Spring JdbcTemplate Example

In this post, we feature a comprehensive Example on Spring JdbcTemplate. When we need to interface with databases the Spring JDBC framework provides solutions to all the low-level details, like open/close a connection, prepare and execute SQL statements, process exceptions and handle transactions. Thus, the only thing a developer must do is just define connection parameters and specify the SQL statement to be executed.

Spring JDBC provides several approaches and different classes to form the basis for a JDBC database access. The most popular approach makes use of

```
JdbcTemplate
```

class. This is the central framework class that manages all the database communication and exception handling.

In order to work with JDBC in Spring we will make use of the Data Access Objects. DAOs in Spring are commonly used for database interaction, using data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way. DAOs provide a means to read and write data to the database and they can expose this functionality through an interface by which the rest of the application can access them.

Here, we shall begin by showing you a simple example of JDBC integration. We will use a simple DAO, to make a simple insert and select to a database. We will continue with examples of the

```
JdbcTemplate
```

class to make SQL operations even easier. We will make use of the methods

```
JdbcTemplate
```

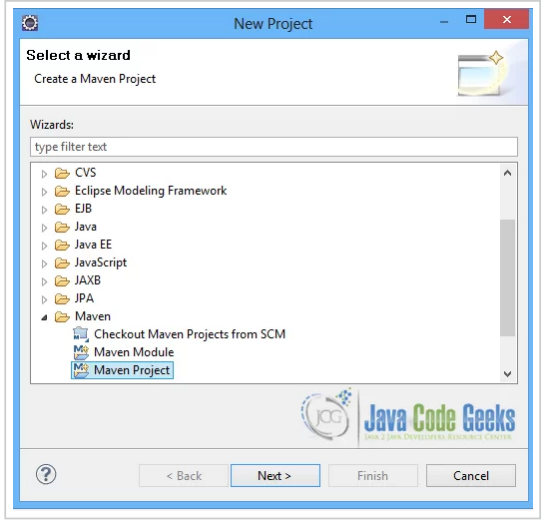
class provides to perform various selects and updates to the database.

Our preferred development environment is Eclipse. We are using Eclipse Juno (4.2) version, along with Maven Integration plugin version 3.1.0. You can download Eclipse from here and Maven Plugin for Eclipse from here. The installation of Maven plugin for Eclipse is out of the scope of this tutorial and will not be discussed. We are also using Spring version 3.2.3 and the JDK 7_u_21. The database used in the example is MySQL Database Server 5.6.

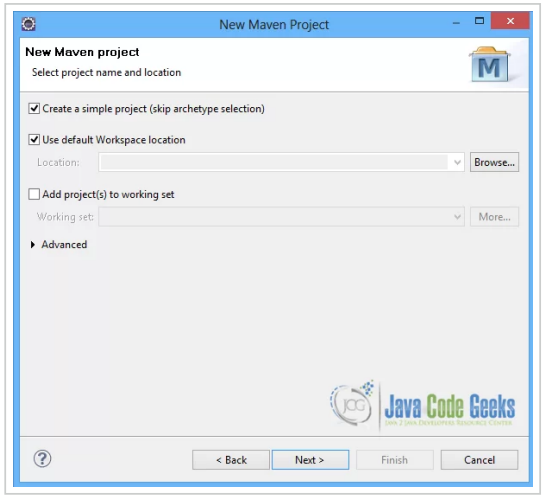
Let's begin,

1. Create a new Maven project

Go to File -> Project ->Maven -> Maven Project.



In the "Select project name and location" page of the wizard, make sure that "Create a simple project (skip archetype selection)" option is **checked**, hit "Next" to continue with default values.



In the "Enter an artifact id" page of the wizard, you can define the name and main package of your project. We will set the "Group Id" variable to

```
"com.javacodegeeks.snippets.enterprise"
```

and the "Artifact Id" variable to

```
"springexample"
```

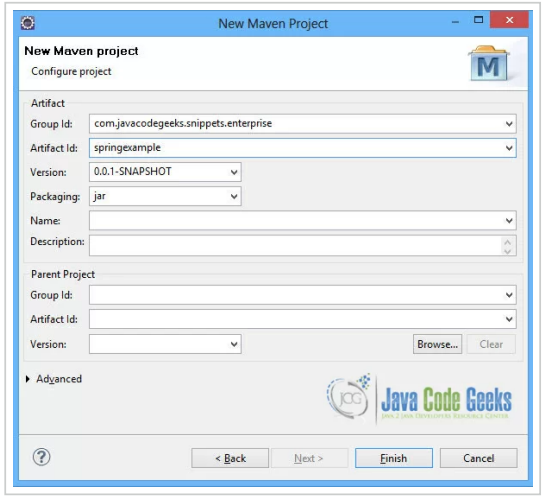
. The aforementioned selections compose the main project package as

```
"com.javacodegeeks.snippets.enterprise.springexample"
```

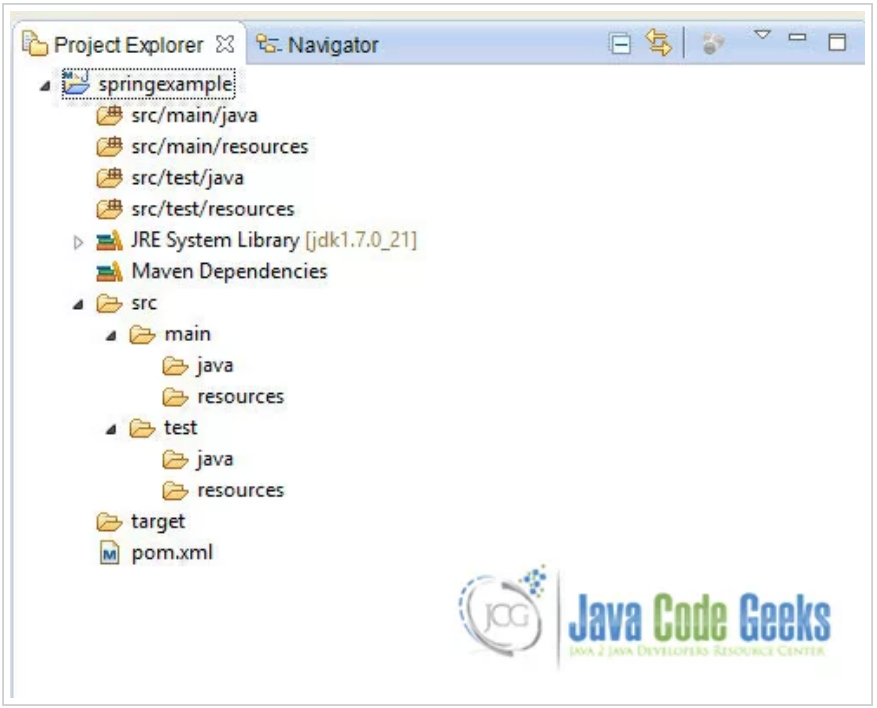
and the project name as

```
"springexample"
```

. Hit “Finish” to exit the wizard and to create your project.



The Maven project structure is shown below:



It consists of the following folders:

- /src/main/java folder, that contains source files for the dynamic content of the application,
- /src/test/java folder contains all source files for unit tests,
- /src/main/resources folder contains configurations files,
- /target folder contains the compiled and packaged deliverables,
- the pom.xml is the project object model (POM) file. The single file that contains all project related configuration.

2. Add Spring 3.2.3 dependency

- Locate the "Properties" section at the "Overview" page of the POM editor and perform the following changes:
  Create a new property with name **org.springframework.version** and value **3.2.3.RELEASE**.
- Navigate to the "Dependencies" page of the POM editor and create the following dependencies (you should fill the "GroupId", "Artifact Id" and "Version" fields of the "Dependency Details" section at that page):
  Group Id : **org.springframework** Artifact Id : **spring-web** Version : **${org.springframework.version}**

Alternatively, you can add the Spring dependencies in Maven's

```
pom.xml
```

file, by directly editing it at the "Pom.xml" page of the POM editor, as shown below:

pom.xml:

```
01  <project xmlns="http://maven.apache.org/POM/4.0.0"; xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
03      <modelVersion>4.0.0</modelVersion>
04      <groupId>com.javacodegeeks.snippets.enterprise</groupId>
05      <artifactId>springexample</artifactId>
06      <version>0.0.1-SNAPSHOT</version>
07
08      <dependencies>
09          <dependency>
10              <groupId>org.springframework</groupId>
11              <artifactId>spring-core</artifactId>
12              <version>${spring.version}</version>
13          </dependency>
14          <dependency>
15              <groupId>org.springframework</groupId>
16              <artifactId>spring-context</artifactId>
17              <version>${spring.version}</version>
18          </dependency>
19      </dependencies>
20
21      <properties>
22          <spring.version>3.2.3.RELEASE</spring.version>
23      </properties>
24  </project>
```

As you can see Maven manages library dependencies declaratively. A local repository is created (by default under {user_home}/.m2 folder) and all required libraries are downloaded and placed there from public repositories. Furthermore intra – library dependencies are automatically resolved and manipulated.

3. Add the JDBC dependencies

The dependencies needed for Spring JDBC are the ones below:

pom.xml

```
01  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
02      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
03      <modelVersion>4.0.0</modelVersion>
04      <groupId>com.javacodegeeks.snippets.enterprise</groupId>
05      <artifactId>springexample</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.26</version>
        </dependency>
         <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
      </dependency>
    </dependencies>

    <properties>
        <spring.version>3.2.3.RELEASE</spring.version>
    </properties>
</project>
```

4. Execute a simple insert and select example

Let's start, by creating a simple table in the database. We create a simple

```
Employee
```

table, that has three columns. The SQL statement that is executed in MySQL Workbench is shown below:

Create a table

```
CREATE TABLE `Employee` (
  `ID` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `NAME` VARCHAR(100) NOT NULL,
  `AGE` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`CUST_ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

We also create a class,

```
Employee.java
```

, that has three fields that are the same to the columns of

```
Employee
```

table.

Employee.java

```
package com.javacodegeeks.snippets.enterprise;
```

```
02
03  public class Employee {
04
05      private int id;
06
07      private String name;
08
09      private int age;
10
11      public Employee(){
12
13      }
14
15      public Employee(int id, String name, int age) {
16          this.id = id;
17          this.name = name;
18          this.age = age;
19      }
20
21      public int getId() {
22          return id;
23      }
24
25      public void setId(int id) {
26          this.id = id;
27      }
28
29      public String getName() {
30          return name;
31      }
32
33      public void setName(String name) {
34          this.name = name;
35      }
36
37      public int getAge() {
38          return age;
39      }
40
41      public void setAge(int age) {
42          this.age = age;
43      }
44
45      @Override
46      public String toString() {
47          return Employee ["id= "+ id + ", name= "+ name + ", age= "+ age
48                  +"]";
49      }
50
51  }
```

The DAO created to interact between the java class and the table is the

`EmployeeDAOImpl.java`

. It has two methods,

`insert(Employee employee)`

and

`findById(int id)`

, that implement the insert and select statements to the database.

Both methods use the

```
DataSource
```

class, a utility class that provides connection to the database. It is part of the JDBC specification and allows a container or a framework to hide connection pooling and transaction management issues from the application code.

In addition, both methods open a

```
Connection
```

to the database and use the

```
PreparedStatement
```

, that is an object representing a precompiled SQL statement.

EmployeeDAO.java

```
1  package com.javacodegeeks.snippets.enterprise.dao;
2
3  import com.javacodegeeks.snippets.enterprise.Employee;
4
5  public interface EmployeeDAO {
6
7          public void insert(Employee employee);
8          public Employee findById(int id);
9  }
```

EmployeeDAOImpl.java

```
01  package com.javacodegeeks.snippets.enterprise.dao.impl;
02
03  import java.sql.Connection;
04  import java.sql.PreparedStatement;
05  import java.sql.ResultSet;
06  import java.sql.SQLException;
07
08  import javax.sql.DataSource;
09
10  import com.javacodegeeks.snippets.enterprise.Employee;
11  import com.javacodegeeks.snippets.enterprise.dao.EmployeeDAO;
12
13  public class EmployeeDAOImpl implements EmployeeDAO
14  {
15      private DataSource dataSource;
16
17      public void setDataSource(DataSource dataSource) {
18          this.dataSource = dataSource;
19      }
20
21      public void insert(Employee employee){
22
23          String sql = "INSERT INTO employee " +
24                  "(ID, NAME, AGE) VALUES (?, ?, ?)";
25          Connection conn = null;
26
27          try {
28              conn = dataSource.getConnection();
29              PreparedStatement ps = conn.prepareStatement(sql);
30              ps.setInt(1, employee.getId());
```

```
            ps.setString(2, employee.getName());
            ps.setInt(3, employee.getAge());
            ps.executeUpdate();
            ps.close();

        } catch (SQLException e) {
            throw new RuntimeException(e);

        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }

    public Employee findById(int id){

        String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";

        Connection conn = null;

        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setInt(1, id);
            Employee employee = null;
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                employee = new Employee(
                    rs.getInt("ID"),
                    rs.getString("NAME"),
                    rs.getInt("AGE")
                );
            }
            rs.close();
            ps.close();
            return employee;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                conn.close();
                } catch (SQLException e) {}
            }
        }
    }
}
```

The

`Datasource`

is configured in

`applicationContext.xml`

file. All parameters needed for the connection to the database are set here. It is defined in other bean definitions using the

`ref`

element.

applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/sche
    xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/


    <bean id="employeeDAO" class="com.javacodegeeks.snippets.enterprise.dao.impl.EmployeeDAOImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <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="root" />
    </bean>
</beans>
```

We can run the example, using the

`App.java`

class. We load the

`employeeBean`

and then create a new

`Employee`

object. We first insert it to the table and then make a select to find it.

App.java

```
package com.javacodegeeks.snippets.enterprise;

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

import com.javacodegeeks.snippets.enterprise.dao.EmployeeDAO;
import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class App {

    public static void main(String[] args) {

            ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.
            EmployeeDAO employeeDAO = (EmployeeDAO) context.getBean("employeeDAO");
            Employee employee1 = new Employee(123, "javacodegeeks", 30);
            employeeDAO.insert(employee1);
            Employee employee2 = employeeDAO.findById(123);
            System.out.println(employee2);
            context.close();
    }
}
```

The output is shown below:

Output

```
Employee [id=123, name=javacodegeeks, age=30]
```

5. Use of JdbcTemplate Class

The

`JdbcTemplate`

class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It handles the creation and release of resources, thus avoiding errors such as forgetting to close the connection. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the

`org.springframework.dao`

package.

A simple insert example in

`JDBCEmployeeDAOImpl.java`

class, using the

`JdbcTemplate`

class is shown below:

JDBCEmployeeDAO.java

```
1  package com.javacodegeeks.snippets.enterprise.dao;
2
3  import com.javacodegeeks.snippets.enterprise.Employee;
4
5  public interface JDBCEmployeeDAO {
6
7      public void insert(Employee employee);
8
9  }
```

JDBCEmployeeDAOImpl.java

```
01  package com.javacodegeeks.snippets.enterprise.dao.impl;
02
03  import javax.sql.DataSource;
04
05  import org.springframework.jdbc.core.JdbcTemplate;
06
07  import com.javacodegeeks.snippets.enterprise.Employee;
08  import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;
09
10  public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
11      private DataSource dataSource;
12      private JdbcTemplate jdbcTemplate;
```

```
13
14      public void setDataSource(DataSource dataSource) {
15          this.dataSource = dataSource;
16      }
17
18      public void insert(Employee employee){
19
20          String sql = "INSERT INTO EMPLOYEE " +
21              "(ID, NAME, AGE) VALUES (?, ?, ?)";
22
23          jdbcTemplate = new JdbcTemplate(dataSource);
24
25          jdbcTemplate.update(sql, new Object[] { employee.getId(),
26                  employee.getName(), employee.getAge()
27          });
28      }
29  }
```

5.1 Select examples

Now, let's see how to make use of the

`JdbcTemplate`

class to make select statements in different ways. We can add new queries in

`EmployeeDAOImpl.java`

class as shown in the following cases.

5.1.1 Select a single row

In order to make a single row select we can implement the

`RowMapper`

interface. Thus, we can override the

`mapRow(ResultSet rs, int rowNum)`

method of

`RowMapper`

to map the table fields to the object, as shown below:

EmployeeRowMapper.java

```
01  package com.javacodegeeks.snippets.enterprise;
02
03  import java.sql.ResultSet;
04  import java.sql.SQLException;
05
06  import org.springframework.jdbc.core.RowMapper;
07
08  @SuppressWarnings("rawtypes")
09  public class EmployeeRowMapper implements RowMapper {
10          public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
```

```
            Employee employee = new Employee();
            employee.setId(rs.getInt("ID"));
            employee.setName(rs.getString("NAME"));
            employee.setAge(rs.getInt("AGE"));
            return employee;
        }
}
```

We add a new method

```
findById(int id)
```

to

```
JDBCEmployeeDAO.java
```

and

```
JDBCEmployeeDAOImpl.java
```

. Here, the

```
queryForObject(String sql, Object[] args, RowMapper rowMapper)
```

method of

```
JDBCTemplate
```

class will create the select with the given sql statement and the given id. It will then map the result that is a single row to the

```
Employee
```

object using the

```
EmployeeRowMapper.java
```

implementation.

JDBCEmployeeDAO.java

```
package com.javacodegeeks.snippets.enterprise.dao;

import com.javacodegeeks.snippets.enterprise.Employee;

public interface JDBCEmployeeDAO {

    public void insert(Employee employee);
    public Employee findById(int id);

}
```

JDBCEmployeeDAOImpl.java

```
package com.javacodegeeks.snippets.enterprise.dao.impl;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

```

```
import com.javacodegeeks.snippets.enterprise.Employee;
import com.javacodegeeks.snippets.enterprise.EmployeeRowMapper;
import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;

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

    public void insert(Employee employee){

        String sql = "INSERT INTO EMPLOYEE " +
            "(ID, NAME, AGE) VALUES (?, ?, ?)";

        jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new Object[] { employee.getId(),
                employee.getName(), employee.getAge()
        });
    }

    @SuppressWarnings({ "unchecked" })
    public Employee findById(int id){

        String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";

        jdbcTemplate = new JdbcTemplate(dataSource);
        Employee employee = (Employee) jdbcTemplate.queryForObject(
                sql, new Object[] { id }, new EmployeeRowMapper());

        return employee;
    }

}
```

Another way to get a single result is to use the

`BeanPropertyRowMapper`

implementation of

`RowMapper`

that converts a row into a new instance of the specified mapped target class. The

`BeanPropertyRowMapper`

maps a row column value to a property of the object by matching their names.

JDBCEmployeeDAOImpl.java

```
package com.javacodegeeks.snippets.enterprise.dao.impl;

import javax.sql.DataSource;

import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import com.javacodegeeks.snippets.enterprise.Employee;
import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;
```

```
public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;

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

    public void insert(Employee employee){

        String sql = "INSERT INTO EMPLOYEE " +
            "(ID, NAME, AGE) VALUES (?, ?, ?)";

        jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new Object[] { employee.getId(),
                employee.getName(), employee.getAge()
        });
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public Employee findById(int id){

        String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";

        jdbcTemplate = new JdbcTemplate(dataSource);
        Employee employee = (Employee) jdbcTemplate.queryForObject(
                sql, new Object[] { id }, new BeanPropertyRowMapper(Employee.class));

        return employee;
    }

}
```

We add the

```
jdbcEmployeeDAOBean
```

to

```
applicationContext.xml
```

:

applicationContext.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop" xmlns:context="http://www.springframework.org/sche
    xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/


    <bean id="employeeDAO" class="com.javacodegeeks.snippets.enterprise.dao.impl.EmployeeDAOImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="jdbcEmployeeDAO" class="com.javacodegeeks.snippets.enterprise.dao.impl.JDBCEmployeeDAOImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <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="root" />
    </bean>
</beans>
```

After loading the new bean to

`App.java`

class we can call its methods, as shown below:

App.java

```
package com.javacodegeeks.snippets.enterprise;

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

import com.javacodegeeks.snippets.enterprise.dao.EmployeeDAO;
import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class App {

    public static void main(String[] args) {

            ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.

            JDBCEmployeeDAO jdbcEmployeeDAO = (JDBCEmployeeDAO) context.getBean("jdbcEmployeeDAO");
            Employee employee3 = new Employee(456, "javacodegeeks", 34);
            jdbcEmployeeDAO.insert(employee3);

            Employee employee4 = jdbcEmployeeDAO.findById(456);
            System.out.println(employee4);
            context.close();
    }
}
```

The output is the one below:

Output

```
Employee [id=456, name=javacodegeeks, age=34]
```

5.1.2 Select total rows

Now, we can query for total number of rows in the database. Again, there are two ways to map the

`Result`

. The easiest way is to use the

`BeanPropertyRowMapper`

, as shown in the example above, but another way is to create our own mapping. We add a new method to query in

`JDBCEmployeeDAOImpl.java`

class. The new method is

`List<Employee> findAll()`

, and it uses the

`queryForInt(String sql)`

method of JdbcTemplate class execute the query, as shown below:

JDBCEmployeeDAO.java

```
01  package com.javacodegeeks.snippets.enterprise.dao;
02
03  import java.util.List;
04
05  import com.javacodegeeks.snippets.enterprise.Employee;
06
07  public interface JDBCEmployeeDAO {
08
09      public void insert(Employee employee);
10      public Employee findById(int id);
11      public List<Employee> findAll();
12  }
```

JDBCEmployeeDAOImpl.java

```
01  package com.javacodegeeks.snippets.enterprise.dao.impl;
02
03  import java.util.ArrayList;
04  import java.util.List;
05  import java.util.Map;
06
07  import javax.sql.DataSource;
08
09  import org.springframework.jdbc.core.BeanPropertyRowMapper;
10  import org.springframework.jdbc.core.JdbcTemplate;
11
12  import com.javacodegeeks.snippets.enterprise.Employee;
13  import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;
14
15  public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
16      private DataSource dataSource;
17      private JdbcTemplate jdbcTemplate;
18
19      public void setDataSource(DataSource dataSource) {
20          this.dataSource = dataSource;
21      }
22
23      public void insert(Employee employee){
24
25          String sql = "INSERT INTO EMPLOYEE " +
26              "(ID, NAME, AGE) VALUES (?, ?, ?)";
27
28          jdbcTemplate = new JdbcTemplate(dataSource);
29
30          jdbcTemplate.update(sql, new Object[] { employee.getId(),
31                  employee.getName(), employee.getAge()
32          });
```

```
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public Employee findById(int id){

        String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";

        jdbcTemplate = new JdbcTemplate(dataSource);
        Employee employee = (Employee) jdbcTemplate.queryForObject(
                sql, new Object[] { id }, new BeanPropertyRowMapper(Employee.class));

        return employee;
    }

    @SuppressWarnings("rawtypes")
    public List<Employee> findAll(){

        jdbcTemplate = new JdbcTemplate(dataSource);
        String sql = "SELECT * FROM EMPLOYEE";

        List<Employee> employees = new ArrayList<Employee>();

        List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
        for (Map row : rows) {
            Employee employee = new Employee();
            employee.setId(Integer.parseInt(String.valueOf(row.get("ID"))));
            employee.setName((String)row.get("NAME"));
            employee.setAge(Integer.parseInt(String.valueOf(row.get("AGE"))));
            employees.add(employee);
        }

        return employees;
    }
}
```

5.1.3 Select a single column

To get a specified column name we create a new method,

```
String findNameById(int id)
```

, where we use the

```
queryForObject(String sql, Object[] args, Class<String> requiredType)
```

method of

```
JdbcTemplate
```

class. In this method we can set the type of the column that the query will return.

JDBCEmployeeDAO.java

```
package com.javacodegeeks.snippets.enterprise.dao;

import java.util.List;

import com.javacodegeeks.snippets.enterprise.Employee;

public interface JDBCEmployeeDAO {

    public void insert(Employee employee);
```

```
    public Employee findById(int id);
    public List<Employee> findAll();
    public String findNameById(int id);
}
```

JDBCEmployeeDAOImpl.java

```
package com.javacodegeeks.snippets.enterprise.dao.impl;

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

import javax.sql.DataSource;

import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import com.javacodegeeks.snippets.enterprise.Employee;
import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;

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

    public void insert(Employee employee){

        String sql = "INSERT INTO EMPLOYEE " +
            "(ID, NAME, AGE) VALUES (?, ?, ?)";

        jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new Object[] { employee.getId(),
                employee.getName(), employee.getAge()
        });
    }

    @SuppressWarnings({ "unchecked", "rawtypes" })
    public Employee findById(int id){

        String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";

        jdbcTemplate = new JdbcTemplate(dataSource);
        Employee employee = (Employee) jdbcTemplate.queryForObject(
                sql, new Object[] { id }, new BeanPropertyRowMapper(Employee.class));

        return employee;
    }

    @SuppressWarnings("rawtypes")
    public List<Employee> findAll(){

        jdbcTemplate = new JdbcTemplate(dataSource);
        String sql = "SELECT * FROM EMPLOYEE";

        List<Employee> employees = new ArrayList<Employee>();

        List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
        for (Map row : rows) {
            Employee employee = new Employee();
            employee.setId(Integer.parseInt(String.valueOf(row.get("ID"))));
            employee.setName((String)row.get("NAME"));
```

```
            employee.setAge(Integer.parseInt(String.valueOf(row.get("AGE"))));
            employees.add(employee);
        }

        return employees;
    }

    public String findNameById(int id){

        String sql = "SELECT NAME FROM EMPLOYEE WHERE ID = ?";

        String name = (String)jdbcTemplate.queryForObject(
                sql, new Object[] { id }, String.class);

        return name;
    }

}
```

We use the new queries in

`App.class`

as shown below:

App.java

```
package com.javacodegeeks.snippets.enterprise;

import java.util.List;

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

import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class App {

    public static void main(String[] args) {

            ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.

            JDBCEmployeeDAO jdbcEmployeeDAO = (JDBCEmployeeDAO) context.getBean("jdbcEmployeeDAO");

            List<Employee> employees = jdbcEmployeeDAO.findAll();
            System.out.println(employees);

            String name = jdbcEmployeeDAO.findNameById(456);
            System.out.println(name);

            context.close();
    }
}
```

In the result below we first see the list of Employees from the

`findAll()`

method and then the value of the name column from the

`findNameById(int id)`

method.

Output

```
[Employee [id=123, name=javacodegeeks, age=30], Employee [id=456, name=javacodegeeks, age=34]]
javacodegeeks
```

5.2 BatchUpdate example

The

`batchUpdate()`

method of

`JdbcTemplate`

class can be used to perform all batch inserts to the database. Below there are two implementations of a

`batchUpdate()`

to the database.

The first one,

`insertBatch1(final List<Employee> employees)`

uses the

`BatchPreparedStatementSetter`

to insert a list of Objects to the database. The

`BatchPreparedStatementSetter`

is passed as the second parameter in the

`batchUpdate()`

method. It provides two methods that can be overriden. The

`getBatchSize()`

method provides the size of the current batch, whereas the

`setValues(PreparedStatement ps, int i)`

method is used to set the values for the parameters of the prepared statement.

The second method

`insertBatch2(final String sql)`

calls the

`batchUpdate()`

method of

`JDBCTemplate`

class to execute an sql statement.

JDBCEmployeeDAO.java

```
01 package com.javacodegeeks.snippets.enterprise.dao;
02
03 import java.util.List;
04
05 import com.javacodegeeks.snippets.enterprise.Employee;
06
07 public interface JDBCEmployeeDAO {
08
09     public void insert(Employee employee);
10     public Employee findById(int id);
11     public List<Employee> findAll();
12     public String findNameById(int id);
13     public void insertBatch1(final List<Employee> employees);
14     public void insertBatch2(final String sql);
15 }
```

JDBCEmployeeDAO.java

```
001 package com.javacodegeeks.snippets.enterprise.dao.impl;
002
003 import java.sql.PreparedStatement;
004 import java.sql.SQLException;
005 import java.util.ArrayList;
006 import java.util.List;
007 import java.util.Map;
008
009 import javax.sql.DataSource;
010
011 import org.springframework.jdbc.core.BatchPreparedStatementSetter;
012 import org.springframework.jdbc.core.BeanPropertyRowMapper;
013 import org.springframework.jdbc.core.JdbcTemplate;
014
015 import com.javacodegeeks.snippets.enterprise.Employee;
016 import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;
017
018 public class JDBCEmployeeDAOImpl implements JDBCEmployeeDAO{
019     private DataSource dataSource;
020     private JdbcTemplate jdbcTemplate;
021
022     public void setDataSource(DataSource dataSource) {
023         this.dataSource = dataSource;
024     }
025
026     public void insert(Employee employee){
027
028         String sql = "INSERT INTO EMPLOYEE " +
029             "(ID, NAME, AGE) VALUES (?, ?, ?)";
030
031         jdbcTemplate = new JdbcTemplate(dataSource);
032
033         jdbcTemplate.update(sql, new Object[] { employee.getId(),
034                 employee.getName(), employee.getAge()
```

```
035         });
036     }
037
038     @SuppressWarnings({ "unchecked", "rawtypes" })
039     public Employee findById(int id){
040
041         String sql = "SELECT * FROM EMPLOYEE WHERE ID = ?";
042
043         jdbcTemplate = new JdbcTemplate(dataSource);
044         Employee employee = (Employee) jdbcTemplate.queryForObject(
045                 sql, new Object[] { id }, new BeanPropertyRowMapper(Employee.class));
046
047         return employee;
048     }
049
050     @SuppressWarnings("rawtypes")
051     public List<Employee> findAll(){
052
053         jdbcTemplate = new JdbcTemplate(dataSource);
054         String sql = "SELECT * FROM EMPLOYEE";
055
056         List<Employee> employees = new ArrayList<Employee>();
057
058         List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
059         for (Map row : rows) {
060             Employee employee = new Employee();
061             employee.setId(Integer.parseInt(String.valueOf(row.get("ID"))));
062             employee.setName((String)row.get("NAME"));
063             employee.setAge(Integer.parseInt(String.valueOf(row.get("AGE"))));
064             employees.add(employee);
065         }
066
067         return employees;
068     }
069
070     public String findNameById(int id){
071
072         jdbcTemplate = new JdbcTemplate(dataSource);
073         String sql = "SELECT NAME FROM EMPLOYEE WHERE ID = ?";
074
075         String name = (String)jdbcTemplate.queryForObject(
076                 sql, new Object[] { id }, String.class);
077
078         return name;
079     }
080
081     public void insertBatchSQL(final String sql){
082
083         jdbcTemplate.batchUpdate(new String[]{sql});
084
085     }
086
087     public void insertBatch1(final List<Employee> employees){
088
089         jdbcTemplate = new JdbcTemplate(dataSource);
090           String sql = "INSERT INTO EMPLOYEE " +
091             "(ID, NAME, AGE) VALUES (?, ?, ?)";
092
093           jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
094
095             public void setValues(PreparedStatement ps, int i) throws SQLException {
096                 Employee employee = employees.get(i);
097                 ps.setLong(1, employee.getId());
098                 ps.setString(2, employee.getName());
099                 ps.setInt(3, employee.getAge() );
100             }
101
```

```
            public int getBatchSize() {
                return employees.size();
            }
          });
        }

    public void insertBatch2(final String sql){
        jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.batchUpdate(new String[]{sql});

    }

}
```

Let's run

`App.java`

class again. We call the two new methods to insert two new rows to the Employee table and then update the table setting all values of a a column to a specified value.

App.java

```
package com.javacodegeeks.snippets.enterprise;

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

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

import com.javacodegeeks.snippets.enterprise.dao.JDBCEmployeeDAO;

public class App {

    public static void main(String[] args) {

            ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.

            JDBCEmployeeDAO jdbcEmployeeDAO = (JDBCEmployeeDAO) context.getBean("jdbcEmployeeDAO");

            Employee emplNew1 = new Employee(23, "John", 23);
            Employee emplNew2 = new Employee(223, "Mark", 43);
            List<Employee> employeesN = new ArrayList();
            employeesN.add(emplNew1);
            employeesN.add(emplNew2);
            jdbcEmployeeDAO.insertBatch1(employeesN);
            System.out.println(" inserted rows: " + employeesN);

            System.out.println(" FindAll : " + jdbcEmployeeDAO.findAll());
            jdbcEmployeeDAO.insertBatch2("UPDATE EMPLOYEE SET NAME ='Mary'");

            List<Employee> employees = jdbcEmployeeDAO.findAll();
            System.out.println("Updated column name of table: " + employees);

            System.out.println(" FindAll : " + jdbcEmployeeDAO.findAll());
            context.close();
    }
}
```

The result is shown below:

Output

```
inserted rows: [Employee [id=23, name=John, age=23], Employee [id=223, name=Mark, age=43]]
 FindAll : [Employee [id=23, name=John, age=23], Employee [id=223, name=Mark, age=43]]
Updated column name of table: [Employee [id=23, name=Mary, age=23], Employee [id=223, name=Mary, age=43]]
 FindAll : [Employee [id=23, name=Mary, age=23], Employee [id=223, name=Mary, age=43]]
```

This was an example of Spring JDBC integration and JdbcTemplate class.
Download the Eclipse project of the Spring JdbcTemplate Example: SpringJdbcTemplateExample.zip