Learn Spring framework:

- Understand the core of Spring
- Understand Spring MVC
- Understand Spring AOP
- Understand Spring Data JPA
- Spring Dependency Injection (XML)
- Spring Dependency Injection (Annotations)

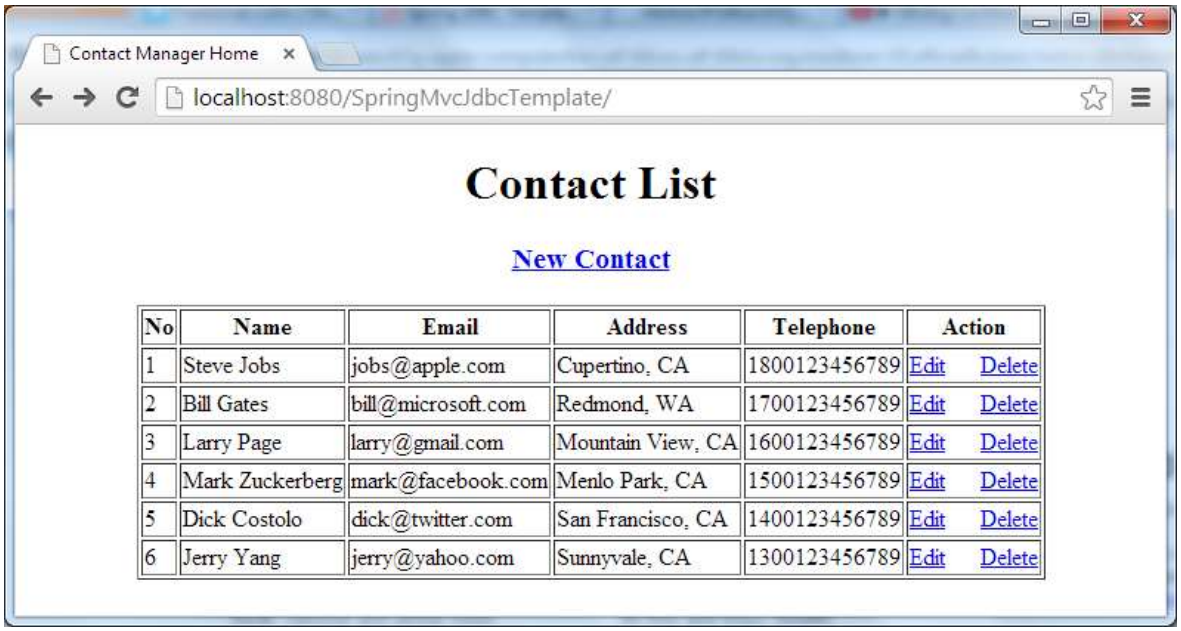
config)

- Spring & Struts Integration (XML)
- Spring & Struts Integration (Java config)
- 14 Tips for Writing Spring MVC Controller

Java Spring MVC with JdbcTemplate Tutorial

Written by Nam Ha Minh
Last Updated on 02 September 2019 | Print Email

Spring makes it easy to work with JDBC through the use of **`JdbcTemplate`** and related classes in the `org.springframework.jdbc.core` and related packages. For an introductory tutorial for the basics of **`JdbcTemplate`**, see: Spring JDBC Template Simple Example. This tutorial goes further by demonstrating how to integrate **`JdbcTemplate`** in a Spring MVC application. The sample application in this tutorial manages a contact list that looks like this:



The sample application is developed using the following pieces of software/technologies (of course you can use newer versions):

- Java 7
- Eclipse Kepler
- Spring framework 4.0
- JSTL 1.2
- MySQL Database 5.5

Maven 3

1. Creating MySQL database

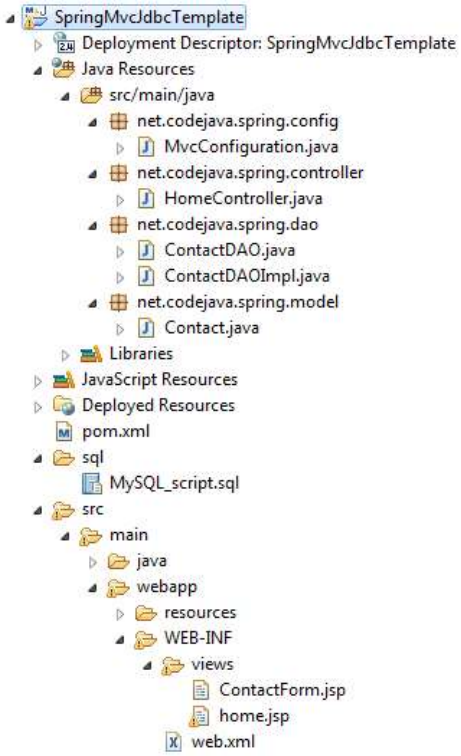
Execute the following MySQL script to create a database named **contactdb** and a table named **contact**:

```
create database contactdb;

CREATE TABLE `contact` (
  `contact_id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL,
  `address` varchar(45) NOT NULL,
  `telephone` varchar(45) NOT NULL,
  PRIMARY KEY (`contact_id`)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8
```

2. Creating Maven Project in Eclipse

It's recommended to use **spring-mvc-archetype** to create the project (See: Creating a Spring MVC project using Maven and Eclipse in one minute). Here's the project's final structure:



The following XML section in `pom.xml`file is for adding dependencies configuration to the project:

```
<properties>
    <java.version>1.7</java.version>
    <spring.version>4.0.3.RELEASE</spring.version>
    <cglib.version>2.2.2</cglib.version>
</properties>

<dependencies>
    <!-- Spring core & mvc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>${spring.version}</version>
        <type>jar</type>
        <scope>compile</scope>
    </dependency>

    <!-- CGLib for @Configuration -->
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib-nodep</artifactId>
        <version>${cglib.version}</version>
        <scope>runtime</scope>
    </dependency>


    <!-- Servlet Spec -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>

</dependencies>
```

3. Coding Model Class

The model class - **`Contact.java`** - is pretty simple:

```
package net.codejava.spring.model;

public class Contact {
    private int id;
    private String name;
    private String email;
    private String address;
    private String telephone;

    public Contact() {
    }

    public Contact(String name, String email, String address, String telephor
        this.name = name;
        this.email = email;
        this.address = address;
        this.telephone = telephone;
    }

    // getters and setters
}
```

This class simply maps a row in the table **contact** to a plain old Java object (POJO) - `Contact`.

4. Coding DAO Classes

The **`ContactDAO`** interface defines methods for performing CRUD operations on the **contact** table:

```
package net.codejava.spring.dao;

import java.util.List;

import net.codejava.spring.model.Contact;

/**
 * Defines DAO operations for the contact model.
 * @author www.codejava.net
 *
 */
public interface ContactDAO {

    public void saveOrUpdate(Contact contact);

    public void delete(int contactId);

    public Contact get(int contactId);

    public List<Contact> list();
}
```

And here is an implementation - **`ContactDAOImpl.java`**:

```
package net.codejava.spring.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import net.codejava.spring.model.Contact;

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

/**
 * An implementation of the ContactDAO interface.
 * @author www.codejava.net
 *
 */
public class ContactDAOImpl implements ContactDAO {

    private JdbcTemplate jdbcTemplate;

    public ContactDAOImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Override
    public void saveOrUpdate(Contact contact) {
        // implementation details goes here...
    }

    @Override
    public void delete(int contactId) {
        // implementation details goes here...
    }

    @Override
    public List<Contact> list() {
        // implementation details goes here...
    }

    @Override
    public Contact get(int contactId) {
        // implementation details goes here...
    }

}
```

Pay attention to the beginning section that declares a **`JdbcTemplate`** and a **`DataSource`** object is injected via the constructor:

```
private JdbcTemplate jdbcTemplate;

public ContactDAOImpl(DataSource dataSource) {
    jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Now, let's look at implementation details of each method.

Insert or update a new contact:

```
public void saveOrUpdate(Contact contact) {
    if (contact.getId() > 0) {
        // update
        String sql = "UPDATE contact SET name=?, email=?, address=?, "
                    + "telephone=? WHERE contact_id=?";
        jdbcTemplate.update(sql, contact.getName(), contact.getEmail(),
                contact.getAddress(), contact.getTelephone(), contact.getId(
    } else {
        // insert
        String sql = "INSERT INTO contact (name, email, address, telephone)"
                    + " VALUES (?, ?, ?, ?)";
        jdbcTemplate.update(sql, contact.getName(), contact.getEmail(),
                contact.getAddress(), contact.getTelephone());
    }

}
```

Note that if the contact object having ID greater than zero, update it; otherwise that is an insert.

NOTE: For more code readability, you can use NamedParameterJdbcTemplate, instead of using question marks (?) as placeholders. Also you can use SimpleJdbcInsert class which is more convenient to use.

Delete a contact:

```
public void delete(int contactId) {
    String sql = "DELETE FROM contact WHERE contact_id=?";
    jdbcTemplate.update(sql, contactId);
}
```

List all contact:

```
public List<Contact> list() {
    String sql = "SELECT * FROM contact";
    List<Contact> listContact = jdbcTemplate.query(sql, new RowMapper<Contact

        @Override
        public Contact mapRow(ResultSet rs, int rowNum) throws SQLException 
            Contact aContact = new Contact();

            aContact.setId(rs.getInt("contact_id"));
            aContact.setName(rs.getString("name"));
            aContact.setEmail(rs.getString("email"));
            aContact.setAddress(rs.getString("address"));
            aContact.setTelephone(rs.getString("telephone"));

            return aContact;
        }

    });

    return listContact;
}
```

e the use of `RowMapper` to map a row in the result set to a POJO object. For more enient, you can use the `BeanPropertyRowMapper` class like this:

```
public List<Contact> list() {
    String sql = "SELECT * FROM Contact";

    return jdbcTemplate.query(sql, BeanPropertyRowMapper.newInstance(Contact.
}
```

Make sure that the Contact class declare the field names exactly match the column names in the database table. Using `BeanPropertyRowMapper` is more convenient, but performance is slower than using `RowMapper`.

Get a particular contact:

```
public Contact get(int contactId) {
    String sql = "SELECT * FROM contact WHERE contact_id=" + contactId;
    return jdbcTemplate.query(sql, new ResultSetExtractor<Contact>() {

        @Override
        public Contact extractData(ResultSet rs) throws SQLException,
                DataAccessException {
            if (rs.next()) {
                Contact contact = new Contact();
                contact.setId(rs.getInt("contact_id"));
                contact.setName(rs.getString("name"));
                contact.setEmail(rs.getString("email"));
                contact.setAddress(rs.getString("address"));
                contact.setTelephone(rs.getString("telephone"));
                return contact;
            }

            return null;
        }

    });
}
```

Notice the use of `ResultSetExtractor` to extract a single row as a POJO. You can also use the `BeanPropertyRowMapper` class like this:

```
public Contact get(int contactId) {
    String sql = "SELECT * FROM Contact WHERE id=" + contactId;
    return jdbcTemplate.queryForObject(sql, BeanPropertyRowMapper.newInstance(
}
```

It greatly simplifies the code, but in return of slower performance.

5. Coding Spring MVC Configuration

Java-based classes and annotations are used to configure this Spring MVC application. Here's code of the **`MvcConfiguration`** class:

```
package net.codejava.spring.config;

import javax.sql.DataSource;

import net.codejava.spring.dao.ContactDAO;
import net.codejava.spring.dao.ContactDAOImpl;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegis
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdap
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan(basePackages="net.codejava.spring")
@EnableWebMvc
public class MvcConfiguration extends WebMvcConfigurerAdapter{

    @Bean
    public ViewResolver getViewResolver(){
        InternalResourceViewResolver resolver = new InternalResourceViewResol
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/r
    }

    @Bean
    public DataSource getDataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/contactdb");
        dataSource.setUsername("root");
        dataSource.setPassword("P@ssw0rd");

        return dataSource;
    }

    @Bean
    public ContactDAO getContactDAO() {
        return new ContactDAOImpl(getDataSource());
    }
}
```

Notice the `getDataSource()` method returns a configured `DataSource` bean. You may have to change the database URL, username and password according to your environments.

The `getContactDAO()` method returns an implementation of the `ContactDAO` interface, which is the `ContactDAOImpl` class. This bean will be injected to the controller class, which is described below.

6. Configuring Spring MVC Dispatcher Servlet

To enable Spring MVC for our Java web application, update the web deployment descriptor file (web.xml) as below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/x
    <display-name>SpringMvcJdbcTemplate</display-name>
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.web.context.support.AnnotationConfigWebApplic
        </param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener>

    <servlet>
        <servlet-name>SpringDispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</ser
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.web.context.support.AnnotationConfigWebAp
            </param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>net.codejava.spring</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringDispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

7. Coding Spring Controller Class

Skeleton of the **`HomeController`** class:

```
public class HomeController {

    @Autowired
    private ContactDAO contactDAO;

    // handler methods go here...
}
```

Notice we use the **`@Autowired`** annotation to let Spring inject an instance of the `ContactDAO` implementation into this controller automatically. Each handler method uses this `contactDAO` object to perform necessary CRUD operations. Let's see implementation details of each method.

dler method for listing all contacts (also served as home page):

```
@RequestMapping(value="/")
public ModelAndView listContact(ModelAndView model) throws IOException{
    List<Contact> listContact = contactDAO.list();
    model.addObject("listContact", listContact);
    model.setViewName("home");

    return model;
}
```

Handler method for displaying new contact form:

```
@RequestMapping(value = "/newContact", method = RequestMethod.GET)
public ModelAndView newContact(ModelAndView model) {
    Contact newContact = new Contact();
    model.addObject("contact", newContact);
    model.setViewName("ContactForm");
    return model;
}
```

Handler method for inserting/updating a contact:

```
@RequestMapping(value = "/saveContact", method = RequestMethod.POST)
public ModelAndView saveContact(@ModelAttribute Contact contact) {
    contactDAO.saveOrUpdate(contact);
    return new ModelAndView("redirect:/");
}
```

Handler method for deleting a contact:

```
@RequestMapping(value = "/deleteContact", method = RequestMethod.GET)
public ModelAndView deleteContact(HttpServletRequest request) {
    int contactId = Integer.parseInt(request.getParameter("id"));
    contactDAO.delete(contactId);
    return new ModelAndView("redirect:/");
}
```

Handler method for retrieving details of a particular contact for editing:

```
@RequestMapping(value = "/editContact", method = RequestMethod.GET)
public ModelAndView editContact(HttpServletRequest request) {
    int contactId = Integer.parseInt(request.getParameter("id"));
    Contact contact = contactDAO.get(contactId);
    ModelAndView model = new ModelAndView("ContactForm");
    model.addObject("contact", contact);

    return model;
}
```

8. Coding Contact Listing Page (Home Page)

Here's source code of the **home.jsp** page that displays the contact list as well as action links for creating new, editing and deleting a contact.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Contact Manager Home</title>
    </head>
    <body>
        <div align="center">
            <h1>Contact List</h1>
            <h3><a href="/newContact">New Contact</a></h3>
            <table border="1">
                <th>No</th>
                <th>Name</th>
                <th>Email</th>
                <th>Address</th>
                <th>Telephone</th>
                <th>Action</th>

                <c:forEach var="contact" items="${listContact}" varStatus="st
                <tr>
                    <td>${status.index + 1}</td>
                    <td>${contact.name}</td>
                    <td>${contact.email}</td>
                    <td>${contact.address}</td>
                    <td>${contact.telephone}</td>
                    <td>
                        <a href="/editContact?id=${contact.id}">Edit</a>
                            
                        <a href="/deleteContact?id=${contact.id}">Delete</a>
                    </td>

                </tr>
                </c:forEach>
            </table>
        </div>
    </body>
</html>
```

Notice this JSP page uses JSTL and EL expressions.

9. Coding Contact Form Page

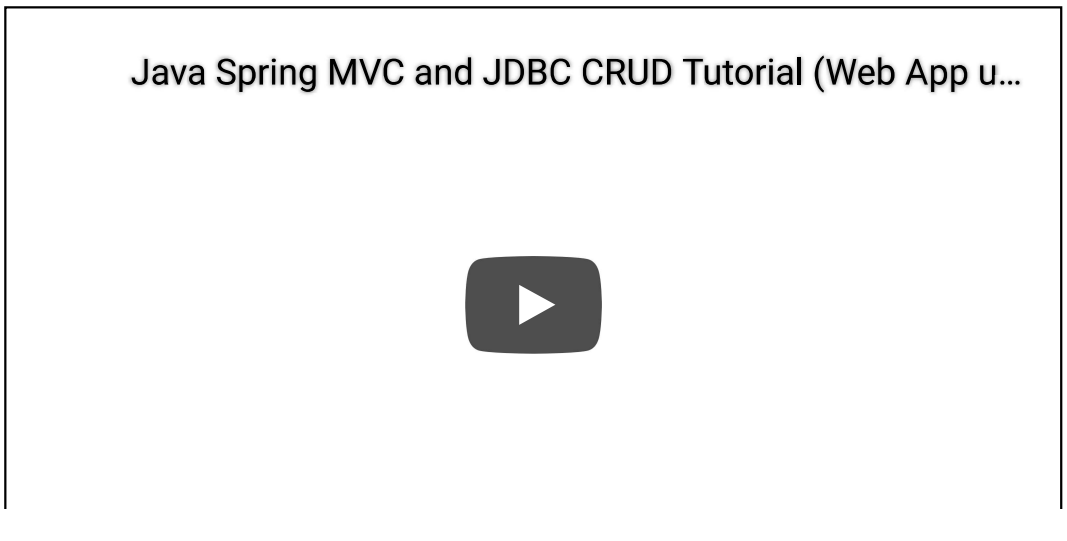
The contact form page (**ContactForm.jsp**) displays details of a contact for creating new or updating old one. Here's its full source code:

```jsp
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>New/Edit Contact</title>
</head>
<body>
    <div align="center">
        <h1>New/Edit Contact</h1>
        <form:form action="saveContact" method="post" modelAttribute="contact
        <table>
            <form:hidden path="id"/>
            <tr>
                <td>Name:</td>
                <td><form:input path="name" /></td>
            </tr>
            <tr>
                <td>Email:</td>
                <td><form:input path="email" /></td>
            </tr>
            <tr>
                <td>Address:</td>
                <td><form:input path="address" /></td>
            </tr>
            <tr>
                <td>Telephone:</td>
                <td><form:input path="telephone" /></td>
            </tr>
            <tr>
                <td colspan="2" align="center"><input type="submit" value="Sa
            </tr>
        </table>
        </form:form>
    </div>
</body>
</html>
```

Notice that this JSP page uses Spring form tags to bind the values of the form to a model object.

To test out the application, you can download the Eclipse project or deploy the attached WAR file at your convenience.

You can also watch the video version of this tutorial below:



Related Spring and Database Tutorials:

- Spring JDBC Template Simple Example
- How to configure Spring MVC JdbcTemplate with JNDI Data Source in Tomcat
- Spring and Hibernate Integration Tutorial (XML Configuration)
- Understand Spring Data JPA with Simple Example
- Spring MVC + Spring Data JPA + Hibernate - CRUD Example

Other Spring Tutorials:

- Understand the core of Spring framework
- Understand Spring MVC
- Understand Spring AOP
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- 14 Tips for Writing Spring MVC Controller
- Spring Web MVC Security Basic Example (XML Configuration)

About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make friend with him on Facebook.

Attachments:

SpringMvcJdbcTemplate.war	**[Deployable WAR file]**	**5521 kB**
SpringMvcJdbcTemplate.zip	**[Eclipse-Maven project]**	**27 kB**

Add comment

Name E-mail

comment

500 symbols left

Notify me of follow-up comments

Comments

#102**vikas** 2019-10-19 05:33
I got this error, how to solve?

HTTP Status 500 - Servlet.init() for servlet SpringDispatcher threw exception

Quote

#101**nagajyothi** 2018-12-28 02:46
i am fresher i want spring project structure using annotations no xml example

Quote

#100**Kompalli Siva** 2018-08-02 14:21
I follwed you .. And got this error.
java.lang.IllegalStateException: Neither BindingResult nor plain target object for bean name 'contact' available as request attribute.

Quote

#99**champ** 2018-07-24 04:54
I just needed to add spring-jdbc and mysql-connector-java into POM file that is required as dependencies. then I took .jar files of them put into a file called 'lib' under the WEB-INF folder. Then I was able to run the app successfully.

Quote

#98**Tanmay** 2018-06-14 06:19
```
@Bean
public EmpDao getEmpDao() {
System.out.println("Loaded");
return new EmpDaoImpl(getDataSource());
}
```

This method is not calling at all . and its giving 404 error

Quote

1 2 3 4 5 6 7 8 9 10 11 »

Refresh comments list