Spring MVC and Hibernate

Agenda

1. Spring MVC - Form Handling
2. Spring MVC Application:With Form Handling
3. Spring – Hibernate Application

Objectives

At the end of this session, you will be able to:

- Understand the role of Spring Form tags in JSP
- Understand How to create a Spring Web Application with form data
- Understand How to persist data using hibernate

Spring MVC Form Handling

Spring MVC Form Handling

- Traditionally when handling from data in Web application we use *HTML form and input tags to accept data from client*
- This data is received by the controller through request object is then *converted into bean / model class*
- With Spring MVC the *above 2 steps are encapsulated*
- Spring MVC has a separate tag library to support form data
- These **Spring tags** are **aware of data binding**, It automatically sets and gets data from the model
- Tags are more similar to HTML tags for easy understanding and familiarity of usage
- As It internally encapsulates HTML tags and its attributes
- Spring form tags are included in the JSP page using **taglib**

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Spring Form Tags

- **Spring MVC Form tags** used for designing forms so that the data binding is taken care
- Let us look at few tags and their uses
- **Form tag** is a container tag that holds other form field representing tags like text fields, radio button etc.
- Attribute *action* is HTML form action attribute
- It is used to specify the URL of the submission document (Absolute / Relative)
- Attribute *method* to indicate how to send the form data (get/post)
- Attribute *modelAttribute* is gives the name of the model attribute under which the form object is exposed

```
<form:form action="InsertDepartment" method="post" modelAttribute="department">
```

- Here *department* is the Model object to / from which the form field values are mapped

Spring Form Tags

- Let us look at few tags and their uses

Tags	Purpose
<input>	This is similar to <input type="text"> of html ; text field
<password>	This is similar to <input type="password"> of html ; text field with encrypted text
<select>	This is a drop down field
<radiobutton>	This is radio button option field
<checkbox>	This is check box option field

- All these tags have path attribute which sets the property path for binding data
- In other words it specifies the model class property name mapped to the field
- Here deptno is a property under the class Department

```
Enter Department No: <sp:input path="deptno" />
```

Spring MVC Application: With Form Handling

Spring MVC Application with form data

To do List:

- Create a Maven Project
- Choose Archetype as maven-archetype-webapp
- Add the required dependencies to the pom.xml file
  - Spring-core, spring-web and spring-webmvc
- Edit the web.xml to include dispatcher servlet
- Create dispatcher-servlet.xml the spring configuration file
- Create Department Bean to represent the model class
- Create index.jsp for department operations menu
- Create InsertDepartment.jsp for accepting department details
- Create result.jsp for showing the result of form data
- Create Department Controller for processing the requests

Spring MVC Application with form data

Editing the pom.xml file : Adding dependencies

- Created Maven Project called DepartmentApplication with Archetype as maven-archetype-webapp
- Edit the pom.xml file with the following dependencies
  - spring-core
  - spring-web and
  - spring-webmvc

```
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>5.2.1.RELEASE</version>
</dependency>

<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>5.2.1.RELEASE</version>
</dependency>

<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.2.1.RELEASE</version>
</dependency>
```

Spring MVC Application with form data

Editing the web.xml file : Adding dispatcher Servlet

- Configure **DispatcherServlet** in web.xml and establish URL mappings
- Edit Web.xml file to include the below script

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
<servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>/</url-pattern>
</servlet-mapping>
```

Spring MVC Application with form data

Create Spring Configuration File

- Next create the **Spring configuration** metadata in a configuration file
- The file name would be dispatcher-servlet.xml Just as the previous demo
- we have to add the component-scan element and view resolver

```
<context:component-scan base-package="com.wipro" />

<!-- Including the bean for View Resolver -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix">
            <value>/WEB-INF/views/</value>
      </property>
      <property name="suffix">
            <value>.jsp</value>
      </property>
</bean>
```

Spring MVC Application with form data

Create Department class

- Department class is an entity class to represent the data
- Create the class under the package com.wipro.bean

```
package com.wipro.bean;


public class Department {
	private int deptno;
	private String dname;
	private String loc;
	public Department() {
	}
	public Department(int deptno, String dname, String loc) {
		super();
		this.deptno = deptno;
		this.dname = dname;
		this.loc = loc;
	}
}
//Required getters and setters are added
```

Spring MVC Application with form data

- Add the following code in Index.jsp to request for a inserting operation

```
<a href="PreInsertDepartment">Insert Department</a>
```

- Create a jsp page called InsertDepartment.jsp under WEB-INF/views
- Add the spring forms tag library

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sp" %>
```

- Create the form for Entering department data

```
<sp:form action="InsertDepartment" method="post" modelAttribute="department">
      Enter Department No: <sp:input path="deptno" />
      Enter Department Name : <sp:input path="dname"/>
      Enter Department Location : <sp:input path="loc" />
      <input type="submit"/>
</sp:form>
```

- Note: modelAttribute "***department***" is the representation of *Department Object*
- *path attribute values (deptno,dname,loc)* are properties of Department class

Spring MVC Application with form data

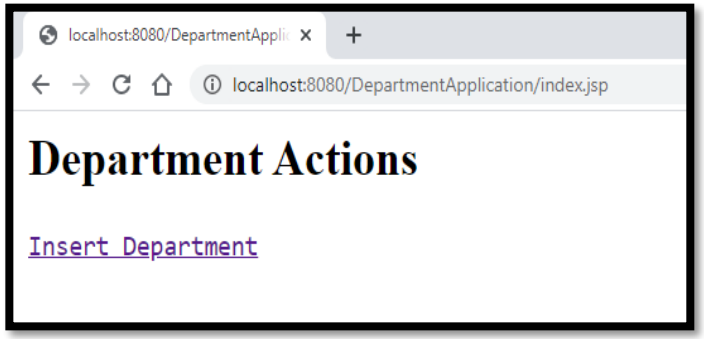
- Let us create the **DepartmentController** class under the package com.wipro.controller
- It handles the PreInsertDepartment request and InsertDepartment request
- PreInsertDepartment request from the index Page Creates a new Department Class Object
- Adds default department no value and returns back along with the InsertDepartment view

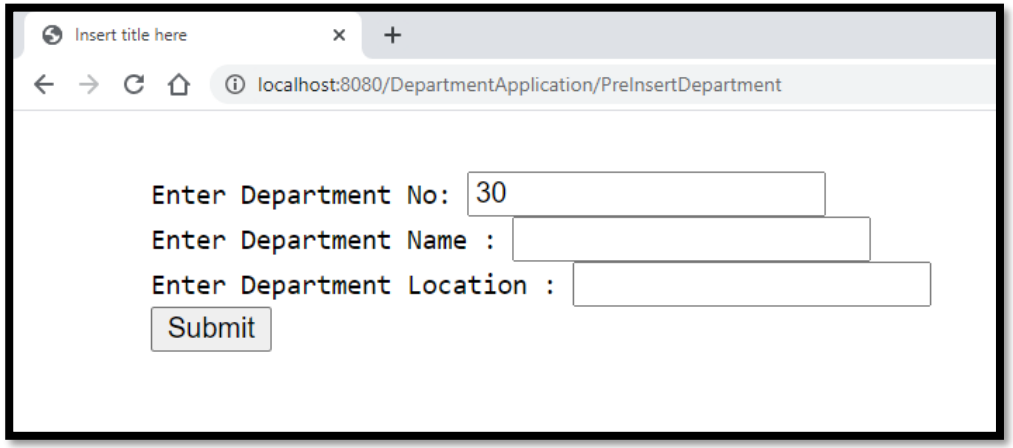
```
@Controller
public class DepartmentController {

@RequestMapping("PreInsertDepartment")
public ModelAndView preInsert() {
Department department = new Department();
department.setDeptno(30); //sets the initial value as 30
ModelAndView mv = new ModelAndView("InsertDepartment","department",department);
return mv;
}
```

This is referred as **modelAttribute** in the form tag

Spring MVC Application with form data

- Executing with Just One request handling implemented in the controller
- clicking on the hyperlink of index.jsp we get the IndertDepartment Page loaded





Spring MVC Application with form data

- On click of submit on IndertDepartment.jsp Page InsertDepartment request is raised
- Controller is edited to handle the new request

```
@RequestMapping("InsertDepartment")
public ModelAndView insertDepartment(@ModelAttribute("department") Department dept) {

ModelAndView mv = new ModelAndView("result","department",dept);
return mv;
}
```

Entered **form data** is now sent to the controller through the model Attribute and is available as the **dept** object

- Now the entered data is displayed on the new result.jsp Page

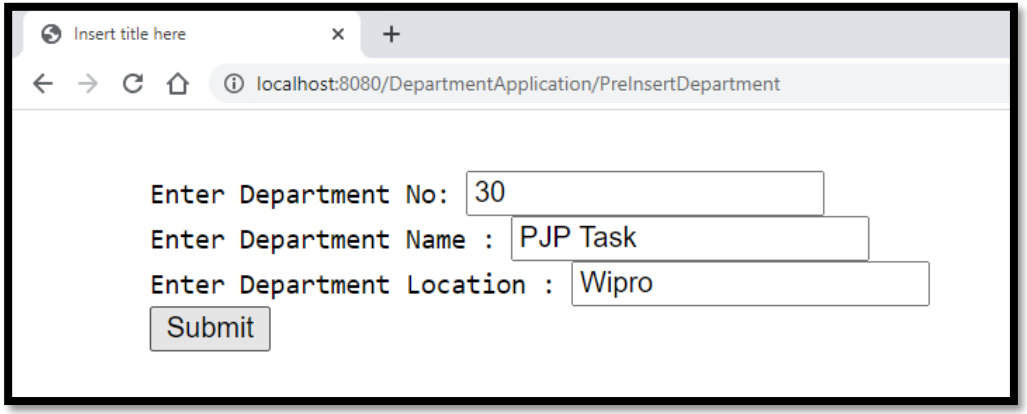
Spring MVC Application with form data

- Create result.jsp Page and add the following content
- Using Expression Language the department object details are displayed

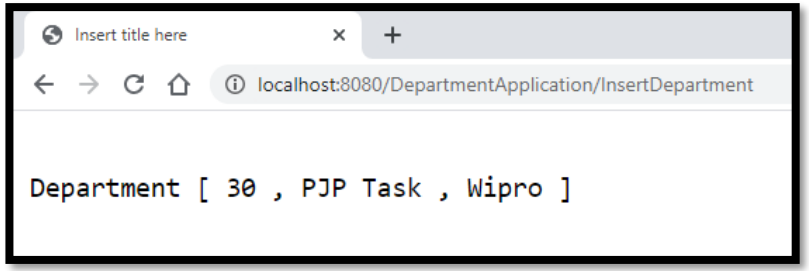
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ page isELIgnored="false" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<pre>
Department [ ${department.deptno } , ${department.dname } , ${department.loc } ]
</pre>
</body>
</html>
```

Spring MVC Application with form data

- On final execution we get



result.jsp



Spring – Hibernate Application

Spring And Hibernate

- Now Let us elevate our DepartmentApplication to **persist the Department data**
- Spring supports multiple ways to persist the data, like
  - Spring with JDBC
  - Spring Data
  - Spring ORM
- In our session we will use Spring ORM
- Spring has various APIs to support easy integration with any framework
- Thus **Spring framework supports integration with Hibernate**, Java Persistence API (JPA) and Java Data Objects (JDO) for Data Persistance and Management
- All *Hibernate configuration* details could be *provided* in *Spring Configuration file*
- With Spring IOC required objects are Autowired and there is no need to create objects of configuration to get SessionFactory object

Spring – Hibernate Application

- To the previously created Application now let us add the Hibernate Requirements
  1. Edit pom.xml file to add hibernate dependencies
  2. Edit Spring Configuration file to add Hibernate configurations
  3. Edit the Department Class to include Hibernate Annotations to mark the mappings
  4. Create DeaprtmentDao class for Data Access Layer
  5. Edit the Controller to use the Dao in order to insert the Department data to the database

Spring – Hibernate Application

1. Edit pom.xml file to add hibernate dependencies

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.4.9.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.2.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.2.2</version>
</dependency>
```

Spring – Hibernate Application

1. Edit pom.xml file to add hibernate dependencies

- In continuation we are going to add the oracle dependency for ojdbc6.jar
- For this we need to explicitly install the jar from local directory to maven repository by issuing the below given command

```
mvn install:install-file -Dfile=D:/ojdbc6.jar -DgroupId=com.oracle -DartifactId=ojdbc6
                                                    -Dversion=11.1.0  -Dpackaging=jar
```

  - –Dfile denotes the location of the ojdbc6.jar
  - –DgroupId denotes groupId of the dependency
  - –DartifactId = artifact Id of the dependency
  - –Dversion = artifact version
- Include this dependency also to the pom.xml

```
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ojdbc6</artifactId>
  <version>11.1.0</version>
</dependency>
```

Spring – Hibernate Application

2. Edit Spring Configuration file to add Hibernate configurations

- Add **<context:annotation-config/>** to activate dependency injection annotations like
  - @Autowired and @Quallifier
- Add bean definition for ***DataSource*** where we configure the connection details like
  - Driver class
  - url, username and password

```
<context:annotation-config />

<bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
<property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl" />
<property name="username" value="scott" />
<property name="password" value="tiger" />
</bean>
```

Spring – Hibernate Application

2. Edit Spring Configuration file to add Hibernate configurations *in continuation*

- Add bean definition for ***SessionFactory*** where we configure the session factory details like
  - dataSource which is mapped to the dataSource bean created earlier
  - hibernateProperties which provides hibernate property information on
    - dialect, hbm2ddl etc.
  - packagesToScan which is used to map the package where entity beans can be found

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="hibernateProperties">
            <props>
                  <prop key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</prop>
                  <prop key="hibernate.show_sql">true</prop>
                  <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
      </property>
      <property name="packagesToScan" value="com.wipro.bean" />
      /*This is one style used in sessionFactory bean  */
</bean>
```

Spring – Hibernate Application

2. Edit Spring Configuration file to add Hibernate configurations continuation

- Adding Hibernate mapping to the configuration can be done using
  - <property name="mappingResources"> by adding the list of *hbm.xml files*
  - <mapping class="Annotated class"> used for mapping *annotated class*
  - <property name="packagesToScan" > used for mapping the *annotated classes path*

- <tx:annotation-driven/> to indicate that **dependencies are Annotated**
- **HibernateTransacionManager** is an implementation for *Single Hibernate Session Factory*
- With this just @Transactional at class level is enough and we don't have to explicitly create beginTransaction , commit or rollback

```
<tx:annotation-driven />

<bean id="transactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Spring – Hibernate Application

2. Edit Spring Configuration file to add Hibernate configurations continuation

- HibernateTemplate is a helper class
- It gives simplified Hibernate Data Access Code

```
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```

- With HibernateTemplate in Spring Framework code for inserting a record of Department class looks like this

```
@Autowired
HibernateTemplate hibernateTemplate;
public boolean insertDepatment(Department department) {

      hibernateTemplate.persist(department);

return true;
}
```

Spring – Hibernate Application

3. Edit the Department Class to include Hibernate Annotations to mark the mappings
   - Include the required getters and setters

```
package com.wipro.bean;

import javax.persistence.*;

@Entity
@Table(name="MYDept")
public class Department {
	@Id
	private int deptno;
	@Column(length = 10)
	private String dname;
	@Column(length = 10)
	private String loc;
	public Department() {
	}
	public Department(int deptno, String dname, String loc) {
		this.deptno = deptno;
		this.dname = dname;
		this.loc = loc;
	}
}
```

Spring – Hibernate Application

4. Create DeaprtmentDao class for Data Access Layer

```
package com.wipro.dao;

import java.util.List;
import javax.transaction.Transactional;

import org.hibernate.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.wipro.bean.Department;

@Repository
@Transactional
public class DepartmentDao {
@Autowired
SessionFactory sessionFactory;
@Autowired
HibernateTemplate hibernateTemplate;
```

@Repository tells that the class is a DAO class

@Transactional tells hibernate transactions has to be managed

@Autowired provides requested object injections

Spring – Hibernate Application

4. Create DeaprtmentDao class for Data Access Layer

```
public int getDepartmentId() {
int id=0;
Session session = sessionFactory.openSession();
Query<Department> qry = session.createQuery("Select max(d.deptno) from Department d");
List l=qry.list();
if(l!=null && l.size()>0) {
Object b=l.get(0);
if(b!=null)
id=(Integer) b;
}
session.close();
return id+10;
}

public boolean insertDepatment(Department department) {
hibernateTemplate.persist(department);
return true;
}
}
```

Spring – Hibernate Application

5. Edit the Controller to use the Dao in order to insert the Department data to the database

```
@Controller
public class DepartmentController {

@Autowired
DepartmentDao dao;

@RequestMapping("PreInsertDepartment")
public ModelAndView preInsert() {
Department department = new Department();
department.setDeptno(dao.getDepartmentId());
ModelAndView mv = new
ModelAndView("InsertDepartment","department",
            department);
return mv;
}
```

```
@RequestMapping("InsertDepartment")
public ModelAndView
insertDepartment(@ModelAttribute("department")
Department dept) {

ModelAndView mv = new
ModelAndView("result","department",dept);
if(dao.insertDepatment(dept))
mv.addObject("msg", "Inserted Successfully");
else
mv.addObject("msg", "Insert Failed");
return mv;
}

}//Class closing
```

Spring – Hibernate Application

6. Execution Result

1. DB before Execution

```
Command Prompt - sqlplus

SQL> select * from Mydept;

    DEPTNO DNAME      LOC
---------- ---------- ------
        10 a          a
        20 b          b
        30 PJP Task   Wipro

SQL> _
```

2. Index Page

localhost:8080/DepartmentAppli

localhost:8080/DepartmentApplication/index.jsp

Department Actions

Insert Department

3. Insert Page

Insert title here

localhost:8080/DepartmentApplication/PreInsertDepartment

Enter Department No: 40

Enter Department Name :

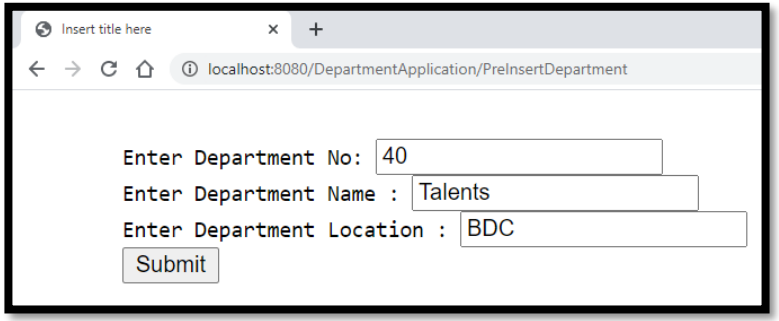
Enter Department Location :

Submit

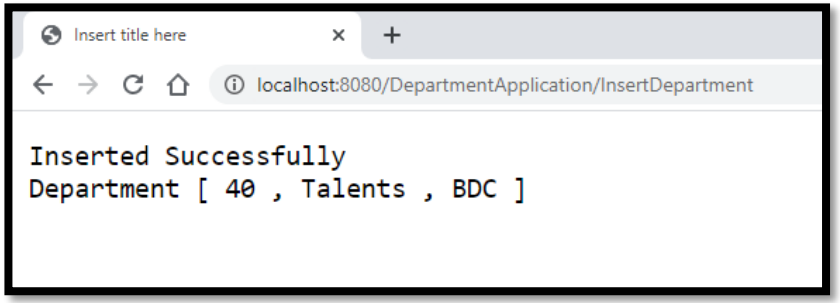
Spring – Hibernate Application

6. Execution Result

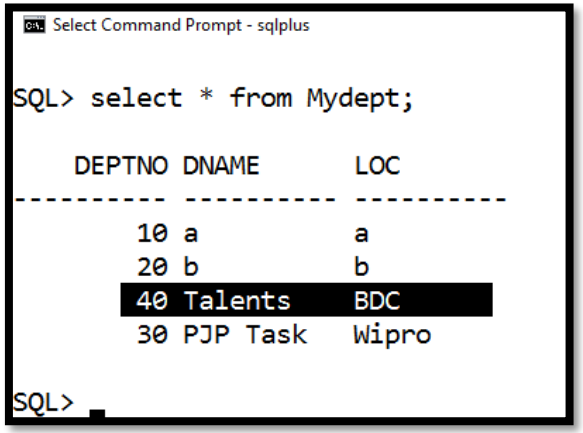
4. Insert Page Filled



5. Result Page



6. DB after execution



Summary

- In this module, we have learnt :

  - **Spring Form Tags and its uses**
  - **Form tags mapping to Entity class**
  - **Spring and Hibernate mapping**

Thank you