

Spring - JDBC Framework Overview

While working with the 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, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

So what you have to do is just define the 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. I'm going to take classic and the most popular approach which makes use of **JdbcTemplate** class of the framework. This is the central framework class that manages all the database communication and exception handling.

JdbcTemplate Class

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts 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 *JdbcTemplate* class are *threadsafe* once configured. So you can configure a single instance of a *JdbcTemplate* 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, and the JdbcTemplate is created in the setter for the DataSource.

Configuring Data Source

Let us create a database table **Student** in our database **TEST**. We 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 as shown in the following code snippet –

```
<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 = "password"/>  
</bean>
```

Data Access Object (DAO)

DAO stands for Data Access Object, 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 DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

Querying for an integer

```
String SQL = "select count(*) from Student";  
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Querying for a long

```
String SQL = "select count(*) from Student";  
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable

```
String SQL = "select age from Student where id = ?";  
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String

```
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class
```

Querying and returning an object

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(
    SQL, new Object[]{10}, new StudentMapper());

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;
    }
}
```

Querying and returning multiple objects

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

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;
    }
}
```

Inserting a row into the table

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table

```
String SQL = "update Student set name = ? where id = ?";  
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deleting a row from the table

```
String SQL = "delete Student where id = ?";  
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

Executing DDL Statements

You can use the **execute(..)** method from *JdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table –

```
String SQL = "CREATE TABLE Student( " +  
    "ID    INT NOT NULL AUTO_INCREMENT, " +  
    "NAME VARCHAR(20) NOT NULL, " +  
    "AGE  INT NOT NULL, " +  
    "PRIMARY KEY (ID));"  
  
jdbcTemplateObject.execute( SQL );
```

Spring JDBC Framework Examples

Based on the above concepts, let us check few important examples which will help you in understanding usage of JDBC framework in Spring –

Sr.No.	Example & Description
1	Spring JDBC Example This example will explain how to write a simple JDBC-based Spring application.
2	SQL Stored Procedure in Spring Learn how to call SQL stored procedure while using JDBC in Spring.

Useful Video Courses



Spring MVC CRUD Example

CRUD (Create, Read, Update and Delete) application is the most important application for creating any project. It provides an idea to develop a large project. In spring MVC, we can develop a simple CRUD application.

Here, we are using **JdbcTemplate** for database interaction.

Create a table

Here, we are using emp99 table present in the MySQL database. It has 4 fields: id, name, salary, and designation. Here, id is auto incremented which is generated by the sequence.

Column Name	Data Type	Nullable	Default	Primary Key
ID	NUMBER	No	-	1
NAME	VARCHAR2(4000)	Yes	-	-
SALARY	NUMBER	Yes	-	-
DESIGNATION	VARCHAR2(4000)	Yes	-	-
				1 - 4

Spring MVC CRUD Example

1. Add dependencies to pom.xml file.

pom.xml



```

<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jasper</artifactId>
  <version>9.0.12</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
  <groupId>javax.servlet</groupId>

```

Next
Stay

```
<artifactId> servlet-api </artifactId>
<version> 3.0-alpha-1 </version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
  <groupId> javax.servlet </groupId>
  <artifactId> jstl </artifactId>
  <version> 1.2 </version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId> mysql </groupId>
  <artifactId> mysql-connector-java </artifactId>
  <version> 8.0.11 </version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
  <groupId> org.springframework </groupId>
  <artifactId> spring-jdbc </artifactId>
  <version> 5.1.1.RELEASE </version>
</dependency>
```

2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the fields exist in the database.

Emp.java

```
package com.javatpoint.beans;

public class Emp {
  private int id;
  private String name;
  private float salary;
  private String designation;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public float getSalary() {
    return salary;
  }
```

Next
Stay

```

}
public void setSalary(float salary) {
    this.salary = salary;
}
public String getDesignation() {
    return designation;
}
public void setDesignation(String designation) {
    this.designation = designation;
}
}
}

```

3. Create the controller class

EmpController.java

```

package com.javatpoint.controllers;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.javatpoint.beans.Emp;
import com.javatpoint.dao.EmpDao;

@Controller
public class EmpController {
    @Autowired
    EmpDao dao;//will inject dao from XML file

    /*It displays a form to input data, here "command" is a reserved request attribute
    *which is used to display object data into form
    */
    @RequestMapping("/empform")
    public String showform(Model m){
        m.addAttribute("command", new Emp());
        return "empform";
    }

    /*It saves object into database. The @ModelAttribute puts request data
    * into model object. You need to mention RequestMethod.POST method
    * because default request is GET*/
    @RequestMapping(value="/save",method = RequestMethod.POST)
    public String save(@ModelAttribute("emp") Emp emp){
        dao.save(emp);
        return "redirect:/viewemp";//will redirect to viewemp request mapping
    }
}

```



```

/* It provides list of employees in model object */
@RequestMapping("/viewemp")
public String viewemp(Model m){
    List<Emp> list=dao.getEmployees();
    m.addAttribute("list",list);
    return "viewemp";
}

/* It displays object data into form for the given id.
 * The @PathVariable puts URL data into variable.*/
@RequestMapping(value="/editemp/{id}")
public String edit(@PathVariable int id, Model m){
    Emp emp=dao.getEmpById(id);
    m.addAttribute("command",emp);
    return "empeditform";
}

/* It updates model object. */
@RequestMapping(value="/editsave",method = RequestMethod.POST)
public String editsave(@ModelAttribute("emp") Emp emp){
    dao.update(emp);
    return "redirect:/viewemp";
}

/* It deletes record for the given id in URL and redirects to /viewemp */
@RequestMapping(value="/deleteemp/{id}",method = RequestMethod.GET)
public String delete(@PathVariable int id){
    dao.delete(id);
    return "redirect:/viewemp";
}
}

```

4. Create the DAO class

Let's create a DAO class to access the required data from the database.

EmpDao.java

```

package com.javatpoint.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.javatpoint.beans.Emp;

public class EmpDao {
    JdbcTemplate template;

    public void setTemplate(JdbcTemplate template) {
        this.template = template;
    }
}

```




```

public int save(Emp p){
    String sql="insert into Emp99(name,salary,designation) values('"+p.getName()+"','"+p.getSalary()+"','"+p.getDesignation()+"')";
    return template.update(sql);
}

public int update(Emp p){
    String sql="update Emp99 set name='"+p.getName()+"', salary='"+p.getSalary()+"',designation='"+p.getDesignation()+"' where id='"+p.getId()+";
    return template.update(sql);
}

public int delete(int id){
    String sql="delete from Emp99 where id="+id+"";
    return template.update(sql);
}

public Emp getEmpById(int id){
    String sql="select * from Emp99 where id=?";
    return template.queryForObject(sql, new Object[]{id},new BeanPropertyRowMapper<Emp>(Emp.class));
}

public List<Emp> getEmployees(){
    return template.query("select * from Emp99",new RowMapper<Emp>(){
        public Emp mapRow(ResultSet rs, int row) throws SQLException {
            Emp e=new Emp();
            e.setId(rs.getInt(1));
            e.setName(rs.getString(2));
            e.setSalary(rs.getFloat(3));
            e.setDesignation(rs.getString(4));
            return e;
        }
    });
}
}

```

5. Provide the entry of controller in the web.xml file

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation='
//java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd' id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```



6. Define the bean in the xml file

spring-servlet.xml

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

    <context:component-scan base-package="com.javatpoint.controllers"></context:component-scan>

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

    <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/test"></property>
        <property name="username" value=""></property>
        <property name="password" value=""></property>
    </bean>

    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="ds"></property>
    </bean>

    <bean id="dao" class="com.javatpoint.dao.EmpDao">
        <property name="template" ref="jt"></property>
    </bean>
</beans>
```

7. Create the requested page

index.jsp



```
<a href="empform">Add Employee</a>
<a href="viewemp">View Employees</a>
```

8. Create the other view components

empform.jsp

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Add New Employee</h1>
<form:form method="post" action="save">
  <table >
    <tr>
      <td>Name : </td>
      <td><form:input path="name" /></td>
    </tr>
    <tr>
      <td>Salary :</td>
      <td><form:input path="salary" /></td>
    </tr>
    <tr>
      <td>Designation :</td>
      <td><form:input path="designation" /></td>
    </tr>
    <tr>
      <td></td>
      <td><input type="submit" value="Save" /></td>
    </tr>
  </table>
</form:form>
```

empeditform.jsp

Here "/SpringMVCCRUDSimple" is the project name, change this if you have different project name

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Edit Employee</h1>
```

Next
Stay

```

<form:form method="POST" action="/SpringMVCCRUDSimple/editsave">
  <table >
    <tr>
      <td></td>
      <td><form:hidden path="id" /></td>
    </tr>
    <tr>
      <td>Name : </td>
      <td><form:input path="name" /></td>
    </tr>
    <tr>
      <td>Salary :</td>
      <td><form:input path="salary" /></td>
    </tr>
    <tr>
      <td>Designation :</td>
      <td><form:input path="designation" /></td>
    </tr>
    <tr>
      <td></td>
      <td><input type="submit" value="Edit Save" /></td>
    </tr>
  </table>
</form:form>

```

viewemp.jsp

```

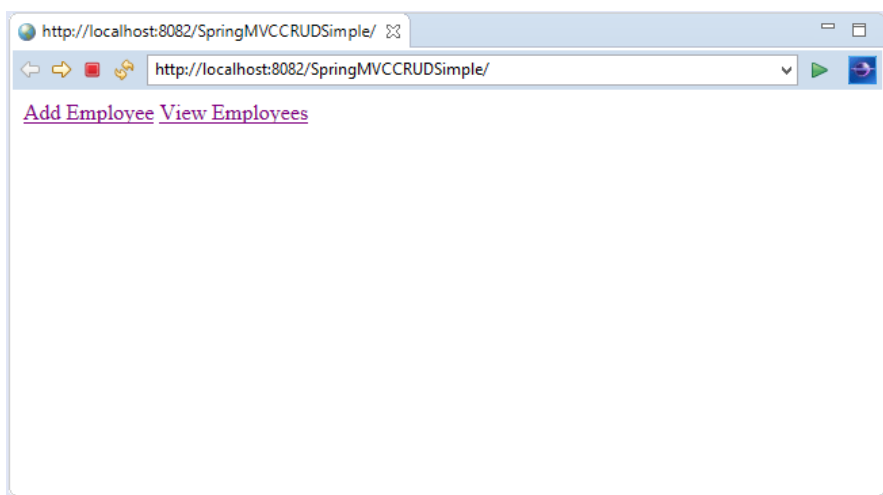
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Employees List</h1>
<table border="2" width="70%" cellpadding="2">
<tr><th>Id</th><th>Name</th><th>Salary</th><th>Designation</th><th>Edit</th><th>Delete</th></tr>
<c:forEach var="emp" items="${list}">
  <tr>
    <td>${emp.id}</td>
    <td>${emp.name}</td>
    <td>${emp.salary}</td>
    <td>${emp.designation}</td>
    <td><a href="editemp/${emp.id}">Edit</a></td>
    <td><a href="deleteemp/${emp.id}">Delete</a></td>
  </tr>
</c:forEach>
</table>
<br/>
<a href="empform">Add New Employee</a>

```

Output:



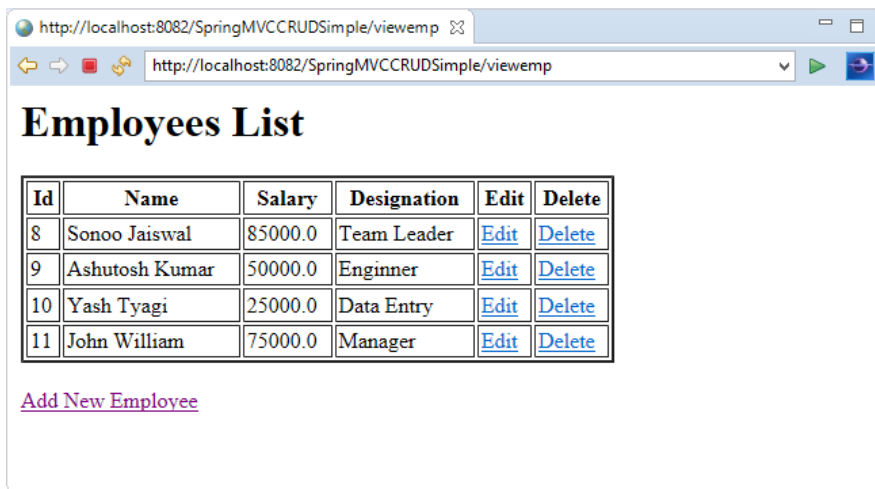


On clicking **Add Employee**, you will see the following form.

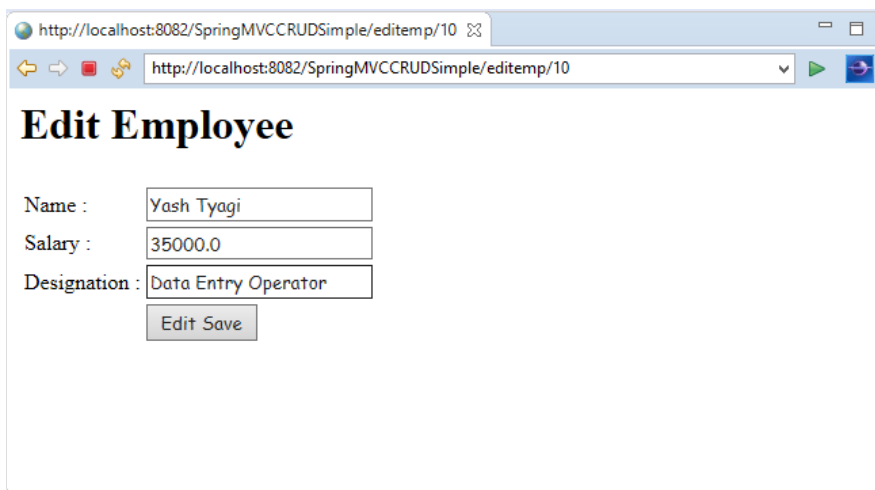
A screenshot of a web browser window showing the 'Add New Employee' form. The address bar shows the URL 'http://localhost:8082/SpringMVCCRUDSimple/empform'. The form has a title 'Add New Employee' in bold. Below the title, there are three input fields: 'Name' with the value 'Sonoo Jaiswal', 'Salary' with the value '85000', and 'Designation' with the value 'Team Leader'. Below these fields is a 'Save' button.

Fill the form and **click Save** to add the entry into the database.

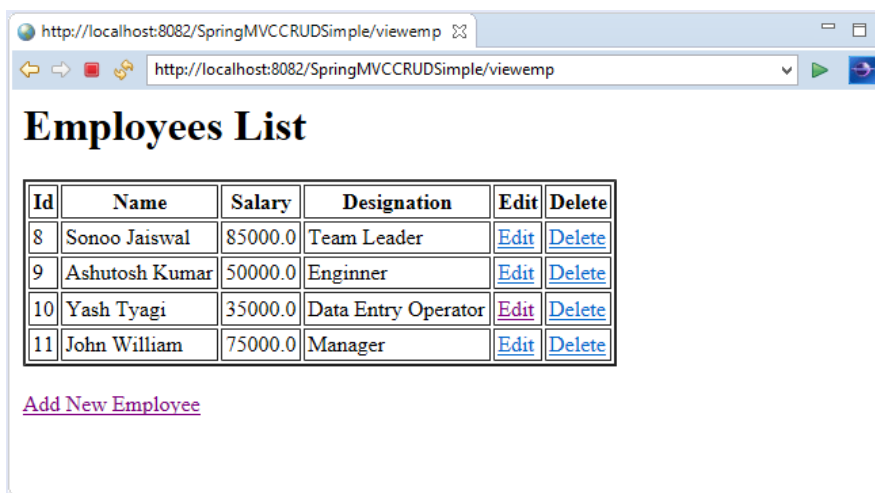




Now, click **Edit** to make some changes in the provided data.

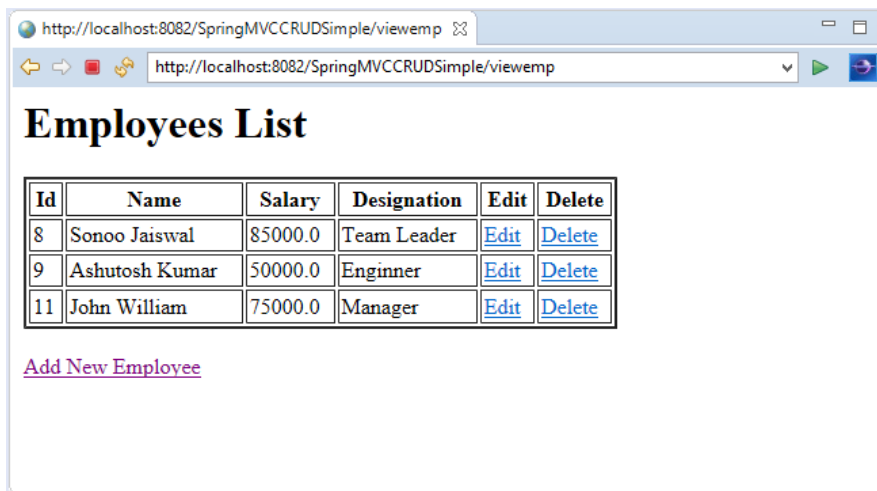


Now, click **Edit Save** to add the entry with changes into the database.



Now, click **Delete** to delete the entry from the database.





Id	Name	Salary	Designation	Edit	Delete
8	Sonoo Jaiswal	85000.0	Team Leader	Edit	Delete
9	Ashutosh Kumar	50000.0	Enginner	Edit	Delete
11	John William	75000.0	Manager	Edit	Delete

[Add New Employee](#)

Download this example (developed using Eclipse)

Download SQL File

Download SQL File

Download MYSQL-connector.jar file

If you are not using maven, download MYSQL-connector.jar.

← Prev



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Next
Stay