Spring Part - 4

Spring and Transaction

Course Contents

Who This Tutor Is For?

This session will let you know how to manage transactions in Spring. You can understand the programmatic as well as declarative transaction management in Spring Framework. Before going through this session you must understand how Spring supports Database connection. You can visit our Spring part -2 (Spring and Database) and Spring part -3 (Spring and Hibernate) to understand working with Spring and Database.

You can download the source codes of the examples given in this tutor from Download Links available at http://springs-download.blogspot.com/

Good Reading…

Author,
Santosh

Introduction:

What does transaction means?

A **transaction** is a logical unit of work that contains one or more SQL statements. A transaction is an atomic unit. The effects of all the SQL statements in a transaction can be either all committed (applied to the database) or all rolled back (undone from the database). To illustrate the concept of a transaction, consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction can consist of three separate operations:

Decrement the savings account
Increment the checking account
Record the transaction in the transaction journal

Your application must allow for two situations. If all three SQL statements can be performed to maintain the accounts in proper balance, the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, the entire transaction must be rolled back so that the balance of all accounts is correct.

In software, all-or-nothing operations are called transactions. Transactions allow you to group several operations into a single unit of work that either fully happens or fully doesn't happen. If everything goes well then the transaction is a success. But if anything goes wrong, the slate is wiped clean and it's as if nothing ever happened.

In Spring part -2 (Spring and Database), we examined Spring's data access support and saw several ways to read from and write data to the database. When writing to a database, we must ensure that the integrity of the data is maintained by performing the updates within a transaction. Spring has rich support for transaction management, both programmatic and declarative.

Spring's transaction management support

Like EJB Spring supports both **programmatic** and **declarative** transaction management support. But Spring's transaction support is different from EJB. EJB uses JTA (Java Transaction Support) implementation. But Spring uses transaction support offered by the persistent mechanism such as JDBC, Hibernate, JDO, third party JTA implementation for distributed (XA) transaction.

Springs transaction manager

Spring does not directly manage transactions. Instead, it comes with a selection of transaction managers that delegate responsibility for transaction management to a platform-specific transaction implementation provided by either JTA or the persistence mechanism. You need to choose the correct transaction manager as per your design.

Transaction manager (org.springframework.*)	Use at
jdbc.datasource.DataSourceTransactionManager	Spring's JDBC
jms.connection.JmsTransactionManager	JMS 1.1+.
jms.connection.JmsTransactionManager102	JMS 1.0.2.
orm.hibernate.HibernateTransactionManager	Hibernate 2
orm.hibernate3.HibernateTransactionManager	Hibernate 3
orm.jpa.JpaTransactionManager	JPA
transaction.jta.WebLogicJtaTransactionManager	distributed transactions in WebLogic.
transaction.jta.WebSphereTransactionManagerFactoryBean	distributed transactions in WebSphere

Each of the transaction managers acts as a façade to a platform-specific transaction implementation. This makes it possible for you to work with a transaction in Spring with little regard to what the actual transaction implementation is. To use a transaction manager, you'll need to declare it in your application context.

Now let's see the example.

We have two tables ***EMP*** & ***DEPT***. The structure of the two tables would look like:

```
mysql> desc emp;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| empid | int(10)     | NO   | PRI | 0       |       |
| name  | varchar(15) | YES  |     | NULL    |       |
| dept  | int(10)     | YES  |     | NULL    |       |
| sal   | float(12,2) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> desc dept;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| deptid   | int(10)      | NO   | PRI | NULL    |       |
| deptname | varchar(20)  | YES  |     | NULL    |       |
| address  | varchar(100) | YES  |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Scenario:

In a organization, there are Employees and each employee are associated with a Department.

We need to add one **Department** and a **collection of employees** in a single transaction. The condition is,

- if the department does not already exist in the DB **as well as** none of the employee from the collection even exists in DB, then insert both Department and all the employees in DB.

- if department does already exist, then neither department or any of the employees should be inserted into DB and so the transaction should be rolled back.

- if department does not exist but any of the employee already exists in DB, then the whole transaction must be rolled back and none of them should be inserted.

Spring Programmatic Transaction

With programmatic transaction management developers work with the Spring transaction abstraction, which can run over any underlying transaction infrastructure. Programmatic transactions are good when you want complete control over transactional boundaries.

When to use Programmatic Transaction Management?

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. Using programmatic transaction may be a good approach.

2 ways you can use programmatic transaction in Springs

Spring provides **two means** of programmatic transaction management:

- Using the TransactionTemplate
- Using a PlatformTransactionManager implementation directly

I would recommend the first approach (using TransactionTemplate)

Now let's see the example…

As we had seen the transaction scenario as:

- if department does not exist and none of the employee already exist in DB, then add both Department and all the employees.
- if department does already exist, then neither department or any of the employees should be inserted into DB and so the transaction should be rolled back.
- if department does not exist but any of the employee already exists in DB, then the transaction must be rolled back and none of them should be inserted.

The client program would look like :

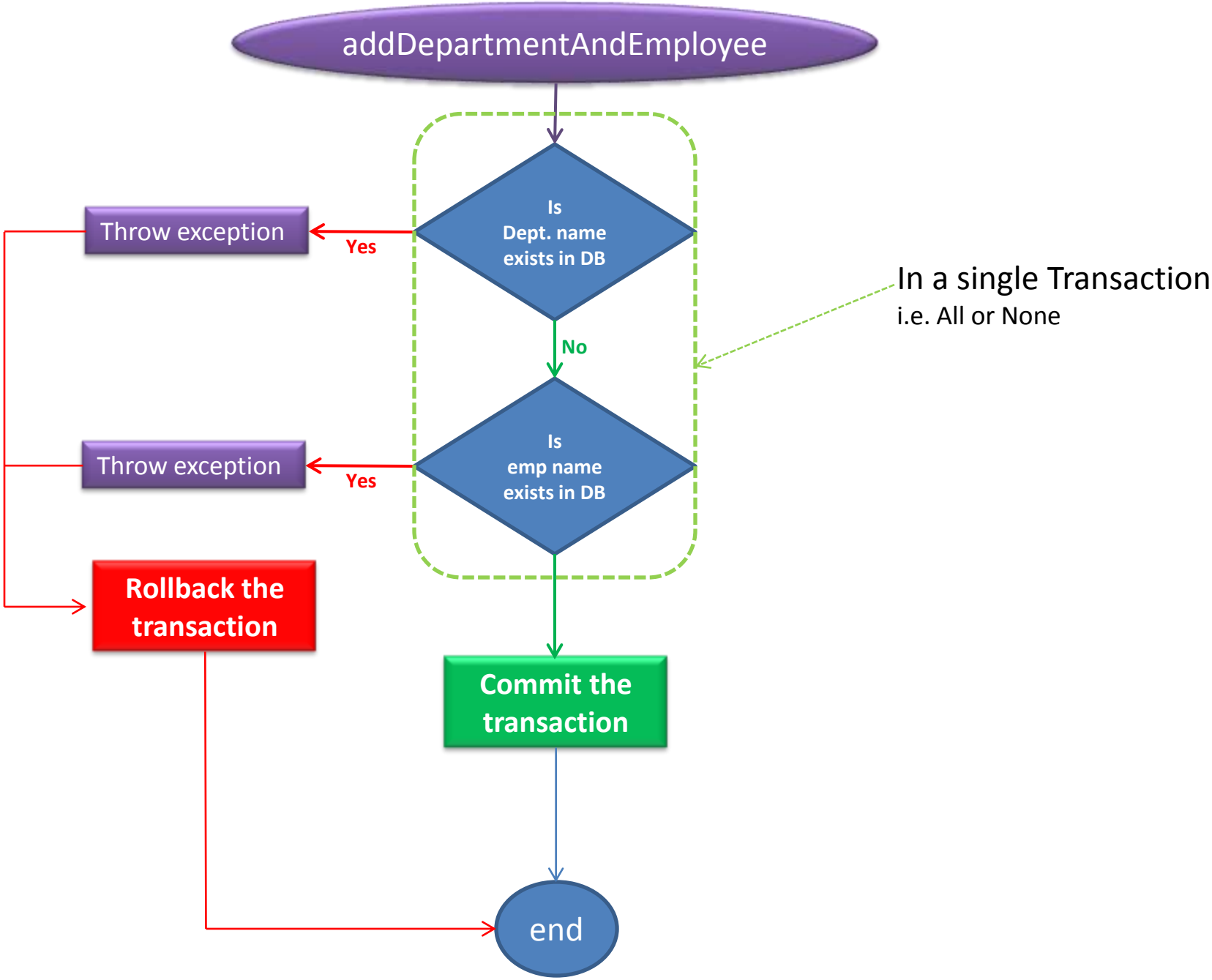
```
OrganizationDao daoService= (OrganizationDao) ctx.getBean("organizationDao");

Department deptPurchase = new Department("Purchase", "MG Road");
Set<Employee> empList = new HashSet<Employee>();
empList.add(new Employee("Santosh", deptPurchase , 232000));
empList.add(new Employee("Srikant", deptPurchase , 238772));
empList.add(new Employee("Kishan", deptPurchase , 98362));
empList.add(new Employee("Billu", deptPurchase , 54755));

daoService.addDepartmentAndEmployee(deptPurchase, empList);
```

Now all opearation happends inside the method:
addDepartmentAndEmployee(Department, List<Employee>).

Let's consider the flow chart to understand when the transaction should be rolled back…


addDepartmentAndEmployee
Is
Dept. name
exists in DB
Yes
Throw exception
No
Is
emp name
exists in DB
Yes
Throw exception
Rollback the
transaction
Commit the
transaction
end
In a single Transaction
i.e. All or None

Using programmatic transaction…

One approach to adding transactions is to programmatically add transactional boundaries directly within the *addDepartmentAndEmployee()* method using Spring's **TransactionTemplate**. Like other template classes in Spring such as JdbcTemplate, TransactionTemplate utilizes a callback mechanism.

Let's follow the below steps:

In Spring config

1. Select appropriate TransactionManager for your application to inject into TransactionTemplate.
2. Select appropriate TransactionTemplate and inject into DAO class.

In DAO class:

3. Define the property for TransactionTemplate.
4. Call TransactionTemplate.execute() method,
   - implementing the TransactionCallback interface,
   - implement the method doTransaction() declared in TransactionCallback interface.

Now let's see these 4 steps one by one in detail.

1: Selecting appropriate TransactionManager for your application.

Already we have seen there are difference transaction managers for different types of application/framework such as *DataSourceTransactionManager* for simple JDBC, *HibernateTransactionManager* for hibernate etc.

I am using *DataSourceTransactionManager* as I am using simple datasource to use JdbcTemplate. The TransactionManager needs a datasource to be declared. (We already have seen how to work with different spring data sources in previous parts).

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="myDataSource" />
</bean>
```

2: Selecting appropriate TransactionTemplate and injecting into DAO

Like there is jdbcTemplate to use the SQL statements, we need to use TransactionTemplate to use the transactions. The transactionTemplate needs a TransactionManager (which we declared in step 1) which is used in the DAO to maintain the transaction.

```
<bean id="organizationDao" class="org.santosh.dao.OrganizationDao">
    <property name="jdbcTemplate" ref="MyJdbcTemplate" />
    <property name="transactionTemplate">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="transactionManager" />
        </bean>
    </property>
</bean>
```

3. Define the property for TransactionTemplate.

```
private TransactionTemplate transactionTemplate;

public TransactionTemplate getTransactionTemplate() {
    return transactionTemplate;
}

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
    this.transactionTemplate = transactionTemplate;
}
```

Concentrate on this class type.

4: Call TransactionTemplate.execute() method

```
public void addDepartmentAndEmployee(final Department dept,
        final Set<Employee> empList) {

    transactionTemplate.execute(new TransactionCallback<Object>() {
        public Object doInTransaction(TransactionStatus ts) {
            try {
                // Start transaction...

                System.out.println("Transaction completed successfully");

            } catch (Exception e) {
                System.out.println("The transaction could not complete because of: "+e.getMessage());
                ts.setRollbackOnly();
            }
            return null;
        }
    });

}
```

Your transaction

Now in our example if you will see, the transaction would look like:

```
transactionTemplate.execute(new TransactionCallback<Object>() {
    public Object doInTransaction(TransactionStatus ts) {
        try {
            // Start transaction...
            addDepartment(dept);

            for(Employee emp:empList)
                addEmployee(emp);

            System.out.println("Transaction completed successfully");

        } catch (Exception e) {
            System.out.println("The transaction could not complete because of:
            ts.setRollbackOnly();
        }
        return null;
    }
});
```

Multiple DB operations in a single transaction

In case any issue with any of the operation, rolls-back happens for all operations executed in the try block. Here where we rollback programmatically ☺

If you see the above example, there are multiple DB operations are happening in a single transaction. If there is any issue with one operation, then all the operations will be rolled back.

The addDepartment() checks if the department name already exists. If so it throws an exception else adds the department.

```
public void addDepartment(Department dept) {
    String SQL_ADD_DEPT = "INSERT INTO dept(deptid, deptname, address) VALUES (?, ?, ?)";

    if (isDepartmentExist(dept.getDeptname()))
        throw new RuntimeException("Department " + dept.getDeptname()+ " already exists, so can not be added again.

    getJdbcTemplate().update(
            SQL_ADD_DEPT,
            new Object[] { getMaxDepartementId()+1, dept.getDeptname(), dept.getAddress()});
    System.out.println("Department " + dept.getDeptname()+ " added in DB");
}
```

Similarly there is the department existence check in the addDepartment() method.

```
public void addDepartment(Department dept) {
    String SQL_ADD_DEPT = "INSERT INTO dept(deptid, deptname, address) VALUES (?, ?, ?)";

    if (isDepartmentExist(dept.getDeptname()))
        throw new RuntimeException("Department " + dept.getDeptname()+ " already exists, so can not be added again.

    getJdbcTemplate().update(
            SQL_ADD_DEPT,
            new Object[] { getMaxDepartementId()+1, dept.getDeptname(), dept.getAddress()});
    System.out.println("Department " + dept.getDeptname()+ " added in DB");
}
```

How the transaction behaves in our example…

If there is no issue in the operation for adding the department, it goes to addEmployee().

The addEmployees() method checks if the Employee name already exists in database or not. If name found for any employee in DB, it throws the exception and as result it rolls back all the operations including the operation of adding the new department.

Here, although there was no issue in adding the department but issue found in adding employee, still the operation of adding the department also rolled back along with the operation of adding employee. The reason is both the operations are executed under a single transaction, So if any exception occurs with any of the process inside a transaction, all the operations inside that transaction needs to be rolled back.

Programmatic Transaction in Hibernate

It is not much difference to use programmatic transaction in Hibernate. You just need to use is the transaction manager class: *org.springframework.orm.hibernate3.HibernateTransactionManager.* You should inject the session factory used in hibernate into the transaction manager in place of data source. So your Spring configuration would look like:

```
<bean id="organizationDao" class="org.santosh.dao.OrganizationDao">
    <property name="hibernateTemplate" ref="myHibernateTemplate"></property>
    <property name="transactionTemplate">
        <bean class="org.springframework.transaction.support.TransactionTemplate">
            <property name="transactionManager" ref="transactionManager" />
        </bean>
    </property>
</bean>

<bean id="myHibernateTemplate"
    class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="mySessionFactory" />
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory" />
</bean>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />

    <property name="mappingResources">
        <list>
            <value>org/santosh/dao/hibernate/Employee.hbm.xml</value>
            <value>org/santosh/dao/hibernate/Department.hbm.xml</value>
        </list>
    </property>
```

With **JDBC** it was using **data source** but in **hibernate**, it is **SessionFactory**

Output

Transaction committed…

No issue with any operation:

```
log4j:WARN No appenders could be found for logger (org.springframework.context.support.ClassPathXmlApplicationContext).
log4j:WARN Please initialize the log4j system properly.
loaded...
Department Purchase added in DB
Employee : Srikant added in DB.
Employee : Santosh added in DB.
Employee : Billu added in DB.
Employee : Kishan added in DB.
Transaction completed successfully
```

Transaction Rolled back…

No issue with Department but employee “Kishan” is still there in DB while executing the client.:

```
log4j:WARN No appenders could be found for logger (org.springframework.context.support.ClassPathXmlApplicationContext).
log4j:WARN Please initialize the log4j system properly.
loaded...
Department Purchase added in DB
Employee : Srikant added in DB.
Employee : Santosh added in DB.
Employee : Billu added in DB.
The transaction could not complete because of: Employee Kishan already exists, so can not be added again...
```

Spring Declarative Transaction

While programmatic transaction management affords you flexibility in precisely defining transaction boundaries in your code, declarative transactions help you decouple an operation from its transaction rules.

Spring's support for declarative transaction management is implemented through Spring's AOP framework

When to use Declarative Transaction Management?

If your applications has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure in Spring. Using Spring, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

Spring provides three ways to declare transactional boundaries in the Spring configuration

1. By proxying transaction using Spring AOP
2. Simple XML-declared transactions
3. Annotation-driven transactions.

By proxying transaction using Spring AOP

In pre-2.0 versions of Spring, declarative transaction management was accomplished by proxying your POJOs with Spring's TransactionProxyFactoryBean. TransactionProxyFactoryBean is a specialization of ProxyFactoryBean that knows how to proxy a POJO's methods by wrapping them with transactional boundaries.

When you are going to configure declarative transaction for proxy transaction, you have to remember and follow the below steps:

Proxy your DAO using the class *TransactionProxyFactoryBean*

e.g. instead

~~*<bean id="organizationDao" class="org.santosh.dao.OrganizationDao">*~~

it should be

<bean id="organizationDao" class=" org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

Define 5 bean properties for TransactionProxyFactoryBean:

- target
- proxyInterface
- transactionManager
- transactionAttributes

We will use the same example what we discussed in *programmatic transaction* in this part. So we have:

- the interface: *OrganizationDao.java*. This has the method: *addDepartmentAndEmployee()* where we need to maintain the transaction.
- the class *OrganizationDaoImpl.java* which implemented the above interface. (This interface is used to make the proxy)

First we will define the bean using the class *TransactionProxyFactoryBean* . This bean will be used in client to call the method that should run within the transaction. Now let's see the configuration.

```
<bean id="organizationDao"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="target" ref="organizationDaoTarget"></property>
    <property name="transactionManager" ref="transactionManager"></property>
    <property name="proxyInterfaces" value="org.santosh.dao.OrganizationDao"></property>
    <property name="transactionAttributes">
        <props>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_SUPPORTS</prop>
        </props>
    </property>
</bean>
```

The client code would look like:

```
OrganizationDao daoService= (OrganizationDao) ctx.getBean("organizationDao");

daoService.<method that must run within a transaction>;
```

1. Property - *target*

*<property name="**target**" ref="organizationDaoTarget">*

Target in a Transaction is nothing but this refers the actual class which contains the implementation of the method where more than one operation run within a transaction. So if any of the operation could not be performed successfully then, all the operations will be rolled back. If there is no issue with any operations defined in the method, the transaction will be committed.

Here *ref="organizationDaoTarget"* is the reference of a bean which refers the implementation class. So you have to define the bean with database template*.
(*If you want to know more about the database templates, please visit our presentation: ***Spring Part – 3 (Spring and Database).***) So make an entry for the target bean as:

```
<bean name="organizationDaoTarget" class="org.santosh.dao.OrganizationDaoImpl">
    <property name="jdbcTemplate" ref="myJdbcTemplate" />
</bean>
```

2. Property - *proxyInterface*

*<property name="**proxyInterface**" value="org.santosh.dao.OrganizationDao">*

This property is related to the property – *Target*.
proxyInterface refers to the interface which is implemented by the target class. This interface is used to proxy the bean. So as you can see the value, the interface *org.santosh.dao.OrganizationDao* contains the declaration of the method which must run in a transaction and this is implemented by the target class.

3. Property - *transactionManager*

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

As we already discussed that Spring comes with a selection of transaction managers that delegate responsibility for transaction management to a platform-specific transaction implementation provided by either JTA or the persistence mechanism. To know more, go back to know the different transaction managers that can be used with Spring. We use *DataSourceTransactionManager*.

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="myDataSource" />
</bean>
```

3. Property - *transactionAttributes*

```
<property name="transactionAttributes">
    <props>
        <prop key="add*">PROPAGATION_REQUIRED</prop>
        <prop key="*">PROPAGATION_SUPPORTS</prop>
    </props>
</property>
```

Transaction attributes are the most important part in Spring Transactions. In Spring declarative transactions are defined with transaction attributes.

A transaction attribute is a description of how transaction policies should be applied to a method.

Regardless of which declarative transaction mechanism you use, you'll have the opportunity to define these attributes.

Let's examine each attribute to understand how it shapes a transaction.

Propagation behavior

Propagation behavior defines the boundaries of the transaction with respect to the client and to the method being called.

Spring defines seven distinct propagation behaviors. These are the constants which you can find in *Interface TransactionDefinition*. These fields are:

static int PROPOGATION_MANDATORY
static int PROPOGATION_REQUIRED
static int PROPOGATION_REQUIRES_NEW
static int PROPOGATION_SUPPORTS
static int PROPOGATION_NOT_SUPPORTED
static int PROPOGATION_NESTED
static int PROPOGATION_NEVER

PROPOGATION_MANDATORY: Support a current transaction; throw an exception if no current transaction exists.

PROPOGATION_REQUIRED: Support a current transaction; create a new one if none exists.

PROPOGATION_REQUIRES_NEW: Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.

PROPOGATION_SUPPORTS: Indicates that the current method does not require a Transactional context, but may run within a transaction if one is already in progress.

PROPOGATION_NOT_SUPPORTED: Indicates that the method should not run within a transaction. If an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.

PROPOGATION_NESTED: Execute within a nested transaction if a current transaction exists, behave like PROPOGATION_REQUIRED else.

PROPOGATION_NEVER: Do not support a current transaction; throw an exception if a current transaction exists

Isolation levels

An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions. In a typical application, multiple transactions run concurrently, often working with the same data to get their job done. One example, in one transaction,

For example, in the *EMP* table, the query is run to updated salary of **Jack** from 4000 to 5000.

```
UPDATE emp SET sal=5000 WHERE name='Jack';
```

But here the it is waiting to commit/rollback the transaction.

Now in another transaction, it run the query to read the salary of Jack as:

```
SELECT sal FROM emp WHERE name='Jack';
```

Now the question is, how much it should print?

the committed value 4000

OR

the uncommitted value 5000

So there we can set the Isolation level. E.g. when we set the Isolation level to TRANSACTION_READ_UNCOMMITTED, the result should be: the uncommitted value – 5000

When we set the Isolation level to TRANSACTION_READ_COMMITTED, the result should be: the committed value – 4000.

If you new to Isolation level, it would look very interesting.

So apart from read committed/read uncommitted, we do have other types of Isolation types. But before we see different isolation levels we can set, we need to know why do we require to set the Isolation levels?

Now let's see the problems in a typical application where we run multiple transactions concurrently. The problems are:

- Dirty Read
- Non-repeatable Read
- Phantom Read

In table the record is:

empid	name	Sal
1	Jack	4000
2	John	4700

There are two transaction *Tx1* and *Tx2*

- ***Dirty Reads***

Tx1 updated one record but not yet committed.

UPDATE emp SET sal=5000 WHERE name='Jack';

Same time, *Tx2* read that updated data (which is not yet committed by Tx1).

SELECT sal FROM emp WHERE name='Jack'

Result : 5000

Now *Tx1* has rollback.

Problem: data obtained by *Tx2* is now invalid i.e. it's 5000 where in DB after rolledback, it remained 4000.

- ***Non-repeatable Reads***

Tx2 read the data from the table in 3 different times.

First time:

SELECT sal FROM emp WHERE name='Jack'

Result: ***4000***

Second time:

SELECT sal FROM emp WHERE name='Jack'

Result: ***5000***

Third time:

SELECT sal FROM emp WHERE name='Jack'

Result: ***6000***

Problem: data obtained by *Tx2* is different each time it executed.
The reason could be: other transactions such as *Tx1* might have updated the data multiple times in different time period and concurrently *Tx2* reads those data.

- ***Phantom Reads***

Tx2 reads all records from a table:

SELECT name FROM emp WHERE sal >=4000 ;

Result :

Jack
John

Tx1 inserted a records into the table:

INSERT INTO emp VALUES (3, 'MIKE', 6300);

Phanton read -> The 2nd execution finds one additional row (Mike) which was not ther

Tx2 again reads all records from a table:

SELECT name FROM emp WHERE sal >=4000 ;

Result :

Jack
John
Mike

Problem: Data obtained by *Tx2* multiple times but it found additional record i.e. employee Mike is found and this is inserted by the concurrent transaction *Tx1*.

To overcome these problems, we need to set the Isolation Level.
Similar to Propogation, Spring defines 5 distinct isolation levels that can be set along with Propogation. These are the constants which you can find in *Interface TransactionDefinition*.

These fields are:

static int ISOLATION_DEFAULT
static int ISOLATION_READ_COMMITTED
static int ISOLATION_READ_UNCOMMITTED
static int ISOLATION_REPEATABLE_READ
static int ISOLATION_SERIALIZABLE

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
ISOLATION_READ_COMMITTED	Prevented	Allowed	Allowed
ISOLATION_READ_UNCOMMITTED	Allowed	Allowed	Allowed
ISOLATION_REPEATABLE_READ	Prevented	Prevented	Allowed
ISOLATION_SERIALIZABLE	Prevented	Prevented	Prevented

ISOLATION_READ_UNCOMMITTED is the most efficient isolation level, but isolates the transaction the least, leaving the transaction open to dirty, non-repeatable, and phantom reads. At the other extreme, ISOLATION_SERIALIZABLE prevents all forms of isolation problems but is the least efficient.

So as the transaction attribute, we have choosed:

propogation as: *PROPOGATION_REQUIRED*

Isolation as: *ISOLATION_SERIALIZABLE*

```
<bean id="organizationDao"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="target" ref="organizationDaoTarget"></property>
    <property name="transactionManager" ref="transactionManager"></property>
    <property name="proxyInterfaces" value="org.santosh.dao.OrganizationDao"></property>
    <property name="transactionAttributes">
        <props>
            <prop key="add*">PROPAGATION_REQUIRED, ISOLATION_SERIALIZABLE</prop>
            <prop key="*">PROPAGATION_SUPPORTS, readOnly</prop>
        </props>
    </property>
</bean>
```

End of proxying transaction using Spring AOP

Do you have Questions ?

Please write to:
santosh.bsil@yahoo.co.in