmployeeDaoImpl.java**

package com.dineshonjava.sdnext.dao.impl;

import java.util.List;

import org.springframework.dao.DataAccessException;

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

import org.springframework.stereotype.Component;

import com.dineshonjava.sdnext.dao.EmpDao;

import com.dineshonjava.sdnext.domain.Employee;

import com.dineshonjava.sdnext.jdbc.utils.EmployeeMapper;

/\*\*

\* @author Dinesh Rajput

\*

\*/

@Component

public class EmployeeDaoImpl extends JdbcDaoSupport implements EmpDao {

// @Autowired

// private JdbcTemplate jdbcTemplateObject;

//

// /\*\*

// \* @param jdbcTemplateObject the jdbcTemplateObject to set

// \*/

// public void setJdbcTemplateObject(JdbcTemplate jdbcTemplateObject) {

// this.jdbcTemplateObject = jdbcTemplateObject;

// }

@Override

public void create(String name, Integer age, Long salary) {

try {

String SQL = "INSERT INTO Employee (name, age, salary) VALUES (?, ?, ?)";

getJdbcTemplate().update(SQL, new Object[]{name, age, salary} );

System.out.println("Created Record Name = " + name + " Age = " + age+ " Salary = " + salary);

// to simulate the exception.

throw new RuntimeException("simulate Error condition") ;

} catch (DataAccessException e) {

System.out.println("Error in creating record, rolling back");

throw e;

}

}

@Override

public Employee getEmployee(Integer empid) {

String SQL = "SELECT \* FROM Employee WHERE empid = ?";

Employee employee = (Employee) getJdbcTemplate().queryForObject(SQL, new Object[]{empid}, new EmployeeMapper());

return employee;

}

@Override

public List listEmployees() {

String SQL = "SELECT \* FROM Employee";

List employees = (List) getJdbcTemplate().query(SQL, new EmployeeMapper());

return null;

}

@Override

public void delete(Integer empid) {

String SQL = "DELETE FROM Employee WHERE empid = ?";

getJdbcTemplate().update(SQL, new Object[]{empid});

System.out.println("Deleted Record with EMPID = " + empid );

}

@Override

public void update(Integer empid, Integer age) {

String SQL = "UPDATE Employee SET age = ? WHERE empid = ?";

getJdbcTemplate().update(SQL, new Object[]{age, empid});

System.out.println("Updated Record with EMPID = " + empid );

}

}

5. Create other required Java classes **Employee, EmployeeMapper and EmpMainApp** under the ***com.dineshonjava.sdnext*** package. You can create rest of the POJO classes if required.  
**Employee.java**

package com.dineshonjava.sdnext.domain;

/\*\*

\* @author Dinesh Rajput

\*

\*/

public class Employee {

private int empid;

private String name;

private int age;

private long salary;

/\*\*

\* @return the empid

\*/

public int getEmpid() {

return empid;

}

/\*\*

\* @param empid the empid to set

\*/

public void setEmpid(int empid) {

this.empid = empid;

}

/\*\*

\* @return the name

\*/

public String getName() {

return name;

}

/\*\*

\* @param name the name to set

\*/

public void setName(String name) {

this.name = name;

}

/\*\*

\* @return the age

\*/

public int getAge() {

return age;

}

/\*\*

\* @param age the age to set

\*/

public void setAge(int age) {

this.age = age;

}

/\*\*

\* @return the salary

\*/

public long getSalary() {

return salary;

}

/\*\*

\* @param salary the salary to set

\*/

public void setSalary(long salary) {

this.salary = salary;

}

public String toString(){

return "EMPLOYEE{empid- "+this.empid+" name- "+this.name+

" age- "+this.age+" salary- "+this.salary+"}";

}

}

**EmployeeMapper.java**

package com.dineshonjava.sdnext.jdbc.utils;

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.dineshonjava.sdnext.domain.Employee;

/\*\*

\* @author Dinesh Rajput

\*

\*/

public class EmployeeMapper implements RowMapper {

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

Employee employee = new Employee();

employee.setEmpid(rs.getInt("empid"));

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

employee.setAge(rs.getInt("age"));

employee.setSalary(rs.getLong("salary"));

return employee;

}

}

**EmpMainApp.java**

package com.dineshonjava.sdnext.main;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.dineshonjava.sdnext.dao.EmpDao;

import com.dineshonjava.sdnext.domain.Employee;

/\*\*

\* @author Dinesh Rajput

\*

\*/

public class EmpMainApp {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

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

EmpDao empDao = (EmpDao) context.getBean("employeeDaoImpl");

System.out.println("------Records Creation--------" );

empDao.create("Raaz", 25, 50000l);

System.out.println("------Listing Multiple Records--------" );

List employees = empDao.listEmployees();

for (Employee employee : employees) {

System.out.print(employee);

}

}

}

6. Create Beans configuration file **spring.xml** under the src folder.

<beans xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p" xmlns:security="http://www.springframework.org/schema/security" xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://www.springframework.org/schema/beans" xsi:schemalocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd

http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security-2.0.4.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

<context:annotation-config></context:annotation-config>

<context:component-scan base-package="com.dineshonjava.sdnext.dao.impl">

</context:component-scan>

<bean class="org.apache.commons.dbcp.BasicDataSource" id="dataSource">

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

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

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

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

<property name="initialSize" value="2"></property>

<property name="maxActive" value="5"></property>

</bean>

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

<tx:attributes>

<tx:method name="create"></tx:method>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:pointcut expression="execution(\* com.dineshonjava.sdnext.dao.impl.EmployeeDaoImpl.create(..))" id="createOperation"></aop:pointcut>

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

</aop:config>

<!-- Initialization for TransactionManager -->

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

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

</bean>

<bean class="com.dineshonjava.sdnext.dao.impl.EmployeeDaoImpl" id="employeeDaoImpl">

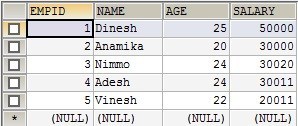
<property name="dataSource" ref="dataSource"></property>

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following exception will be raised. In this case **transaction will be rolled back and no record will be created in the database table**.

**Output:**  
Dec 12, 2012 11:59:23 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh  
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@1ed2ae8: display name [org.springframework.context.support.ClassPathXmlApplicationContext@1ed2ae8]; startup date [Wed Dec 12 23:59:23 IST 2012]; root of context hierarchy  
Dec 12, 2012 11:59:23 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
INFO: Loading XML bean definitions from class path resource [spring.xml]  
Dec 12, 2012 11:59:24 PM org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition  
INFO: Overriding bean definition for bean ’employeeDaoImpl’: replacing [Generic bean: class [com.dineshonjava.sdnext.dao.impl.EmployeeDaoImpl]; scope=singleton; abstract=false; lazyInit=false; autowireCandidate=true; autowireMode=0; dependencyCheck=0; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null] with [Generic bean: class [com.dineshonjava.sdnext.dao.impl.EmployeeDaoImpl]; scope=singleton; abstract=false; lazyInit=false; autowireCandidate=true; autowireMode=0; dependencyCheck=0; factoryBeanName=null; factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in class path resource [spring.xml]]  
Dec 12, 2012 11:59:24 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory  
INFO: Bean factory for application context [org.springframework.context.support.ClassPathXmlApplicationContext@1ed2ae8]: org.springframework.beans.factory.support.DefaultListableBeanFactory@698403  
Dec 12, 2012 11:59:24 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons  
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@698403: defining beans [org.springframework.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotationProcessor,employeeDaoImpl,dataSource,txAdvice,org.springframework.aop.config.internalAutoProxyCreator,createOperation,org.springframework.aop.support.DefaultBeanFactoryPointcutAdvisor#0,transactionManager]; root of factory hierarchy  
**——Records Creation——–**  
**Created Record Name = Raaz Age = 25 Salary = 50000**  
Exception in thread “main” java.lang.RuntimeException: simulate Error condition  
at com.dineshonjava.sdnext.dao.impl.EmployeeDaoImpl.create(EmployeeDaoImpl.java:37)  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)  
at java.lang.reflect.Method.invoke(Method.java:597)  
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:310)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:182)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:149)  
at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:106)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:171)  
at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:90)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:171)  
at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204)  
at $Proxy3.create(Unknown Source)  
at com.dineshonjava.sdnext.main.EmpMainApp.main(EmpMainApp.java:25)



You can try above example after removing exception, and in this case it should commit the transaction and you should see a record in the database.

**Rolling back:**  
The previous section outlined the basics of how to specify the transactional settings for the classes, typically *service, DaoImpl* layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework’s transaction infrastructure that a transaction’s work is to be rolled back is to throw an Exception from code that is currently executing in the context of a transaction. The Spring Framework’s transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and will mark the transaction for rollback.

However, please note that the Spring Framework’s transaction infrastructure code will, by default, only mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of RuntimeException. (Errors will also – by default – result in a rollback.) Checked exceptions that are thrown from a transactional method will not result in the transaction being rolled back.

Exactly which Exception types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific Exception type.

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

<tx:attributes>

<tx:method name="create" read-only="false" rollback-for="NoProductInStockException">

</tx:method>

<tx:method name="\*">

</tx:method>

</tx:attributes>

</tx:advice>

**Configuring different transactional semantics for different beans:**  
Consider the scenario where you have a number of **service & Dao** layer objects, and you want to apply totally different transactional configuration to each of them. This is achieved by defining distinct **<aop:advisor/>** elements with differing ‘**pointcut**‘ and ‘**advice-ref**‘ attribute values.

<beans>

<aop:config>

<aop:pointcut expression="execution(\* x.y.service.\*Service.\*(..))" id="defaultServiceOperation">

</aop:pointcut>

<aop:pointcut expression="execution(\* x.y.service.ddl.DefaultDdlManager.\*(..))" id="noTxServiceOperation">

</aop:pointcut>

<aop:advisor advice-ref="defaultTxAdvice" pointcut-ref="defaultServiceOperation">

</aop:advisor>

<aop:advisor advice-ref="noTxAdvice" pointcut-ref="noTxServiceOperation">

</aop:advisor>

</aop:config>

<!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->

<bean class="x.y.service.DefaultFooService" id="fooService"></bean>

<!-- this bean will also be transactional, but with totally different transactional settings -->

<bean class="x.y.service.ddl.DefaultDdlManager" id="anotherFooService">

</bean>

<tx:advice id="defaultTxAdvice">

<tx:attributes>

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

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

</tx:attributes>

</tx:advice>

<tx:advice id="noTxAdvice">

<tx:attributes>

<tx:method name="\*" propagation="NEVER"></tx:method>

</tx:attributes>

</tx:advice>

<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>