Spring - 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 RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the 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 transaction 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.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows −

- Begin the transaction using begin transaction command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform commit otherwise rollback all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs. Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

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.

Programmatic vs. Declarative

Spring supports two types of transaction management −

- **Programmatic transaction management** − This means that you have to 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.**

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by the **org.springframework.transaction.PlatformTransactionManager** interface, which is as follows −

```
public interface PlatformTransactionManager {
  TransactionStatus getTransaction(TransactionDefinition definition);
  throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;
  void rollback(TransactionStatus status) throws TransactionException;
}
```

Sr.No	Method & Description
1	**TransactionStatus getTransaction(TransactionDefinition definition)** This method returns a currently active transaction or creates a new one, according to the specified propagation behavior.
2	**void commit(TransactionStatus status)** This method commits the given transaction, with regard to its status.
3	**void rollback(TransactionStatus status)** This method performs a rollback of the given transaction.

The TransactionDefinition is the core interface of the transaction support in Spring and it is defined as follows −

```
public interface TransactionDefinition {
  int getPropagationBehavior();
  int getIsolationLevel();
  String getName();
  int getTimeout();
  boolean isReadOnly();
}
```

Sr.No	Method & Description
1	**int getPropagationBehavior()** This method returns the propagation behavior. Spring offers all of the transaction propaga-

	tion options familiar from EJB CMT.
2	**int getIsolationLevel()** This method returns the degree to which this transaction is isolated from the work of other transactions.
3	**String getName()** This method returns the name of this transaction.
4	**int getTimeout()** This method returns the time in seconds in which the transaction must complete.
5	**boolean isReadOnly()** This method returns whether the transaction is read-only.

Following are the possible values for isolation level −

Sr.No	Isolation & Description
1	**TransactionDefinition.ISOLATION_DEFAULT** This is the default isolation level.
2	**TransactionDefinition.ISOLATION_READ_COMMITTED** Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
3	**TransactionDefinition.ISOLATION_READ_UNCOMMITTED** Indicates that dirty reads, non-repeatable reads, and phantom reads can occur.
4	**TransactionDefinition.ISOLATION_REPEATABLE_READ** Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
5	**TransactionDefinition.ISOLATION_SERIALIZABLE** Indicates that dirty reads, non-repeatable reads, and phantom reads are prevented.

Following are the possible values for propagation types −

Sr.No.	Propagation & Description
1	**TransactionDefinition.PROPAGATION_MANDATORY** Supports a current transaction; throws an exception if no current transaction exists.
2	**TransactionDefinition.PROPAGATION_NESTED** Executes within a nested transaction if a current transaction exists.
3	**TransactionDefinition.PROPAGATION_NEVER** Does not support a current transaction; throws an exception if a current transaction exists.
4	**TransactionDefinition.PROPAGATION_NOT_SUPPORTED** Does not support a current transaction; rather always execute nontransactionally.
5	**TransactionDefinition.PROPAGATION_REQUIRED** Supports a current transaction; creates a new one if none exists.
6	**TransactionDefinition.PROPAGATION_REQUIRES_NEW** Creates a new transaction, suspending the current transaction if one exists.
7	**TransactionDefinition.PROPAGATION_SUPPORTS** Supports a current transaction; executes non-transactionally if none exists.

8	**TransactionDefinition.TIMEOUT_DEFAULT** Uses the default timeout of the underlying transaction system, or none if timeouts are not supported.

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
  boolean isNewTransaction();
  boolean hasSavepoint();
  void setRollbackOnly();
  boolean isRollbackOnly();
  boolean isCompleted();
}
```

Sr.No.	Method & Description
1	**boolean hasSavepoint()** This method returns whether this transaction internally carries a savepoint, i.e., has been created as nested transaction based on a savepoint.
2	**boolean isCompleted()** This method returns whether this transaction is completed, i.e., whether it has already been committed or rolled back.
3	**boolean isNewTransaction()** This method returns true in case the present transaction is new.
4	**boolean isRollbackOnly()** This method returns whether the transaction has been marked as rollback-only.
5	**void setRollbackOnly()** This method sets the transaction as rollback-only.

Programmatic Transaction Management

Programmatic transaction management approach allows you to manage the transaction with the help of programming in your source code. That gives you extreme flexibility, but it is difficult to maintain.

Before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us consider a Student table, which can be created in MySQL TEST database with the following DDL −

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Second table is Marks in which we will maintain marks for students based on years. Here SID is the foreign key for Student table.

```
CREATE TABLE Marks(
   SID INT NOT NULL,
   MARKS  INT NOT NULL,
   YEAR   INT NOT NULL
);
```

Let us use **PlatformTransactionManager** directly to implement the programmatic approach to implement transactions. To start a new transaction, you need to have a instance of **TransactionDefinition** with the appropriate transaction attributes. For this example, we will simply create an instance **of DefaultTransactionDefinition** to use the default transaction attributes.

Once the **TransactionDefinition** is created, you can start your transaction by calling getTransaction() method, which returns an instance of TransactionStatus. The TransactionStatus objects helps in tracking the current status of the transaction and finally, if everything goes fine, you can use commit() method of PlatformTransactionManager to commit the transaction, otherwise you can use rollback() to rollback the complete operation.

Now, let us write our Spring JDBC application which will implement simple operations on Student and Marks tables. Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

Steps	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.

3 Add Spring JDBC specific latest libraries **mysql-connector-java.jar**, **org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already.

4 Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it.

5 Create other required Java classes *StudentMarks*, *StudentMarksMapper*, *StudentJDBCTemplate* and *MainApp* under the *com.tutorialspoint* package. You can create rest of the POJO classes if required.

6 Make sure you already created **Student** and **Marks** tables in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password.

7 Create Beans configuration file *Beans.xml* under the **src** folder.

8 The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Following is the content of the Data Access Object interface file **StudentDAO.java**

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
  /**
    * This is the method to be used to initialize
    * database resources ie. connection.
  */
  public void setDataSource(DataSource ds);

  /**
    * This is the method to be used to create
    * a record in the Student and Marks tables.
  */
  public void create(String name, Integer age, Integer marks, Integer year);

  /**
    * This is the method to be used to list down
    * all the records from the Student and Marks tables.
  */
  public List<StudentMarks> listStudents();
}
```

Following is the content of the StudentMarks.java file

```
package com.java.spring.tx.bean;

public class StudentMarks {
```

```
private Integer age;
private String name;
private Integer id;
private Integer marks;
private Integer year;
private Integer sid;

public Integer getAge() {
return age;
}

public void setAge(Integer age) {
this.age = age;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public Integer getId() {
return id;
}

public void setId(Integer id) {
this.id = id;
}

public Integer getMarks() {
return marks;
}

public void setMarks(Integer marks) {
this.marks = marks;
}

public Integer getYear() {
return year;
}

public void setYear(Integer year) {
this.year = year;
}

public Integer getSid() {
return sid;
}

public void setSid(Integer sid) {
this.sid = sid;
}
}
```

Following is the content of the StudentMarksMapper.java file

```
package com.java.spring.tx.repository;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.java.spring.tx.bean.StudentMarks;

public class StudentMarksMapper implements RowMapper<StudentMarks> {
      public StudentMarks mapRow(ResultSet rs, int rowNum) throws SQLException {
            StudentMarks studentMarks = new StudentMarks();
            studentMarks.setId(rs.getInt("id"));
            studentMarks.setName(rs.getString("name"));
            studentMarks.setAge(rs.getInt("age"));
            studentMarks.setSid(rs.getInt("sid"));
            studentMarks.setMarks(rs.getInt("marks"));
            studentMarks.setYear(rs.getInt("year"));

            return studentMarks;
      }
}
```

Following is the implementation class file StudentJDBCTemplate.java for the defined DAO interface StudentDAO

```
package com.java.spring.tx.repository;

import java.util.List;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
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;

import com.java.spring.tx.bean.StudentMarks;

public class StudentJDBCTemplate implements StudentDAO {

      private DataSource dataSource;
      private JdbcTemplate jdbcTemplate;
      private PlatformTransactionManager pTransactionManager;

      public void setDataSource(DataSource dataSource) {
            this.dataSource=dataSource;
            this.jdbcTemplate = new JdbcTemplate(dataSource);
      }

      public void setpTransactionManager(PlatformTransactionManager pTransactionMa-
nager) {
            this.pTransactionManager = pTransactionManager;
      }

      public void create(int id, String name, Integer age, Integer marks, Integer
year) {
```

```
            TransactionDefinition transDef = new DefaultTransactionDefinition();
            TransactionStatus transStatu = pTransactionManager.getTransac-
tion(transDef);

            try {
                  String sql ="insert into StudentTrans(id, name, age) values(?, ?,
?)";
                  jdbcTemplate.update(sql, id, name, age);

                  // Get the latest student id to be used in Marks table
                  String queryForInt = "Select max(id) from StudentTrans";
                  List<Map<String, Object>> queryForList = jdbcTemplate.queryFor-
List(queryForInt);

                  String SQL3 = "insert into Marks(sid, marks, year) " + "values
(?, ?, ?)";
                  jdbcTemplate.update(SQL3, 20, marks, year);

                  System.out.println("Created Name = " + name + ", Age = " + age);
                  pTransactionManager.commit(transStatu);

            } catch (DataAccessException e) {
                  System.out.println("Error in creating record, rolling back");
                  pTransactionManager.rollback(transStatu);
                  throw e;
            }
      }

      public List<StudentMarks> listStudents() {
            String sql = "select * from StudentTrans, Marks where Student-
Trans.id=Marks.sid";
            List<StudentMarks> listStudMarks = jdbcTemplate.query(sql, new Student-
MarksMapper());
            return listStudMarks;
      }

}
```

Now let us move with the main application file MainApp.java, which is as follows –

```
package com.java.spring.tx;

import java.util.List;

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

import com.java.spring.tx.bean.StudentMarks;
import com.java.spring.tx.repository.StudentJDBCTemplate;

public class MainApp {
      public static void main(String[] args) {
            ApplicationContext context = new ClassPathXmlApplicationContext("re-
sources/Beans.xml");
            StudentJDBCTemplate studJdbcTemplate = (StudentJDBCTemplate)con-
text.getBean("studentJDBCTemplate");

            System.out.println("------Records creation--------");
            studJdbcTemplate.create(10, "Zara", 11, 99, 2010);
            studJdbcTemplate.create(20, "Nuha", 20, 97, 2010);
            studJdbcTemplate.create(30, "Ayan", 25, 100, 2011);

            System.out.println("------Listing all the records--------");
            List<StudentMarks> studentMarks = studJdbcTemplate.listStudents();

            for (StudentMarks record : studentMarks) {
                  System.out.print("ID : " + record.getId());
                  System.out.print(", Name : " + record.getName());
                  System.out.print(", Marks : " + record.getMarks());
                  System.out.print(", Year : " + record.getYear());
                  System.out.println(", Age : " + record.getAge());
            }
      }
}
```

Following is the configuration file Beans.xml

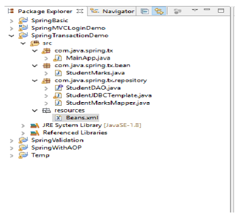
```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

<!-- Initialization for data source -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerData-
Source">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="Admin@123"/>
</bean>

<!-- Initialization for TransactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSour-
ceTransactionManager">
<property name = "dataSource"  ref = "dataSource" />
</bean>

<!-- Definition for studentJDBCTemplate bean -->
<bean id="studentJDBCTemplate" class="com.java.spring.tx.repository.StudentJDBCTem-
plate">
<property name="dataSource" ref="dataSource"/>
<property name="pTransactionManager" ref="transactionManager"/>
</bean>

</beans>
```



```
> SpringBasic
> SpringMVCLoginDemo
v SpringTransactionDemo
  v src
    v com.java.spring.tx
      > MainApp.java
    v com.java.spring.tx.bean
      > StudentMarks.java
    v com.java.spring.tx.repository
      > StudentDAO.java
      > StudentJDBCTemplate.java
      > StudentMarksMapper.java
    v resources
        Beans.xml
  > JRE System Library [JavaSE-1.8]
  > Referenced Libraries
> SpringValidation
> SpringWithAOP
> Temp
```

Spring Declarative Transaction Management

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 at the same time we define a **pointcut** that matches all methods we wish to make transaction and reference the transactional advice.
- If a method name has been included in the transactional configuration, then the 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.

Let us see how the above-mentioned steps work but before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us take a Student table, which can be created in MySQL TEST database with the following DDL −

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Second table is Marks in which we will maintain marks for the students based on years. Here SID is the foreign key for the Student table.

```
CREATE TABLE Marks(
   SID INT NOT NULL,
   MARKS  INT NOT NULL,
   YEAR   INT NOT NULL
);
```

Now, let us write our Spring JDBC application which will implement simple operations on the Student and Marks tables. Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

Step	Description
1	Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project.
2	Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter.
3	Add other required libraries *mysql-connector-java.jar*, *aopalliance-x.y.jar*, *org.springframework.jdbc.jar*, and *org.springframework.transaction.jar* in the project. You can download required libraries if you do not have them already.
4	Create DAO interface *StudentDAO* and list down all the required methods. Though it is not required and you can directly write *StudentJDBCTemplate* class, but as a good practice, let's do it.
5	Create other required Java classes *StudentMarks*, *StudentMarksMapper*, *StudentJDBCTemplate* and *MainApp* under the *com.tutorialspoint* package. You can create rest of the POJO classes if required.
6	Make sure you already created **Student** and **Marks** tables in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the given username and password.
7	Create Beans configuration file *Beans.xml* under the **src** folder.
8	The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

Following is the content of the Data Access Object interface file StudentDAO.java

```
package com.java.spring.dec.tx.repo;

import java.util.List;

import javax.sql.DataSource;

import com.java.spring.dec.tx.bean.StudentMarks;

public interface StudentDAO {
/** This is the method to be used to initialize database resources
ie. connection.*/
public void setDataSource(DataSource ds);

/** This is the method to be used to create a record in the Student
and Marks tables. */
public void create(int id, String name, Integer age, Integer marks,
Integer year);

/** This is the method to be used to list down all the records from
the Student and Marks tables.*/
public List<StudentMarks> listStudents();
}
```

Following is the content of the StudentMarks.java file

```
package com.java.spring.dec.tx.bean;

public class StudentMarks {
private Integer age;
private String name;
private Integer id;
private Integer marks;
private Integer year;
private Integer sid;

public void setAge(Integer age) {
this.age = age;
}

public Integer getAge() {
return age;
}

public void setName(String name) {
this.name = name;
}

public String getName() {
return name;
}

public void setId(Integer id) {
this.id = id;
}

public Integer getId() {
return id;
}

public void setMarks(Integer marks) {
this.marks = marks;
}

public Integer getMarks() {
return marks;
}

public void setYear(Integer year) {
this.year = year;
}

public Integer getYear() {
```

```
return year;
}

public void setSid(Integer sid) {
this.sid = sid;
}

public Integer getSid() {
return sid;
}
}
```

Following is the content of the StudentMarksMapper.java file

```
package com.java.spring.dec.tx.repo;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

import com.java.spring.dec.tx.bean.StudentMarks;

public class StudentMarksMapper implements RowMapper<StudentMarks> {
public StudentMarks mapRow(ResultSet rs, int rowNum) throws SQLException {
StudentMarks studentMarks = new StudentMarks();
studentMarks.setId(rs.getInt("id"));
studentMarks.setName(rs.getString("name"));
studentMarks.setAge(rs.getInt("age"));
studentMarks.setSid(rs.getInt("sid"));
studentMarks.setMarks(rs.getInt("marks"));
studentMarks.setYear(rs.getInt("year"));

return studentMarks;
}
}
```

Following is the implementation class file StudentJDBCTemplate.java for the defined DAO interface StudentDAO

```
package com.java.spring.dec.tx.repo;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;

import com.java.spring.dec.tx.bean.StudentMarks;

public class StudentJDBCTemplate implements StudentDAO {

private JdbcTemplate jdbcTemplate;
private DataSource dataSource;

public void setDataSource(DataSource dataSource) {
this.dataSource=dataSource;
this.jdbcTemplate = new JdbcTemplate(dataSource);
}
public List<StudentMarks> listStudents() {
String SQL = "select * from StudentTrans, Marks where StudentTrans.id = Marks.sid";
List<StudentMarks> studentMarks = jdbcTemplate.query(SQL, new StudentMarksMap-
per());

return studentMarks;
}

public void create(int id, String name, Integer age, Integer marks, Integer year) {
try {
String SQL1 = "insert into StudentTrans (id, name, age) values (?, ?, ?)";
jdbcTemplate.update(SQL1, id, name, age);

// Get the latest student id to be used in Marks table
//String SQL2 = "select max(id) from Student";
//int sid = jdbcTemplate.queryForInt(SQL2);

String SQL3 = "insert into Marks(sid, marks, year) " + "values (?, ?, ?)";
jdbcTemplate.update(SQL3, id, marks, year);
System.out.println("Created Name = " + name + ", Age = " + age);

// to simulate the exception.
//throw new RuntimeException("simulate Error condition");
} catch (DataAccessException e) {
System.out.println("Error in creating record, rolling back");
throw e;
}
}
}
```

Now let us move with the main application file MainApp.java, which is as follows

```
package com.java.spring.dec.tx;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.java.spring.dec.tx.bean.StudentMarks;
import com.java.spring.dec.tx.repo.StudentDAO;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("re-
sources/Beans.xml");

      StudentDAO studentJDBCTemplate = (StudentDAO)context.getBean("studentJDBCTem-
plate");

      System.out.println("------Records creation--------" );
      studentJDBCTemplate.create(101, "Zara", 11, 99, 2010);
      studentJDBCTemplate.create(102, "Nuha", 20, 97, 2010);
      studentJDBCTemplate.create(103, "Ayan", 25, 100, 2011);

      System.out.println("------Listing all the records--------" );
      List<StudentMarks> studentMarks = studentJDBCTemplate.listStudents();

      for (StudentMarks record : studentMarks) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         System.out.print(", Marks : " + record.getMarks());
         System.out.print(", Year : " + record.getYear());
         System.out.println(", Age : " + record.getAge());
      }
   }
}
```

```
Following is the configuration file Beans.xml
```

```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: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/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">

   <!-- Initialization for data source -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerData-
Source">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
<property name="username" value="system"/>
<property name="password" value="Admin@123"/>
</bean>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="create"/>
</tx:attributes>
</tx:advice>

   <aop:config>
   <aop:pointcut id="createOperation" expression="execution(*
com.java.spring.dec.tx.repo.StudentJDBCTemplate.create(..))"/>
   <aop:advisor advice-ref="txAdvice" pointcut-ref="createOperation"/>
   </aop:config>

   <!-- Initialization for TransactionManager -->
   <bean id="transactionManager" class="org.springframework.jdbc.datasource.Data-
SourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate" class="com.java.spring.dec.tx.repo.StudentJDBC-
Template">
   <property name = "dataSource" ref = "dataSource"/>
   </bean>
</beans>
```

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

```
------Records creation--------
Created Name = Zara, Age = 11
Exception in thread "main" java.lang.RuntimeException: simulate Error condition
```

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