Spring Bean Definition Inheritance & Bean Definition Template

- Create your project with name SpringEx and a package com.example. This should be under the src folder of your created project.
- Add the Spring Libraries that are required using the Add External JARs options.
- Create HelloWorld.java, HelloIndia.java and MainApp.java under the above made package.
- Write the Beans.xml configuration file under the src folder.
- Finally write code for all **Java files** and Bean config file and run the application as described.

HelloWorld.java

```
package com.example;
public class HelloWorld {
  private String message1;
  private String message2;
  public void setMessage1(String message){
    this.message1 = message;
  }
  public void setMessage2(String message){
    this.message2 = message;
  }
  public void getMessage1(){
    System.out.println("World Message1 : " + message1);
  }
  public void getMessage2(){
    System.out.println("World Message2 : " + message2);
  }
}
```

HelloIndia.java

```
package com.example;
public class HelloIndia {
  private String message1;
  private String message2;
  private String message3;
  public void setMessage1(String message){
    this.message1 = message;
  }
  public void setMessage2(String message){
    this.message2 = message;
  }
  public void setMessage3(String message){
    this.message3 = message;
  }
  public void getMessage1(){
    System.out.println("India Message1 : " + message1);
  }
  public void getMessage2(){
    System.out.println("India Message2 : " + message2);
  }
  public void getMessage3(){
    System.out.println("India Message3 : " + message3);
  }
}
```

MainApp.java

```
package com.example;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCon
text;
public class MainApp {
  public static void main(String[] args) {
    ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
    HelloWorld objA = (HelloWorld)
context.getBean("helloWorld");
    objA.getMessage1();
    objA.getMessage2();
    HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
    objB.getMessage1();
    objB.getMessage2();
    objB.getMessage3();
  }
}
```

Output:

```
World Message1 : Hello World!
World Message2 : Hello Second World!
India Message1 : Hello India!
India Message2 : Hello Second World!
India Message3 : Namaste India!
```

BeanTemplate – Bean Inheritance

```
<?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">
  <bean id = "beanTeamplate" abstract = "true">
    <property name = "message1" value = "Hello World!"/>
    <property name = "message2" value = "Hello Second World!"/>
    <property name = "message3" value = "Namaste India!"/>
  </bean>
    <bean id = "helloIndia" class = "com.example.HelloIndia"
parent = "beanTeamplate">
    <property name = "message1" value = "Hello India!"/>
    <property name = "message3" value = "Namaste India!"/>
  </bean>
</beans>
```

Spring AOP Overview

Most of the enterprise applications have some common crosscutting concerns that are applicable to different types of Objects and modules. Some of the common crosscutting concerns are logging, transaction management, data validation, etc. In Object Oriented Programming, modularity of application is achieved by Classes whereas in Aspect Oriented Programming application modularity is achieved by Aspects and they are configured to cut across different classes. Spring AOP takes out the direct dependency of crosscutting tasks from classes that we can't achieve through normal object oriented programming model. For example, we can have a separate class for logging but again the functional classes will have to call these methods to achieve logging across the application.

Aspect Oriented Programming Core Concepts

Before we dive into the implementation of Spring AOP implementation, we should understand the core concepts of AOP.

1. **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using `@Aspect` annotation.
2. **Join Point**: A join point is a specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
3. **Advice**: Advices are actions taken for a particular join point. In terms of programming, they are methods that get executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as Struts2 interceptors or Servlet Filters.

4. **Pointcut**: Pointcut is expressions that are matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.
5. **Target Object**: They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advice are included based on their configuration.
6. **AOP proxy**: Spring AOP implementation uses JDK dynamic proxy to create the Proxy classes with target classes and advice invocations, these are called AOP proxy classes. We can also use CGLIB proxy by adding it as the dependency in the Spring AOP project.
7. **Weaving**: It is the process of linking aspects with other objects to create the advised proxy objects. This can be done at compile time, load time or at runtime. Spring AOP performs weaving at the runtime.

AOP Advice Types

Based on the execution strategy of advice, they are of the following types.

1. **Before Advice**: These advices runs before the execution of join point methods. We can use `@Before` annotation to mark an advice type as Before advice.
2. **After (finally) Advice**: An advice that gets executed after the join point method finishes executing, whether normally or by throwing an exception. We can create after advice using `@After` annotation.
3. **After Returning Advice**: Sometimes we want advice methods to execute only if the join point method executes normally. We can use `@AfterReturning` annotation to mark a method as after returning advice.
4. **After Throwing Advice**: This advice gets executed only when join point method throws exception, we can use it to rollback

the transaction declaratively. We use `@AfterThrowing` annotation for this type of advice.

5. **Around Advice**: This is the most important and powerful advice. This advice surrounds the join point method and we can also choose whether to execute the join point method or not. We can write advice code that gets executed before and after the execution of the join point method. It is the responsibility of around advice to invoke the join point method and return values if the method is returning something. We use `@Around` annotation to create around advice methods.

XML Schema Based AOP with Spring

XML Schema Based AOP Example

To understand the above-mentioned concepts related to XML Schema-based AOP, let us write an example which will implement few of the advices. To write our example with few advices, let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

St	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 AOP specific libraries aspectjrt.jar, aspectjweaver.jar and aspectj.jar in the project.
4	Create Java classes Logging, Student and MainApp under the com.tutorialspoint package.
5	Create Beans configuration file Beans.xml under the src folder.

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

Here is the content of Logging.java file. This is actually a sample of aspect module which defines the methods to be called at various points.

```
package com.spring.AOP.XML;

public class Logging {
   /**
      * This is the method which I would like to
execute
      * before a selected method execution.
   */
   public void beforeAdvice(){
      System.out.println("Going to setup student
profile.");
   }

   /**
      * This is the method which I would like to
execute
      * after a selected method execution.
   */
   public void afterAdvice(){
      System.out.println("Student profile has been
setup.");
   }

   /**
      * This is the method which I would like to
execute
      * when any method returns.
   */
   public void afterReturningAdvice(Object retVal) {
      System.out.println("Returning:" +
retVal.toString() );
   }
   /**
      * This is the method which I would like to
execute
      * if there is an exception raised.
   */
   public void
AfterThrowingAdvice(IllegalArgumentException ex){
      System.out.println("There has been an
exception: " + ex.toString());
   }
}
```

Following is the content of the Student.java file

```
package com.spring.AOP.XML;

public class Student {
   private Integer age;
   private String name;

   public void setAge(Integer age) {
      this.age = age;
   }
   public Integer getAge() {
      System.out.println("Age : " + age );
      return age;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getName() {
      System.out.println("Name : " + name );
      return name;
   }
   public void printThrowException(){
       System.out.println("Exception raised");
      throw new IllegalArgumentException();
   }
}
```

Following is the content of the MainApp.java file

```
package com.spring.AOP.XML;

import
org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlAppli
cationContext;

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

      Student student = (Student)
context.getBean("student");
      student.getName();
      student.getAge();
      student.printThrowException();
   }
}
```

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

   <aop:config>
      <aop:aspect id = "log" ref = "logging">
```

```
         <aop:pointcut id = "selectAll"
            expression = "execution(*
com.spring.AOP.XML.*.*(..))"/>

         <aop:before pointcut-ref = "selectAll"
method = "beforeAdvice"/>
         <aop:after pointcut-ref = "selectAll" method
= "afterAdvice"/>
         <aop:after-returning pointcut-ref =
"selectAll"
            returning = "retVal" method =
"afterReturningAdvice"/>

         <aop:after-throwing pointcut-ref =
"selectAll"
            throwing = "ex" method =
"AfterThrowingAdvice"/>
      </aop:aspect>
   </aop:config>

   <!-- Definition for student bean -->
   <bean id = "student" class =
"com.spring.AOP.XML.Student">
      <property name = "name" value = "Zara" />
      <property name = "age" value = "11"/>
   </bean>

   <!-- Definition for logging aspect -->
   <bean id = "logging" class =
"com.spring.AOP.XML.Logging"/>

</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 message –

```
Going to setup student profile.
Name : Zara
Student profile has been setup.
Returning:Zara
```

```
Going to setup student profile.
Age : 11
Student profile has been setup.
Returning:11
Going to setup student profile.
Exception raised
Student profile has been setup.
There has been an exception:
java.lang.IllegalArgumentException
.....
other exception content
```

The above defined <aop:pointcut> selects all the methods defined under the package com.spring.AOP.XML. Let us suppose, you want to execute your advice before or after a particular method, you can define your pointcut to narrow down your execution by replacing stars (*) in pointcut definition with the actual class and method names. Following is a modified XML configuration file to show the concept –

```
<?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"
   xsi:schemaLocation = "http://
www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/
spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-
aop-3.0.xsd ">

   <aop:config>
      <aop:aspect id = "log" ref = "logging">
         <aop:pointcut id = "selectAll"
            expression = "execution(*
com.spring.AOP.XML.Student.getName(..))"/>
```

```
         <aop:before pointcut-ref = "selectAll"
method = "beforeAdvice"/>
         <aop:after pointcut-ref = "selectAll" method
= "afterAdvice"/>
      </aop:aspect>
   </aop:config>

   <!-- Definition for student bean -->
   <bean id = "student" class =
"com.spring.AOP.XML.Student">
      <property name = "name" value = "Zara" />
      <property name = "age" value = "11"/>
   </bean>

   <!-- Definition for logging aspect -->
   <bean id = "logging" class =
"com.spring.AOP.XML.Logging"/>

</beans>
```

If you execute the sample application with these configuration changes, it will print the following message –

```
Going to setup student profile.
Name : Zara
Student profile has been setup.
Age : 11
Exception raised
.....
other exception content
```

Spring JdbcTemplate Tutorial

Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

Problems of JDBC API

The problems of JDBC API are as follows:

- We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- We need to perform exception handling code on the database logic.
- We need to handle transaction.
- Repetition of all these codes from one to another database logic is a time consuming task.

Advantage of Spring JdbcTemplate

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

Spring Jdbc Approaches

Spring framework provides following approaches for JDBC database access:

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall

JdbcTemplate class

It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of excepion classes defined in the **org.springframework.dao** package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

Let's see the methods of spring JdbcTemplate class.

No.	Method	Description
1)	public int update(String query)	is used to insert, update and delete records.
2)	public int update(String query,Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
3)	public void execute(String query)	is used to execute DDL query.
4)	public T execute(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
5)	public T query(String sql, ResultSetExtractor rse)	is used to fetch records using ResultSetExtractor.
6)	public List query(String sql, RowMapper rse)	is used to fetch records using RowMapper.

```
CREATE TABLE Student(
   stud_id   INT NOT NULL AUTO_INCREMENT,
   stud_name VARCHAR(20) NOT NULL,
   mark  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Step s	Description
1	Create a project with a name Spring-JDBCTemplateand create a package com.spring.JDBCTemplate 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 Student, StudentMapper, StudentJDBCTemplate and MainApp under the com.spring.JDBCTemplate package.
6	Make sure you already created Student table 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.

Student.java - POJO Class

```
package com.spring.JDBCTemplate;
```

```
public class Student {
private int id,mark;
private String name;

public int getId() {
        return id;
}
public void setId(int id) {
        this.id = id;
}
public int getMark() {
        return mark;
}
public void setMark(int mark) {
        this.mark = mark;
}
public String getName() {
        return name;
}
public void setName(String name) {
        this.name = name;
}

public Student() {}
public Student(int id, String name,int mark) {
        super();
        this.id = id;
        this.mark = mark;
        this.name = name;
}

}
```

StudentDAO.java

```
package com.spring.JDBCTemplate;
import org.springframework.jdbc.core.JdbcTemplate;
import java.util.*;

public class StudentDAO {
    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
      this.jdbcTemplate = jdbcTemplate;
    }
    public int saveStudent(Student s) {
        String query = "insert into student values('"+s.getId()
+"','"+s.getName()+"','"+s.getMark()+"')";
        return jdbcTemplate.update(query);
    }
    public int updateStudent(Student s) {
        String query = "update student set
stud_name='"+s.getName()+"',mark='"+s.getMark()+"' where
stud_id='"+s.getId()+"'";
        return jdbcTemplate.update(query);
    }
    public int deleteStudent(Student s){
      String query="delete from student where
stud_id='"+s.getId()+"' ";
      return jdbcTemplate.update(query);
    }
    public List<Student> listStudents() {
       String SQL = "select * from student";
       List <Student> students = jdbcTemplate.query(SQL, new
StudentMapper());
       return students;
     }
}
```

StudentMapper.java

```
package com.spring.JDBCTemplate;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper{
	public Object mapRow(ResultSet rs, int rowNum) throws
SQLException {
		Student student = new Student();
	      student.setId(rs.getInt("stud_id"));
	      student.setName(rs.getString("stud_name"));
	      student.setMark(rs.getInt("mark"));

		return student;
	}

}
```

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 https://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.spring.JDBCTemplate</groupId>
  <artifactId>Spring-JDBCTemplate</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
  <dependency>
  		<groupId>org.springframework</groupId>
  		<artifactId>spring-context</artifactId>
  		<version>5.1.0.RELEASE</version>
  	</dependency>
<!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j
-->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.0.32</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
```

```xml
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.1</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.0.RELEASE</version>
</dependency>

  </dependencies>
</project>
```

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"
 xsi:schemaLocation="http://www.springframework.org/schema/
beans
   http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd">
   <bean id = "dataSource" class =
"org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name = "driverClassName" value =
"com.mysql.cj.jdbc.Driver"/>
      <property name = "url" value = "jdbc:mysql://localhost:3306/
mydb"/>
      <property name = "username" value = "root"/>
      <property name = "password" value = "Remember001"/>
      </bean>
      <bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="dataSource"></property>
</bean>

  <bean id = "jdbc"
      class = "com.spring.JDBCTemplate.StudentDAO">
      <property name = "jdbcTemplate" ref = "jdbcTemplate" />
   </bean>
   </beans>
```

Test.java

```
package com.spring.JDBCTemplate;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationCont
ext;
import java.util.*;

public class Test {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");
		StudentDAO dao =
(StudentDAO)context.getBean("jdbc");
		/*int status = dao.saveStudent(new Student(5,"Amit",76));
		System.out.println(status);*/

		/* int status=dao.updateStudent(new
Student(2,"Bhuvana",97));
	  System.out.println(status); */

	 /* Student s=new Student();
	  s.setId(2);
	  int status=dao.deleteStudent(s);
	  System.out.println(status);*/

	  System.out.println("------Listing Multiple Records--------" );
	    List<Student> students = dao.listStudents();

	    for (Student record : students) {
	       System.out.print("ID : " + record.getId() );
	       System.out.print(", Name : " + record.getName() );
	       System.out.println(", Age : " + record.getMark());
        }
    }
}

Output:

------Listing Multiple Records--------
ID : 1, Name : Rahul, Age : 80
ID : 3, Name : Harini, Age : 92
ID : 4, Name : Vinod, Age : 78
ID : 5, Name : Amit, Age : 76
```

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.

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 ofDefaultTransactionDefinition 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.

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.spring.PTM;
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.spring.PTM;

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;
	}
	 StudentMarks(){}
}
```

Following is the content of the StudentMarksMapper.java file

```
package com.spring.PTM;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

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.spring.PTM;


import java.util.List;

import org.springframework.dao.DataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.ResultSetExtractor;

import org.springframework.jdbc.support.rowset.SqlRowSet;

import org.springframework.transaction.PlatformTransactionManager;

import org.springframework.transaction.TransactionDefinition;

import org.springframework.transaction.TransactionStatus;

import org.springframework.transaction.support.DefaultTransactionDefinition;


import javax.sql.DataSource;


public class StudentJDBCTemplate implements StudentDAO{
   private DataSource dataSource;
      private JdbcTemplate jdbcTemplateObject;
      private PlatformTransactionManager transactionManager;

   public void setDataSource(DataSource ds) {
         this.dataSource = ds;
       this.jdbcTemplateObject = new JdbcTemplate(dataSource);

   }

   public void setTransactionManager(PlatformTransactionManager transactionManager) {
       this.transactionManager = transactionManager;
      }

   public void create(String name, int age, int marks, int year) {
         TransactionDefinition def = new DefaultTransactionDefinition();
                   TransactionStatus status = transactionManager.getTransaction(def);

       try {
          String SQL1 = "insert into Student (name, age) values (?, ?)";
          jdbcTemplateObject.update( SQL1, name, age);
```

```
            // Get the latest student id to be used in Marks table
            String SQL2 = "select max(id) from Student";
               int sid = jdbcTemplateObject.queryForObject(SQL2,
Integer.class);


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


             System.out.println("Created Name = " + name + ", Age =
" + age);
            transactionManager.commit(status);
         }
         catch (DataAccessException e) {
                System.out.println("Error in creating record, rolling
back");
            transactionManager.rollback(status);
            throw e;
         }
         return;


      }


      public List<StudentMarks> listStudents() {
           String SQL = "select * from Student, Marks where
Student.id=Marks.sid";
```

```
                List <StudentMarks> studentMarks = 
jdbcTemplateObject.query(SQL,
           new StudentMarksMapper());


         return studentMarks;
       }


     }
```

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

```
package com.spring.PTM;
import java.util.List;
import org.springframework.context.ApplicationContext;
import 
org.springframework.context.support.ClassPathXmlApplicationCont
ext;
import com.spring.PTM.StudentJDBCTemplate;


public class MainApp {


     public static void main(String[] args) {
          ApplicationContext context = new 
ClassPathXmlApplicationContext("Beans.xml");
          StudentJDBCTemplate studentJDBCTemplate =
(StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      System.out.println("------Records creation--------" );
      studentJDBCTemplate.create("Zara", 11, 99, 2010);
      studentJDBCTemplate.create("Nuha", 20, 97, 2010);
      studentJDBCTemplate.create("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 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.Dri
verManagerDataSource">
      <property name = "driverClassName"
value = "com.mysql.cj.jdbc.Driver"/>
      <property name = "url" value =
"jdbc:mysql://localhost:3306/sample"/>
      <property name = "username" value
= "root"/>
      <property name = "password" value
= "Remember001"/>
   </bean>

   <!-- Initialization for
TransactionManager -->
   <bean id = "transactionManager"
      class =
"org.springframework.jdbc.datasource.Dat
aSourceTransactionManager">
```

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

   <!-- Definition for
studentJDBCTemplate bean -->
   <bean id = "studentJDBCTemplate"
      class =
"com.spring.PTM.StudentJDBCTemplate">
      <property name = "dataSource" ref
= "dataSource" />
      <property name =
"transactionManager" ref =
"transactionManager" />
   </bean>

</beans>
```