spring JDBC

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

Spring JDBC Framework takes care of all the low-level details starting from opening the connection, preparing and executing the SQL statement, processing exceptions, handling transactions, and finally closing the connection.

This tutorial will take you through simple and practical approaches while learning JDBC framework provided by Spring.

Audience

This tutorial has been prepared for the beginners to help them understand the basic to advanced concepts related to JDBC framework of Spring.

Prerequisites

Before you start practicing the various types of examples given in this tutorial, we assume that you are already aware about computer programs and computer programming languages.

Copyright & Disclaimer

Table of Contents

Spring JDBC Basics

1. Spring JDBC – Overview

While working with database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections, etc. However, Spring JDBC Framework takes care of all the low-level details starting from opening the connection, preparing and executing the SQL statement, processing exceptions, handling transactions, and finally closing the connection.

What you have do is just define connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. In this tutorial, we will take classic and the most popular approach which makes use of JDBC Template class of the framework. This is the central framework class that manages all the database communication and exception handling.

JDBC Template Class

JDBC Template class executes SQL queries, updates statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JDBC Template class are threadsafe once configured. So, you can configure a single instance of a JDBC Template and then safely inject this shared reference into multiple DAOs.

A common practice when using the JDBC Template class is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes. The JDBC Template is created in the setter for the DataSource.

Data Access Object (DAO)

DAO stands for **Data Access Oject** which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies such as JDBC, Hibernate, JPA, or JDO in a consistent way.

2. Spring JDBC – Environment Setup

This chapter takes you through the process of setting up Spring-AOP on Windows and Linux based systems. Spring AOP can be easily installed and integrated with your current Java environment and MAVEN by following a few simple steps without any complex setup procedures. User administration is required while installation.

System Requirements

JDK	Java SE 2 JDK 1.5 or above
Memory	1 GB RAM (recommended)
Disk Space	No minimum requirement
Operating System Version	Windows XP or above, Linux

Let us now proceed with the steps to install Spring AOP.

Step 1: Verify your Java Installation

First of all, you need to have Java Software Development Kit (SDK) installed on your system. To verify this, execute any of the following two commands depending on the platform you are working on.

If the Java installation has been done properly, then it will display the current version and specification of your Java installation. A sample output is given in the following table.

Platform	Command	Sample Output
Windows	Open command console and type: **\>java -version**	Java version "1.7.0_60" Java (TM) SE Run Time Environment (build 1.7.0_60-b19) Java Hotspot (TM) 64-bit Server VM (build 24.60-b09,mixed mode)
Linux	Open command terminal and type: **$java -version**	java version "1.7.0_25" Open JDK Runtime Environment (rhel-2.3.10.4.el6_4-x86_64) Open JDK 64-Bit Server VM (build 23.7-b01, mixed mode)

We assume the readers of this tutorial have Java SDK version 1.7.0_60 installed on their system. In case you do not have Java SDK, download its current version from http://www.oracle.com/technetwork/java/javase/downloads/index.html and have it installed.

Step 2: Set your Java Environment

Set the environment variable JAVA_HOME to point to the base directory location where Java is installed on your machine. For example,

Platform	Description
Windows	Set JAVA_HOME to C:\ProgramFiles\java\jdk1.7.0_60
Linux	Export JAVA_HOME=/usr/local/java-current

Append the full path of Java compiler location to the System Path.

Platform	Description
Windows	Append the String "C:\Program Files\Java\jdk1.7.0_60\bin" to the end of the system variable PATH.
Linux	Export PATH=$PATH:$JAVA_HOME/bin/

Execute the command **java -version** from the command prompt as explained above.

Step 3: Download Maven Archive

Download Maven 3.3.3 from http://maven.apache.org/download.cgi

OS	Archive name
Windows	apache-maven-3.3.3-bin.zip
Linux	apache-maven-3.3.3-bin.tar.gz
Mac	apache-maven-3.3.3-bin.tar.gz

Step 4: Extract the Maven Archive

Extract the archive to the directory you wish to install Maven 3.3.3. The subdirectory apache-maven-3.3.3 will be created from the archive.

OS	Location (can be different based on your installation)
Windows	C:\Program Files\Apache Software Foundation\apache-maven-3.3.3
Linux	/usr/local/apache-maven
Mac	/usr/local/apache-maven

Step 5: Set Maven environment variables

Add M2_HOME, M2, MAVEN_OPTS to environment variables.

OS	Output
Windows	Set the environment variables using system properties. *M2_HOME=C:\Program Files\Apache Software Foundation\apache-maven-3.3.3* *M2=%M2_HOME%\bin* *MAVEN_OPTS=-Xms256m -Xmx512m*
Linux	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-3.3.3* *export M2=$M2_HOME/bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*
Mac	Open command terminal and set environment variables. *export M2_HOME=/usr/local/apache-maven/apache-maven-3.3.3* *export M2=$M2_HOME/bin* *export MAVEN_OPTS=-Xms256m -Xmx512m*

Step 6: Add Maven Bin Directory Location to System Path

Now append M2 variable to System Path.

OS	Output
Windows	Append the string ;%M2% to the end of the system variable, Path.
Linux	export PATH=$M2:$PATH
Mac	export PATH=$M2:$PATH

Step 7: Verify Maven installation

Now open console, execute the following **mvn** command.

OS	Task	Command
Windows	Open Command Console	c:\> mvn --version
Linux	Open Command Terminal	$ mvn --version
Mac	Open Terminal	machine:< joseph$ mvn --version

Finally, verify the output of the above commands, which should be something as follows:

OS	Output
Windows	Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T17:27:37+05:30) Maven home: C:\Program Files\Apache Software Foundation\apache-maven-3.3.3 Java version: 1.7.0_75, vendor: Oracle Corporation Java home: C:\Program Files\Java\jdk1.7.0_75\jre Default locale: en_US, platform encoding: Cp1252
Linux	Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T17:27:37+05:30) Maven home: /usr/local/apache-maven/apache-maven-3.3.3 Java version: 1.7.0_75, vendor: Oracle Corporation Java home: /usr/local/java-current/jdk1.7.0_75/jre
Mac	Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-22T17:27:37+05:30) Maven home: /usr/local/apache-maven/apache-maven-3.3.3 Java version: 1.7.0_75, vendor: Oracle Corporation Java home: /Library/Java/Home/jdk1.7.0_75/jre

Step 8: Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So, I would suggest you should have the latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from http://www.eclipse.org/downloads/. Once you have downloaded the installation, unpack the binary distribution into a convenient location. For example, in C:\eclipse on Windows, or /usr/local/eclipse on Linux/Unix. Finally, set PATH variable appropriately.

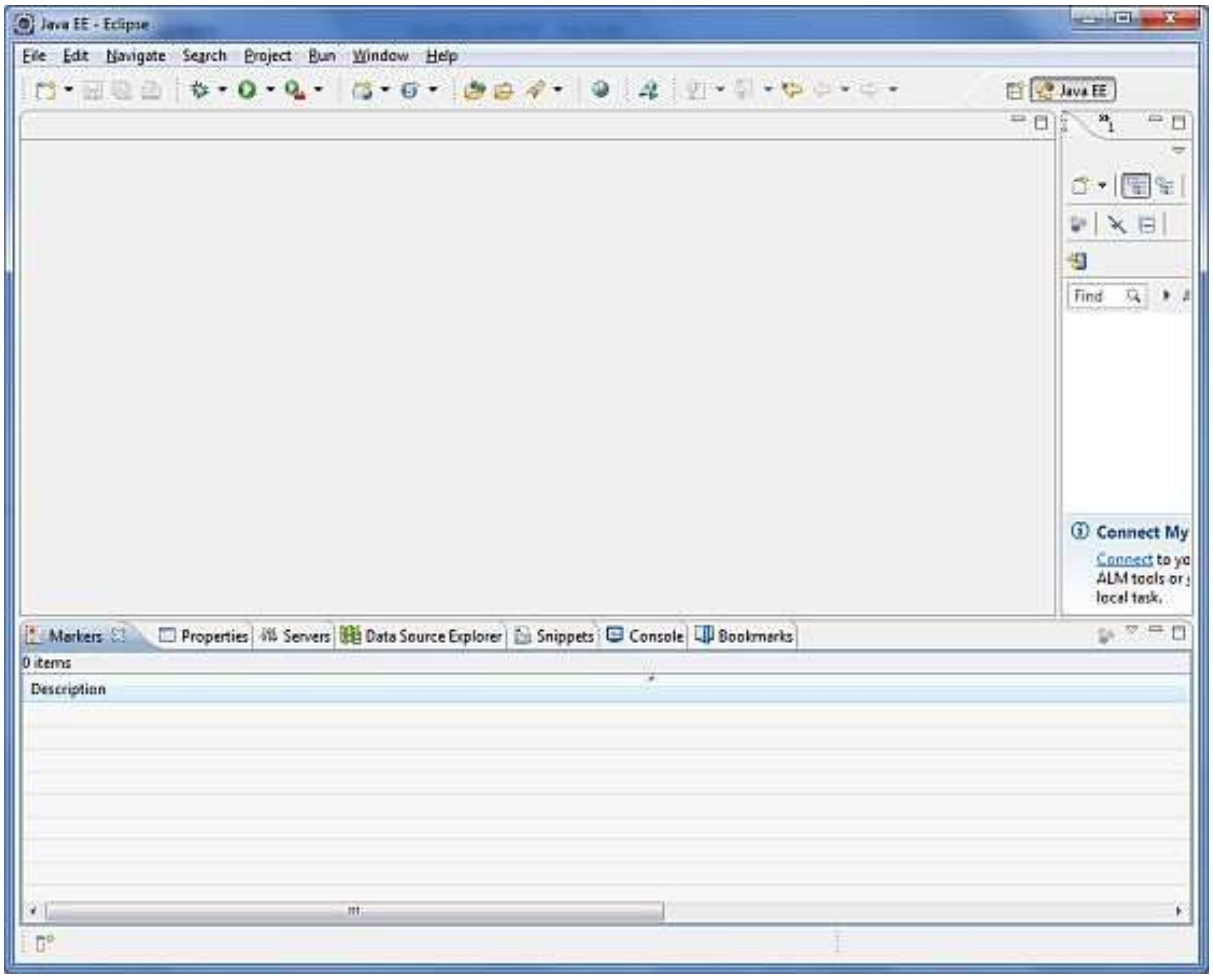
Eclipse can be started by executing the following commands on Windows machine, or you can simply double-click on eclipse.exe.

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine.

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display the following result.



Once you are done with this last step, you are ready to proceed for your first JDBC example which you will see in the next chapter.

3. Spring JDBC – Configure a Data Source

Let us create a database table **Student** in our database **TEST**. I assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

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

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code shown as follows:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
   <property name="username" value="root"/>
   <property name="password" value="admin"/>
</bean>
```

In the next chapter, we'll write the first application using the database configured.

4. Spring JDBC – First Application

To understand the concepts related to Spring JDBC framework with JDBC Template class, let us write a simple example which will implement Insert and Read operations on the following Student table.

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

Let us proceed to write a simple console based Spring JDBC Application, which will demonstrate JDBC concepts.

Create Project

Let's open the command console, go the C:\MVN directory and execute the following **mvn** command.

```
C:\MVN>mvn archetype:generate -DgroupId=com.tutorialspoint -DartifactId=Student
     -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Maven will start processing and will create the complete Java application project structure.

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources
@ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources
@ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
```

```
Downloading:
https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/ma

ven-archetype-quickstart/1.0/maven-archetype-quickstart-1.0.jar

Downloaded:
https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/mav

en-archetype-quickstart/1.0/maven-archetype-quickstart-1.0.jar (5 KB at 1.1
KB/sec)

Downloading:
https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/ma

ven-archetype-quickstart/1.0/maven-archetype-quickstart-1.0.pom

Downloaded:
https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/mav

en-archetype-quickstart/1.0/maven-archetype-quickstart-1.0.pom (703 B at 1.2
KB/sec)

[INFO] ----------------------------------------------------------------------

[INFO] Using following parameters for creating project from Old (1.x)
Archetype: maven-archetype-quickstart:1.0

[INFO] ----------------------------------------------------------------------

[INFO] Parameter: groupId, Value: com.tutorialspoint

[INFO] Parameter: packageName, Value: com.tutorialspoint

[INFO] Parameter: package, Value: com.tutorialspoint

[INFO] Parameter: artifactId, Value: Student

[INFO] Parameter: basedir, Value: C:\MVN

[INFO] Parameter: version, Value: 1.0-SNAPSHOT

[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\Student

[INFO] ----------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ----------------------------------------------------------------------

[INFO] Total time: 01:17 min

[INFO] Finished at: 2017-02-19T21:11:14+05:30

[INFO] Final Memory: 15M/114M

[INFO] ----------------------------------------------------------------------
```

Now go to C:/MVN directory. You'll see a Java application project created named student (as specified in artifactId). Update the POM.xml to include Spring JDBC dependencies. Add Student.java, StudentMapper.java, MainApp.java, StudentDAO.java and StudentJDBCTemplate.java files.

POM.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.tutorialspoint</groupId>
   <artifactId>Student</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Student</name>
   <url>http://maven.apache.org</url>
   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
      </dependency>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-jdbc</artifactId>
         <version>4.1.0.RELEASE</version>
      </dependency>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-context</artifactId>
         <version>4.1.4.RELEASE</version>
      </dependency>
   </dependencies>
</project>
```

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 table.
    */
   public void create(String name, Integer age);
   public Student getStudent(Integer id);
   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public void create(String name, Integer age) {
      String SQL = "insert into Student (name, age) values (?, ?)";

      jdbcTemplateObject.update( SQL, name, age);
      System.out.println("Created Record Name = " + name + " Age = " + age);
      return;
   }

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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);
      studentJDBCTemplate.create("Nuha", 2);
      studentJDBCTemplate.create("Ayan", 15);

      System.out.println("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Records Creation--------
Created Record Name = Zara Age = 11
Created Record Name = Nuha Age = 2
Created Record Name = Ayan Age = 15
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
```

Basic CRUD Examples

5. Spring JDBC – Create Query

The following example will demonstrate how to create a query using Insert query with the help of Spring JDBC. We'll insert a few records in Student Table.

Syntax

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

Where,

- **insertQuery** - Insert query having placeholders.
- **jdbcTemplateObject** - StudentJDBCTemplate object to insert student object in database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will insert a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 table.
    */
```

```
   public void create(String name, Integer age);
   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public void create(String name, Integer age) {
      String insertQuery = "insert into Student (name, age) values (?, ?)";
```

```
      jdbcTemplateObject.update( insertQuery, name, age);
      System.out.println("Created Record Name = " + name + " Age = " + age);
      return;
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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);
      studentJDBCTemplate.create("Nuha", 2);
      studentJDBCTemplate.create("Ayan", 15);

      System.out.println("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
```

```
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Records Creation--------
Created Record Name = Zara Age = 11
Created Record Name = Nuha Age = 2
Created Record Name = Ayan Age = 15
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
```

6. Spring JDBC – Read Query

Following example will demonstrate how to read a query using Spring JDBC. We'll read available records in Student Table.

Syntax

```
String selectQuery = "select * from Student";
List <Student> students = jdbcTemplateObject.query(selectQuery, new StudentMapper());
```

Where,

- **selectQuery** - Select query to read students.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student object from database.
- **StudentMapper** - StudentMapper is a RowMapper object to map each fetched record to student object.

To understand above mentioned concepts related to Spring JDBC, let us write an example which will select a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
```

```
                                    new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
```

7. Spring JDBC – Update a Query

Following example will demonstrate how to update a query using Spring JDBC. We'll update the available records in Student Table.

Syntax

```
String updateQuery = "update Student set age = ? where id = ?";
jdbcTemplateObject.update(updateQuery, age, id);
```

Where,

- **updateQuery** - Update query to update student with place holders.
- **jdbcTemplateObject** - StudentJDBCTemplate object to update student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 update
```

```
    * a record into the Student table.
    */
   public void update(Integer id, Integer age);

   /**
    * This is the method to be used to list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

```
   }

   public void update(Integer id, Integer age){
      String SQL = "update Student set age = ? where id = ?";
      jdbcTemplateObject.update(SQL, age, id);
      System.out.println("Updated Record with ID = " + id );
      return;
   }

    public Student getStudent(Integer id) {
      String SQL = "select * from Student where id = ?";
      Student student = jdbcTemplateObject.queryForObject(SQL,
                        new Object[]{id}, new StudentMapper());
      return student;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("----Updating Record with ID = 2 -----" );
      studentJDBCTemplate.update(2, 20);
```

```
      System.out.println("----Listing Record with ID = 2 -----" );
      Student student = studentJDBCTemplate.getStudent(2);
      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
----Updating Record with ID = 2 -----
Updated Record with ID = 2
----Listing Record with ID = 2 -----
ID : 2, Name : Nuha, Age : 20
```

8. Spring JDBC – Delete Query

The following example will demonstrate how to delete a query using Spring JDBC. We'll delete one of the available records in Student Table.

Syntax

```
String deleteQuery = "delete from Student where id = ?";
jdbcTemplateObject.update(deleteQuery, id);
```

Where,

- **deleteQuery** - Delete query to delete student with placeholders.
- **jdbcTemplateObject** - StudentJDBCTemplate object to delete student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will delete a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
```

```
     * all the records from the Student table.
     */
    public List<Student> listStudents();
    /**
     * This is the method to be used to delete
     * a record from the Student table corresponding
     * to a passed student id.
     */
    public void delete(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
```

```
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
                                new StudentMapper());
      return students;
   }

   public void delete(Integer id){
      String SQL = "delete from Student where id = ?";
      jdbcTemplateObject.update(SQL, id);
      System.out.println("Deleted Record with ID = " + id );
      return;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("----Delete Record with ID = 2 -----" );
```

```
      studentJDBCTemplate.delete(2);

      System.out.println("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
----Updating Record with ID = 2 -----
Updated Record with ID = 2
----Listing Record with ID = 2 -----
ID : 2, Name : Nuha, Age : 20
```

Advanced JDBC Examples

9. Spring JDBC – Calling a Stored Procedure

Following example will demonstrate how to call a stored procedure using Spring JDBC. We'll read one of the available records in Student Table by calling a stored procedure. We'll pass an id and receive a student record.

Syntax

```
SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("getRecord");

SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);

Map<String, Object> out = jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out_name"));

student.setAge((Integer) out.get("out_age"));
```

Where,

- **jdbcCall** - SimpleJdbcCall object to represent a stored procedure.
- **in** - SqlParameterSource object to pass a parameter to a stored procedure.
- **student** - Student object.
- **out** - Map object to represent the output of stored procedure call result.

The **SimpleJdbcCall** class can be used to call a stored procedure with IN and OUT parameters. You can use this approach while working with either of the RDBMS such as Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.

To understand the approach, consider the following MySQL stored procedure, which takes student Id and returns the corresponding student's name and age using OUT parameters. Let us create this stored procedure in TEST database using MySQL command prompt –

```
DELIMITER $$


DROP PROCEDURE IF EXISTS `TEST`.`getRecord` $$

CREATE PROCEDURE `TEST`.`getRecord` (

IN in_id INTEGER,

OUT out_name VARCHAR(20),

OUT out_age  INTEGER)

BEGIN

   SELECT name, age

   INTO out_name, out_age
```

```
   FROM Student where id = in_id;
END $$


DELIMITER ;
```

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will call a stored procedure. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

```
public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;

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

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

   public Student getStudent(Integer id) {
      SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("getRecord");

      SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);
      Map<String, Object> out = jdbcCall.execute(in);
```

```
      Student student = new Student();
      student.setId(id);
      student.setName((String) out.get("out_name"));
      student.setAge((Integer) out.get("out_age"));
      return student;
   }
}
```

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The execute method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure.

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      Student student = studentJDBCTemplate.getStudent(1);
      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Name : Zara, Age : 11
```

10. Spring JDBC – Calling a Stored Function

Following example will demonstrate how to call a stored function using Spring JDBC. We'll read one of the available records in Student Table by calling a stored function. We'll pass an id and receive a student name.

Syntax

```
SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withFunctionName("get_student_name");


SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);
String name = jdbcCall.executeFunction(String.class, in);
Student student = new Student();
student.setId(id);
student.setName(name);
```

Where,

- **in** - SqlParameterSource object to pass a parameter to a stored function.
- **jdbcCall** - SimpleJdbcCall object to represent a stored function.
- **jdbcTemplateObject** - StudentJDBCTemplate object to call a stored function from database.
- **student** - Student object.

The **SimpleJdbcCall** class can be used to call a stored function with IN parameter and a return value. You can use this approach while working with either of the RDBMS such as Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.

To understand the approach, consider the following MySQL stored procedure, which takes student Id and returns the corresponding student's name. So let us create this stored function in your TEST database using MySQL command prompt –

```
DELIMITER $$


DROP FUNCTION IF EXISTS `TEST`.`get_student_name` $$
CREATE FUNCTION `get_student_name` (in_id INTEGER)
RETURNS varchar(200)
BEGIN
DECLARE out_name VARCHAR(200);
   SELECT name
   INTO out_name
```

```
   FROM Student where id = in_id;

RETURN out_name
;

DELIMITER ;
```

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will call a stored function. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
```

```
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;

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

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

   public Student getStudent(Integer id) {
      SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withFunctionName("get_student_name");

      SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);
      String name = jdbcCall.executeFunction(String.class, in);
      Student student = new Student();
      student.setId(id);
```

```
      student.setName(name);
      return student;
   }
}
```

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored function. The executeFunction method takes the IN parameters and returns a String as specified in the stored function.

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      Student student = studentJDBCTemplate.getStudent(1);

      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
   }
}
```

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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Name : Zara
```

11. Spring JDBC – Handling a BLOB

Following example will demonstrate how to update a BLOB using an Update Query with the help of Spring JDBC. We'll update the available records in Student Table.

Student Table

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

Syntax

```
MapSqlParameterSource in = new MapSqlParameterSource();
in.addValue("id", id);
in.addValue("image",  new SqlLobValue(new ByteArrayInputStream(imageData),
   imageData.length, new DefaultLobHandler()), Types.BLOB);
String SQL = "update Student set image = :image where id = :id";
NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
jdbcTemplateObject.update(SQL, in);
```

Where,

- **in** - SqlParameterSource object to pass a parameter to update a query.
- **SqlLobValue** - Object to represent an SQL BLOB/CLOB value parameter.
- **jdbcTemplateObject** - NamedParameterJdbcTemplate object to update student object in database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 update
    * a record into the Student table.
    */
   public void updateImage(Integer id, byte[] imageData);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
   private Integer id;
   private byte[] image;

   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 byte[] getImage() {
      return image;
   }
   public void setImage(byte[] image) {
      this.image = image;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      student.setImage(rs.getBytes("image"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.jdbc.core.support.SqlLobValue;
import org.springframework.jdbc.support.lob.DefaultLobHandler;
import java.io.ByteArrayInputStream;
import java.sql.Types;

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

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

   public void updateImage(Integer id, byte[] imageData) {
      MapSqlParameterSource in = new MapSqlParameterSource();
      in.addValue("id", id);
      in.addValue("image",  new SqlLobValue(new ByteArrayInputStream(imageData),
         imageData.length, new DefaultLobHandler()), Types.BLOB);
      String SQL = "update Student set image = :image where id = :id";

      NamedParameterJdbcTemplate jdbcTemplateObject = new NamedParameterJdbcTemplate(dataSource);
      jdbcTemplateObject.update(SQL, in);
      System.out.println("Updated Record with ID = " + id );
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      byte[] imageData = {0,1,0,8,20,40,95};
      studentJDBCTemplate.updateImage(1, imageData);
   }
}
```

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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
```

```
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Updated Record with ID = 1
```

You can check the byte[] stored by querying the database.

12. Spring JDBC – Handling a CLOB

Following example will demonstrate how to update a CLOB using an Update Query with the help of Spring JDBC. We'll update the available records in Student Table.

Student Table

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

Syntax

```
MapSqlParameterSource in = new MapSqlParameterSource();
in.addValue("id", id);
in.addValue("description",  new SqlLobValue(description, new
DefaultLobHandler()), Types.CLOB);

String SQL = "update Student set description = :description where id = :id";
NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
jdbcTemplateObject.update(SQL, in);
```

Where,

- **in** - SqlParameterSource object to pass a parameter to update a query.
- **SqlLobValue** - Object to represent an SQL BLOB/CLOB value parameter.
- **jdbcTemplateObject** - NamedParameterJdbcTemplate object to update student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example, which will update a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 update
    * a record into the Student table.
    */
   public void updateDescription(Integer id, String description);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
   private Integer id;
   private String description;
   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 String getDescription() {
      return description;
   }
   public void setDescription(String description) {
      this.description = description;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
```

```
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      student.setDescription(rs.getString("description"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.jdbc.core.support.SqlLobValue;
import org.springframework.jdbc.support.lob.DefaultLobHandler;
import java.io.ByteArrayInputStream;
import java.sql.Types;

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

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

   public void updateDescription(Integer id, String description) {

      MapSqlParameterSource in = new MapSqlParameterSource();
      in.addValue("id", id);
      in.addValue("description",  new SqlLobValue(description,
```

```
         new DefaultLobHandler()), Types.CLOB);

      String SQL = "update Student set description = :description where id = :id";
      NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
      jdbcTemplateObject.update(SQL, in);
      System.out.println("Updated Record with ID = " + id );
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      studentJDBCTemplate.updateDescription(1,
         "This can be a very long text upto 4 GB of size.");
   }
}
```

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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Updated Record with ID = 1
```

You can check the description stored by querying the database.

Spring JDBC Batch Examples

13. Spring JDBC – Batch Operation

Following example will demonstrate how to make a batch update using Spring JDBC. We'll update the available records in Student table in a single batch operation.

Syntax

```
String SQL = "update Student set age = ? where id = ?";
int[] updateCounts = jdbcTemplateObject.batchUpdate(SQL,
   new BatchPreparedStatementSetter() {
      public void setValues(PreparedStatement ps, int i) throws SQLException {
         ps.setInt(1, students.get(i).getAge());

         ps.setInt(2, students.get(i).getId());
      }
      public int getBatchSize() {
         return students.size();
      }
   });
```

Where,

- **SQL** - Update query to update student's age.
- **jdbcTemplateObject** - StudentJDBCTemplate object to update student object in database.
- **BatchPreparedStatementSetter** - Batch executor, set values in PerparedStatement per item identified by list of objects student and index **i**. getBatchSize() returns the size of the batch.
- **updateCounts** - Int array containing updated row count per update query.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a batch operation. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();

   public void batchUpdate(final List<Student> students);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.sql.PreparedStatement;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import java.sql.SQLException;

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

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }

   public void batchUpdate(final List<Student> students){
      String SQL = "update Student set age = ? where id = ?";
      int[] updateCounts = jdbcTemplateObject.batchUpdate(SQL,
         new BatchPreparedStatementSetter() {
            public void setValues(PreparedStatement ps, int i) throws SQLException {
                 ps.setInt(1, students.get(i).getAge());
                ps.setInt(2, students.get(i).getId());
            }

            public int getBatchSize() {
               return students.size();
```

```
            }
         });
      System.out.println("Records updated!");
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

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

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

      StudentJDBCTemplate studentJDBCTemplate =
         (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      List<Student> initialStudents = studentJDBCTemplate.listStudents();
      System.out.println("Initial Students");
      for(Student student2: initialStudents){
         System.out.print("ID : " + student2.getId() );
         System.out.println(", Age : " + student2.getAge());
      }

      Student student = new Student();
      student.setId(1);
      student.setAge(10);

      Student student1 = new Student();
```

```
      student1.setId(3);
      student1.setAge(10);

      List<Student> students = new ArrayList<Student>();
      students.add(student);
      students.add(student1);

      studentJDBCTemplate.batchUpdate(students);

      List<Student> updatedStudents = studentJDBCTemplate.listStudents();
      System.out.println("Updated Students");
      for(Student student3: updatedStudents){
         System.out.print("ID : " + student3.getId() );
         System.out.println(", Age : " + student3.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
```

```
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Initial Students
ID : 1, Age : 11
ID : 3, Age : 15
Records updated!
Updated Students
ID : 1, Age : 10
ID : 3, Age : 10
```

14. Spring JDBC – Objects Batch Operation

Following example will demonstrate how to make a batch update using objects in Spring JDBC. We'll update the available records in Student table in a single batch operation.

Syntax

```
String SQL = "update Student set age = :age where id = :id";

SqlParameterSource[] batch =
SqlParameterSourceUtils.createBatch(students.toArray());

NamedParameterJdbcTemplate jdbcTemplateObject = new NamedParameterJdbcTemplate(dataSource);

int[] updateCounts = jdbcTemplateObject.batchUpdate(SQL,batch);

System.out.println("records updated!");
```

Where,

- **SQL** - Update query to update student's age.
- **jdbcTemplateObject** - StudentJDBCTemplate object to update student object in database.
- **batch** - SqlParameterSource object to represent a batch of object.
- **updateCounts** - Int array containing updated row count per update query.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a batch operation. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();

   public void batchUpdate(final List<Student> students);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
```

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

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

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
                                new StudentMapper());
      return students;
   }

   public void batchUpdate(final List<Student> students){
      String SQL = "update Student set age = :age where id = :id";
      SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(students.toArray());
      NamedParameterJdbcTemplate jdbcTemplateObject = new NamedParameterJdbcTemplate(dataSource);

      int[] updateCounts = jdbcTemplateObject.batchUpdate(SQL,batch);
      System.out.println("Records updated!");
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class MainApp {
   public static void main(String[] args) {
```

```
      ApplicationContext context =
         new ClassPathXmlApplicationContext("Beans.xml");

      StudentJDBCTemplate studentJDBCTemplate =
         (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      List<Student> initialStudents = studentJDBCTemplate.listStudents();
      System.out.println("Initial Students");
      for(Student student2: initialStudents){
         System.out.print("ID : " + student2.getId() );
         System.out.println(", Age : " + student2.getAge());
      }

      Student student = new Student();
      student.setId(1);
      student.setAge(15);

      Student student1 = new Student();
      student1.setId(3);
      student1.setAge(16);

      List<Student> students = new ArrayList<Student>();
      students.add(student);
      students.add(student1);

      studentJDBCTemplate.batchUpdate(students);

      List<Student> updatedStudents = studentJDBCTemplate.listStudents();
      System.out.println("Updated Students");
      for(Student student3: updatedStudents){
         System.out.print("ID : " + student3.getId() );
         System.out.println(", Age : " + student3.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Initial Students
ID : 1, Age : 10
ID : 3, Age : 10
Records updated!
Updated Students
ID : 1, Age : 15
ID : 3, Age : 16
```

15. Spring JDBC – Multiple Batches Operation

Following example will demonstrate how to make multiple batch updates in a single call using Spring JDBC. We'll update the available records in Student table in a multiple batch operation where batch size is 1.

Syntax

```
String SQL = "update Student set age = ? where id = ?";
int[][] updateCounts = jdbcTemplateObject.batchUpdate(SQL,students,1,
   new ParameterizedPreparedStatementSetter<Student>() {
      public void setValues(PreparedStatement ps, Student student)
         throws SQLException {
         ps.setInt(1, student.getAge());
         ps.setInt(2, student.getId());
      }
   });
```

Where,

- **SQL** - Update query to update student's age.
- **jdbcTemplateObject** - StudentJDBCTemplate object to update student object in the database.
- **ParameterizedPreparedStatementSetter** - Batch executor, set values in PerparedStatement per item identified by the list of objects student.
- **updateCounts** - Int[][] array containing updated row count per update query per batch.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update multiple batch operation. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
   public void batchUpdate(final List<Student> students);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;
import java.sql.PreparedStatement;
import java.util.List;
import javax.sql.DataSource;
```

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ParameterizedPreparedStatementSetter;
import java.sql.SQLException;

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

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

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }

   public void batchUpdate(final List<Student> students){
      String SQL = "update Student set age = ? where id = ?";
      int[][] updateCounts = jdbcTemplateObject.batchUpdate(SQL,students,1,
         new ParameterizedPreparedStatementSetter<Student>() {
            public void setValues(PreparedStatement ps, Student student)
              throws SQLException {
              ps.setInt(1, student.getAge());

              ps.setInt(2, student.getId());
            }
         });
      System.out.println("Records updated!");
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.ArrayList;
import java.util.List;

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

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

      StudentJDBCTemplate studentJDBCTemplate =
         (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      List<Student> initialStudents = studentJDBCTemplate.listStudents();
      System.out.println("Initial Students");
      for(Student student2: initialStudents){
         System.out.print("ID : " + student2.getId() );
         System.out.println(", Age : " + student2.getAge());
      }

      Student student = new Student();
      student.setId(1);
      student.setAge(17);

      Student student1 = new Student();
      student1.setId(3);
      student1.setAge(18);

      List<Student> students = new ArrayList<Student>();
      students.add(student);
      students.add(student1);
```

```
      studentJDBCTemplate.batchUpdate(students);

      List<Student> updatedStudents = studentJDBCTemplate.listStudents();
      System.out.println("Updated Students");
      for(Student student3: updatedStudents){
         System.out.print("ID : " + student3.getId() );
         System.out.println(", Age : " + student3.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Initial Students
ID : 1, Age : 15
ID : 3, Age : 16
records updated!
Updated Students
ID : 1, Age : 17
ID : 3, Age : 18
```

Spring JDBC Objects

16. Spring JDBC – JDBC Template Class

The **org.springframework.jdbc.core.JdbcTemplate** class is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving the application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the **org.springframework.dao** package.

Class Declaration

Following is the declaration for org.springframework.jdbc.core.JdbcTemplate class –

```
public class JdbcTemplate
   extends JdbcAccessor
      implements JdbcOperations
```

Usage

Step 1: Create a JdbcTemplate object using a configured datasource.

Step 2: Use JdbcTemplate object methods to make database operations.

Example

Following example will demonstrate how to read a query using JdbcTemplate class. We'll read the available records in Student Table.

Syntax

```
String selectQuery = "select * from Student";
List <Student> students = jdbcTemplateObject.query(selectQuery, new StudentMapper());
```

Where,

- **selectQuery** - Select query to read students.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student object from the database.
- **StudentMapper** - StudentMapper is a RowMapper object to map each fetched record to the student object.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will select a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

```
   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
                                new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
```

17. Spring JDBC – PreparedStatementSetter Interface

The **org.springframework.jdbc.core.PreparedStatementSetter** interface acts as a general callback interface used by the JdbcTemplate class. This interface sets values on a PreparedStatement provided by the JdbcTemplate class, for each of a number of updates in a batch using the same SQL.

Implementations are responsible for setting any necessary parameters. SQL with placeholders will already have been supplied. It's easier to use this interface than PreparedStatementCreator. The JdbcTemplate will create the PreparedStatement, with the callback only being responsible for setting parameter values.

Interface Declaration

Following is the declaration for org.springframework.jdbc.core.PreparedStatementSetter interface –

```
public interface PreparedStatementSetter
```

Usage

Step 1 - Create a JdbcTemplate object using a configured datasource.

Step 2 - Use JdbcTemplate object methods to make database operations while passing PreparedStatementSetter object to replace place holders in query.

Example

Following example will demonstrate how to read a query using JdbcTemplate class and PreparedStatementSetter interface. We'll read available record of a student in Student Table.

Syntax

```
final String SQL = "select * from Student where id = ? ";
List <Student> students = jdbcTemplateObject.query(SQL, new
PreparedStatementSetter() {
   public void setValues(PreparedStatement preparedStatement) throws SQLException {
      preparedStatement.setInt(1, id);
  }
},
new StudentMapper());
```

Where,

- **SQL** - Select query to read students.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student object from database.
- **PreparedStatementSetter** - PreparedStatementSetter object to set parameters in query.

- **StudentMapper** - StudentMapper is a RowMapper object to map each fetched record to student object.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will select a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down a record from the Student
     * table corresponding to a passed student id.
     */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

```
   private Integer id;
   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public Student getStudent(final Integer id) {
      final String SQL = "select * from Student where id = ? ";
      List <Student> students = jdbcTemplateObject.query(SQL, new
PreparedStatementSetter() {
            public void setValues(PreparedStatement preparedStatement) throws SQLException {
                  preparedStatement.setInt(1, id);
            }
      },
      new StudentMapper());
      return students.get(0);
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

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

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

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      Student student = studentJDBCTemplate.getStudent(1);
      System.out.print("ID : " + student.getId() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Age : 17
```

18. Spring JDBC – ResultSetExtractor Interface

The **org.springframework.jdbc.core.ResultSetExtractor** interface is a callback interface used by JdbcTemplate's query methods. Implementations of this interface perform the actual work of extracting results from a ResultSet, but don't need to worry about exception handling.

SQLExceptions will be caught and handled by the calling JdbcTemplate. This interface is mainly used within the JDBC framework itself. A RowMapper is usually a simpler choice for ResultSet processing, mapping one result object per row instead of one result object for the entire ResultSet.

Interface Declaration

Following is the declaration for org.springframework.jdbc.core.ResultSetExtractor interface –

```
public interface ResultSetExtractor
```

Usage

Step 1 - Create a JdbcTemplate object using a configured datasource.

Step 2 - Use JdbcTemplate object methods to make database operations while parsing the resultset using ResultSetExtractor.

Example

Following example will demonstrate how to read a query using JdbcTemplate class and ResultSetExtractor interface. We'll read available record of a student in Student Table.

Syntax

```
public List<Student> listStudents() {
   String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
         new ResultSetExtractor<List<Student>>(){
            public List<Student> extractData(
               ResultSet rs) throws SQLException, DataAccessException {
               List<Student> list=new ArrayList<Student>();
               while(rs.next()){
                  Student student = new Student();
                  student.setId(rs.getInt("id"));
                  student.setName(rs.getString("name"));
                  student.setAge(rs.getInt("age"));
                  student.setDescription(rs.getString("description"));
```

```
                student.setImage(rs.getBytes("image"));
                list.add(student);
            }
            return list;
        }
    });
  return students;
}
```

Where,

- **SQL** - Select query to read students.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student object from database.
- **ResultSetExtractor** - ResultSetExtractor object to parse resultset object.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will select a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import java.util.ArrayList;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

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

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

   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
            new ResultSetExtractor<List<Student>>(){
               public List<Student> extractData(
                  ResultSet rs) throws SQLException, DataAccessException {
                  List<Student> list=new ArrayList<Student>();
                  while(rs.next()){
                     Student student = new Student();
                     student.setId(rs.getInt("id"));
                     student.setName(rs.getString("name"));
                     student.setAge(rs.getInt("age"));
                     student.setDescription(rs.getString("description"));
                     student.setImage(rs.getBytes("image"));
                     list.add(student);
                  }
                  return list;
               }
```

```
            });
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

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

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

      StudentJDBCTemplate studentJDBCTemplate =
         (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      List<Student> students = studentJDBCTemplate.listStudents();

      for(Student student: students){
         System.out.print("ID : " + student.getId() );
         System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Age : 17
ID : 3, Age : 18
```

19. Spring JDBC – RowMapper Interface

The **org.springframework.jdbc.core.RowMapper<T>** interface is used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object. SQLExceptions if any thrown will be caught and handled by the calling JdbcTemplate.

Interface Declaration

Following is the declaration for **org.springframework.jdbc.core.RowMapper<T>** interface –

```
public interface RowMapper<T>
```

Usage

Step 1 - Create a JdbcTemplate object using a configured datasource.

Step 2 - Create a StudentMapper object implementing RowMapper interface.

Step 3 - Use JdbcTemplate object methods to make database operations while using StudentMapper object.

Following example will demonstrate how to read a query using spring jdbc. We'll map read records from Student Table to Student object using StudentMapper object.

Syntax

```
String SQL = "select * from Student";
List <Student> students = jdbcTemplateObject.query(SQL,
   new StudentMapper());
```

Where,

- **SQL** - Read query to read all student records.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student records from database.
- **StudentMapper** - StudentMapper object to map student records to student objects.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will read a query and map result using StudentMapper object. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;


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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJDBCTemplate implements StudentDao {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 17
ID : 3, Name : Ayan, Age : 18
```

20. Spring JDBC - NamedParameterJdbcTemplate Class

The **org.springframework.jdbc.core.NamedParameterJdbcTemplate** class is a template class with a basic set of JDBC operations, allowing the use of named parameters rather than traditional '?' placeholders. This class delegates to a wrapped JdbcTemplate once the substitution from named parameters to JDBC style '?' placeholders is done at execution time. It also allows to expand a list of values to the appropriate number of placeholders.

Interface Declaration

Following is the declaration for **org.springframework.jdbc.core.NamedParameterJdbcTemplate** class:

```
public class NamedParameterJdbcTemplate
   extends Object
      implements NamedParameterJdbcOperations
```

Syntax

```
MapSqlParameterSource in = new MapSqlParameterSource();
in.addValue("id", id);
in.addValue("description",  new SqlLobValue(description, new
DefaultLobHandler()), Types.CLOB);

String SQL = "update Student set description = :description where id = :id";
NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
jdbcTemplateObject.update(SQL, in);
```

Where,

- **in** - SqlParameterSource object to pass a parameter to update a query.
- **SqlLobValue** - Object to represent an SQL BLOB/CLOB value parameter.
- **jdbcTemplateObject** - NamedParameterJdbcTemplate object to update student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 update
    * a record into the Student table.
    */
   public void updateDescription(Integer id, String description);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

public class Student {
   private Integer age;
   private String name;
   private Integer id;
   private String description;
   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 String getDescription() {
      return description;
   }
   public void setDescription(String description) {
      this.description = description;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
```

```
      student.setAge(rs.getInt("age"));
       student.setDescription(rs.getString("description"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.jdbc.core.support.SqlLobValue;
import org.springframework.jdbc.support.lob.DefaultLobHandler;
import java.io.ByteArrayInputStream;
import java.sql.Types;

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

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

   public void updateDescription(Integer id, String description) {

      MapSqlParameterSource in = new MapSqlParameterSource();
      in.addValue("id", id);
      in.addValue("description",  new SqlLobValue(description,
         new DefaultLobHandler()), Types.CLOB);
```

```
      String SQL = "update Student set description = :description where id = :id";
      NamedParameterJdbcTemplate jdbcTemplateObject = new
NamedParameterJdbcTemplate(dataSource);
      jdbcTemplateObject.update(SQL, in);
      System.out.println("Updated Record with ID = " + id );
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      studentJDBCTemplate.updateDescription(1,
         "This can be a very long text upto 4 GB of size.");
   }
}
```

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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
Updated Record with ID = 1
```

You can check the description stored by querying the database.

21. Spring JDBC – SimpleJdbcInsert Class

The **org.springframework.jdbc.core.SimpleJdbcInsert** class is a multi-threaded, reusable object providing easy insert capabilities for a table. It provides meta data processing to simplify the code needed to construct a basic insert statement. The actual insert is being handled using Spring's JdbcTemplate.

Class Declaration

Following is the declaration for **org.springframework.jdbc.core.SimpleJdbcInsert** class–

```
public class SimpleJdbcInsert
   extends AbstractJdbcInsert
      implements SimpleJdbcInsertOperations
```

Following example will demonstrate how to insert a query using Spring JDBC. We'll insert one record in Student Table using SimpleJdbcInsert object.

Syntax

```
jdbcInsert = new SimpleJdbcInsert(dataSource).withTableName("Student");
Map<String,Object> parameters = new HashMap<String,Object>();
parameters.put("name", name);
parameters.put("age", age);
jdbcInsert.execute(parameters);
```

Where,

- **jdbcInsert** - SimpleJdbcInsert object to insert record in student table.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student object in database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will insert a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 table.
    */
   public void create(String name, Integer age);
   /**
    * This is the method to be used to list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

```
   private Integer id;

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;


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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
```

```
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

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

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

public class StudentJDBCTemplate implements StudentDao {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;
   SimpleJdbcInsert jdbcInsert;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
      this.jdbcInsert = new SimpleJdbcInsert(dataSource).withTableName("Student");
   }

   public void create(String name, Integer age) {
      Map<String,Object> parameters = new HashMap<String,Object>();
      parameters.put("name", name);
      parameters.put("age", age);
      jdbcInsert.execute(parameters);
      System.out.println("Created Record Name = " + name + " Age = " + age);
```

```
      return;
   }
   public List<Student> listStudents() {
      String SQL = "select * from Student";
      List <Student> students = jdbcTemplateObject.query(SQL,
         new StudentMapper());
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("Nuha", 2);

      System.out.println("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Records Creation--------
Created Record Name = Nuha Age = 12
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 17
ID : 3, Name : Ayan, Age : 18
ID : 4, Name : Nuha, Age : 12
```

22. Spring JDBC – SimpleJdbcCall Class

The **org.springframework.jdbc.core.SimpleJdbcCall** class is a multi-threaded, reusable object representing a call to a stored procedure or a stored function. It provides meta data processing to simplify the code needed to access basic stored procedures/functions.

All you need to provide is the name of the procedure/function and a map containing the parameters when you execute the call. The names of the supplied parameters will be matched up with in and out parameters declared when the stored procedure was created.

Class Declaration

Following is the declaration for **org.springframework.jdbc.core.SimpleJdbcCall** class –

```
public class SimpleJdbcCall
   extends AbstractJdbcCall
      implements SimpleJdbcCallOperations
```

Following example will demonstrate how to call a stored procedure using Spring SimpleJdbcCall. We'll read one of the available records in Student Table by calling a stored procedure. We'll pass an id and receive a student record.

Syntax

```
SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("getRecord");

SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);

Map<String, Object> out = jdbcCall.execute(in);

Student student = new Student();

student.setId(id);

student.setName((String) out.get("out_name"));

student.setAge((Integer) out.get("out_age"));
```

Where,

- **jdbcCall** - SimpleJdbcCall object to represent a stored procedure.
- **in** - SqlParameterSource object to pass a parameter to a stored procedure.
- **student** - Student object.
- **out** - Map object to represent the output of stored procedure call result.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will call a stored procedure. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;

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

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

   public Student getStudent(Integer id) {
      SimpleJdbcCall jdbcCall = new
SimpleJdbcCall(dataSource).withProcedureName("getRecord");

      SqlParameterSource in = new MapSqlParameterSource().addValue("in_id", id);
      Map<String, Object> out = jdbcCall.execute(in);

      Student student = new Student();
      student.setId(id);
      student.setName((String) out.get("out_name"));
      student.setAge((Integer) out.get("out_age"));
      return student;
   }
}
```

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The execute

method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure.

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

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

      StudentJDBCTemplate studentJDBCTemplate =
      (StudentJDBCTemplate)context.getBean("studentJDBCTemplate");

      Student student = studentJDBCTemplate.getStudent(1);

      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Name : Zara, Age : 11
```

23. Spring JDBC – SqlQuery Class

The **org.springframework.jdbc.object.SqlQuery** class provides a reusable operation object representing a SQL query.

Class Declaration

Following is the declaration for **org.springframework.jdbc.object.SqlQuery** class –

```
public abstract class SqlQuery<T>
   extends SqlOperation
```

Usage

Step 1 - Create a JdbcTemplate object using a configured datasource.

Step 2 - Create a StudentMapper object implementing RowMapper interface.

Step 3 - Use JdbcTemplate object methods to make database operations while using SqlQuery object.

Following example will demonstrate how to read a Query using SqlQuery Object. We'll map read records from Student Table to Student object using StudentMapper object.

Syntax

```
String sql = "select * from Student";
SqlQuery<Student> sqlQuery = new SqlQuery<Student>() {
   @Override
   protected RowMapper<Student> newRowMapper(Object[] parameters,
      Map<?, ?> context) {
      return new StudentMapper();
   }
};
sqlQuery.setDataSource(dataSource);
sqlQuery.setSql(sql);
List <Student> students = sqlQuery.execute();
```

Where,

- **SQL** - Read query to read all student records.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student records from the database.
- **StudentMapper** - StudentMapper object to map the student records to student objects.

- **SqlQuery** - SqlQuery object to query student records and map them to student objects.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will read a query and map the result using StudentMapper object. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * all the records from the Student table.
    */
   public List<Student> listStudents();
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

```
   private Integer id;

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
```

```
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

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

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.object.SqlQuery;

public class StudentJDBCTemplate implements StudentDao {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

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

   public List<Student> listStudents() {
      String sql = "select * from Student";
      SqlQuery<Student> sqlQuery = new SqlQuery<Student>() {

      @Override
      protected RowMapper<Student> newRowMapper(Object[] parameters,
            Map<?, ?> context) {
            return new StudentMapper();
         }
      };
```

```
      sqlQuery.setDataSource(dataSource);
      sqlQuery.setSql(sql);
      List <Student> students = sqlQuery.execute();
      return students;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("------Listing Multiple Records--------" );
      List<Student> students = studentJDBCTemplate.listStudents();
      for (Student record : students) {
         System.out.print("ID : " + record.getId() );
         System.out.print(", Name : " + record.getName() );
         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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
------Listing Multiple Records--------
ID : 1, Name : Zara, Age : 17
ID : 3, Name : Ayan, Age : 18
ID : 4, Name : Nuha, Age : 12
```

24. Spring JDBC – SqlUpdate Class

The **org.springframework.jdbc.object.SqlUpdate** class provides reusable operation object representing a SQL update.

Class Declaration

Following is the declaration for **org.springframework.jdbc.object.SqlUpdate** class –

```
public abstract class SqlUpdate<T>
   extends SqlOperation
```

Usage

Step 1 - Create a JdbcTemplate object using a configured datasource.

Step 2 - Create a StudentMapper object implementing RowMapper interface.

Step 3 - Use JdbcTemplate object methods to carry out database operations while using SqlUpdate object.

Following example will demonstrate how to update a Query using SqlUpdate Object. We'll map update records from Student Table to Student object using StudentMapper object.

Syntax

```
String SQL = "update Student set age = ? where id = ?";

SqlUpdate sqlUpdate = new SqlUpdate(dataSource,SQL);
sqlUpdate.declareParameter(new SqlParameter("age", Types.INTEGER));
sqlUpdate.declareParameter(new SqlParameter("id", Types.INTEGER));
sqlUpdate.compile();

sqlUpdate.update(age.intValue(),id.intValue());
```

Where,

- **SQL** - Update query to update student records.
- **jdbcTemplateObject** - StudentJDBCTemplate object to read student records the from database.
- **StudentMapper** - StudentMapper object to map student records to student objects.
- **sqlUpdate** - SqlUpdate object to update student records.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will read a query and map result using StudentMapper object. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 update
    * a record into the Student table.
    */
   public void update(Integer id, Integer age);

   /**
    * This is the method to be used to list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

```
public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class StudentJDBCTemplate implements StudentDao {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

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

   public void update(Integer id, Integer age){
      String SQL = "update Student set age = ? where id = ?";
      SqlUpdate sqlUpdate = new SqlUpdate(dataSource,SQL);
      sqlUpdate.declareParameter(new SqlParameter("age", Types.INTEGER));
      sqlUpdate.declareParameter(new SqlParameter("id", Types.INTEGER));
```

```
      sqlUpdate.compile();

      sqlUpdate.update(age.intValue(),id.intValue());
      System.out.println("Updated Record with ID = " + id );
      return;
   }

    public Student getStudent(Integer id) {
      String SQL = "select * from Student where id = ?";
      Student student = jdbcTemplateObject.queryForObject(SQL,
                        new Object[]{id}, new StudentMapper());
      return student;
   }
}
```

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.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("----Updating Record with ID = 1 -----" );
      studentJDBCTemplate.update(1, 10);

      System.out.println("----Listing Record with ID = 1 -----" );
      Student student = studentJDBCTemplate.getStudent(1);
      System.out.print("ID : " + student.getId() );
```

```
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>

   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
----Updating Record with ID = 1 -----
Updated Record with ID = 1
----Listing Record with ID = 1 -----
ID : 1, Name : Zara, Age : 10
```

25. Spring JDBC – StoredProcedure Class

The **org.springframework.jdbc.core.StoredProcedure** class is the superclass for object abstractions of RDBMS stored procedures. This class is abstract and it is intended that subclasses will provide a typed method for invocation that delegates to the supplied execute(java.lang.Object...) method. The inherited SQL property is the name of the stored procedure in the RDBMS.

Class Declaration

Following is the declaration for **org.springframework.jdbc.core.StoredProcedure** class–

```
public abstract class StoredProcedure
   extends SqlCall
```

Following example will demonstrate how to call a stored procedure using Spring StoredProcedure. We'll read one of the available records in Student Table by calling a stored procedure. We'll pass an id and receive a student record.

Syntax

```
class StudentProcedure extends StoredProcedure{
   public StudentProcedure(DataSource dataSource, String procedureName){
      super(dataSource,procedureName);
      declareParameter(new SqlParameter("in_id", Types.INTEGER));
      declareParameter(new SqlOutParameter("out_name", Types.VARCHAR));
      declareParameter(new SqlOutParameter("out_age", Types.INTEGER));
      compile();
   }

   public Student execute(Integer id){
      Map<String, Object> out = super.execute(id);
      Student student = new Student();
      student.setId(id);
      student.setName((String) out.get("out_name"));
      student.setAge((Integer) out.get("out_age"));
      return student;
   }
}
```

Where,

- **StoredProcedure** - StoredProcedure object to represent a stored procedure.
- **StudentProcedure** - StudentProcedure object extends StoredProcedure to declare input, output variable, and map result to Student object.
- **student** - Student object.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will call a stored procedure. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Step	Description
1	Update the project *Student* created under chapter ***Spring JDBC - First Application***.
2	Update the bean configuration 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 list down
    * a record from the Student table corresponding
    * to a passed student id.
    */
   public Student getStudent(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.tutorialspoint;

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

   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;
   }
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.tutorialspoint;

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

public class StudentMapper implements RowMapper<Student> {
```

```
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setId(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));
      return student;
   }
}
```

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

```
package com.tutorialspoint;

import java.sql.Types;
import java.util.List;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.object.StoredProcedure;

public class StudentJDBCTemplate implements StudentDao {
   private DataSource dataSource;
   private JdbcTemplate jdbcTemplateObject;

   public void setDataSource(DataSource dataSource) {
      this.dataSource = dataSource;
      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
   }
   public Student getStudent(Integer id) {
      StudentProcedure studentProcedure = new StudentProcedure(dataSource, "getRecord");
      return studentProcedure.execute(id);
```

```
   }
}

class StudentProcedure extends StoredProcedure{
   public StudentProcedure(DataSource dataSource, String procedureName){
      super(dataSource,procedureName);
      declareParameter(new SqlParameter("in_id", Types.INTEGER));
      declareParameter(new SqlOutParameter("out_name", Types.VARCHAR));
      declareParameter(new SqlOutParameter("out_age", Types.INTEGER));
      compile();
   }

   public Student execute(Integer id){
      Map<String, Object> out = super.execute(id);
      Student student = new Student();
      student.setId(id);
      student.setName((String) out.get("out_name"));
      student.setAge((Integer) out.get("out_age"));
      return student;
   }
}
```

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The execute method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure.

Following is the content of the **MainApp.java** file.

```
package com.tutorialspoint;

import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.tutorialspoint.StudentJDBCTemplate;

public class MainApp {
```

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

      Student student = studentJDBCTemplate.getStudent(1);
      System.out.print("ID : " + student.getId() );
      System.out.print(", Name : " + student.getName() );
      System.out.println(", Age : " + student.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.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
      <property name="username" value="root"/>
      <property name="password" value="admin"/>
   </bean>
   <!-- Definition for studentJDBCTemplate bean -->
   <bean id="studentJDBCTemplate"
      class="com.tutorialspoint.StudentJDBCTemplate">
      <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 message.

```
ID : 1, Name : Zara, Age : 10
```