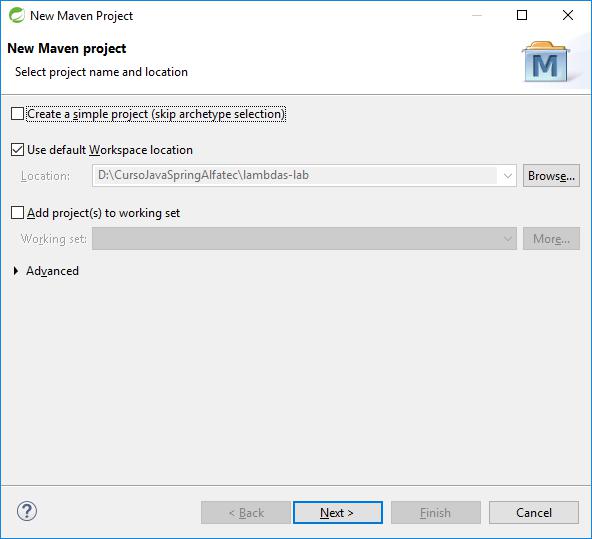
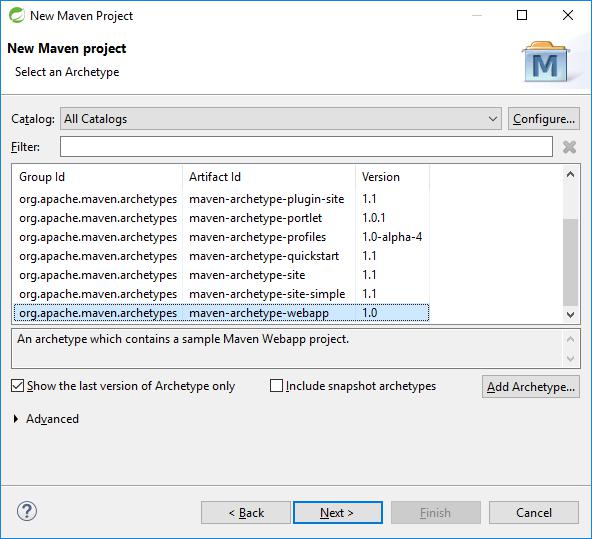
**Capítulo 1. Aplicación Base y Configuración del Entorno 1.1. Crear la estructura de directorios del proyecto**

Necesitamos crear un directorio donde alojar todos los archivos que vayamos creando, así que comenzaremos creando un proyecto en el Workspace llamado **'springapp'**. Utilizaremos Maven para facilitar la creación de la estructura de directorios del proyecto, así como la inclusión de las librerías necesarias.

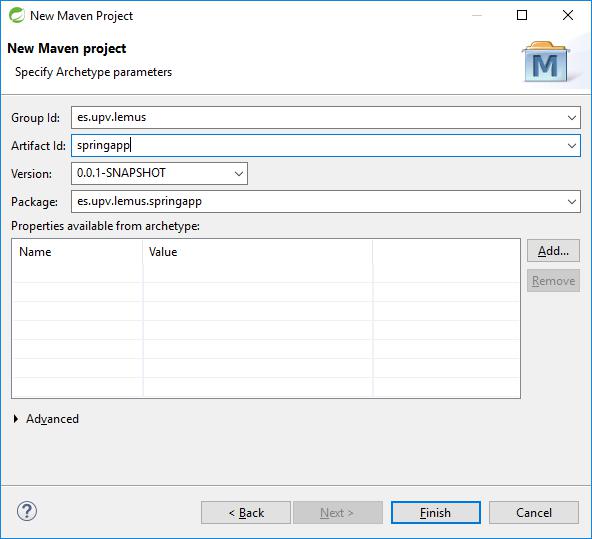


*Figura 1. Creación del nuevo proyecto usando Maven.*

3



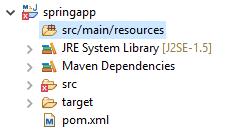
*Figura 2. Selección del arquetipo del proyecto.*

**

*Figura 3. Definición del nombre del proyecto.*

Aparecerá un warning en el proyecto causado por el uso de diferentes versiones de la JDK, ajustarlas conveniente configurando el **Build Path** del proyecto. En las últimas versiones de STS también aparece un error en el fichero JSP porque todavía no se ha incorporado la dependencia 'servlet-api'.

4

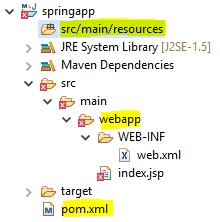


*Figura 4. Estructura de directorios creada.*

Se puede seguir adelante con el tutorial a pesar de este error, ya que lo solucionaremos en el [paso 1.5](https://www.uv.es/grimo/teaching/SpringMVCv4PasoAPaso/part1.html#step1.5).

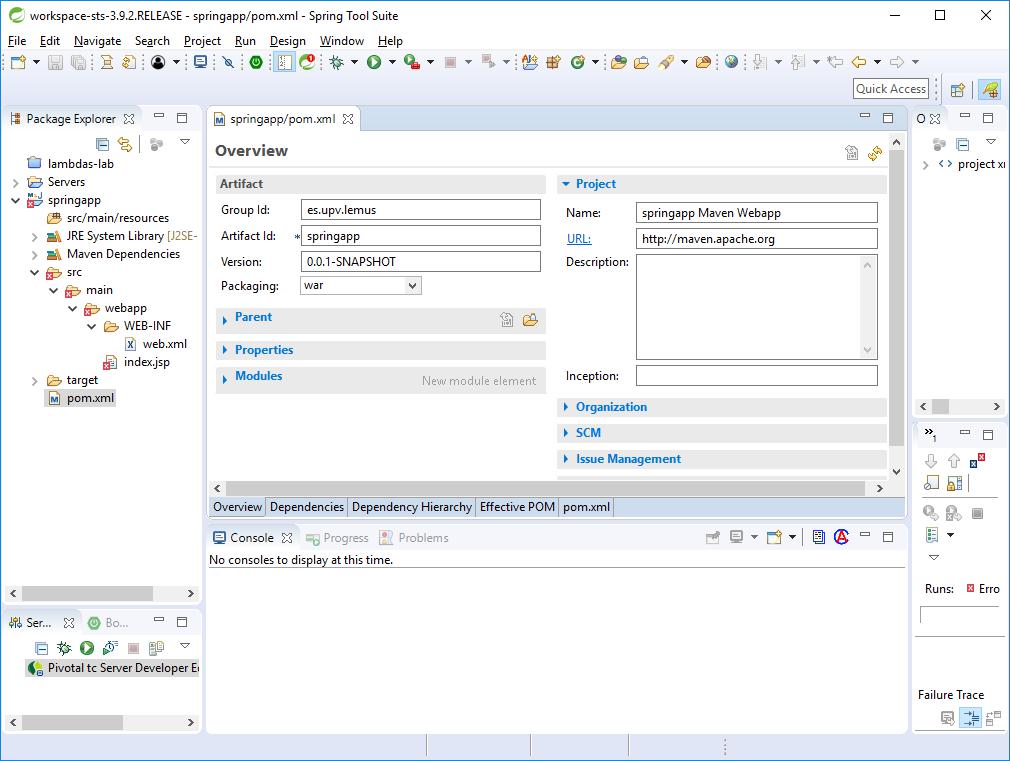
Una vez que el proyecto se ha creado, podemos identificar la siguiente estructura:

* El subdirectorio **'src/main/resources'** que contendrá todos los recursos utilizados por la aplicación.
* El subdirectorio **'src/main/webapp'** que alojará todos los archivos que no sean código fuente Java, como archivos JSP y de configuración.
* El directorio **'target'** donde se generará el archivo WAR que usaremos para almacenar y desplegar rápidamente nuestra aplicación.
* El fichero **'pom.xml'** que contiene las dependencias Maven.

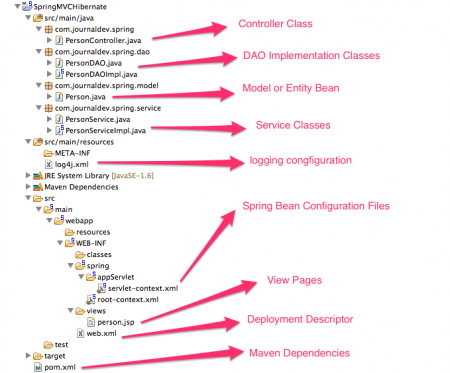


A continuación puedes ver una captura de pantalla que muestra como quedaría la estructura de directorios si has seguido las instrucciones correctamente. *(La imagen muestra dicha estructura desde el SpringSource Tool Suite (STS).*

5



*Figura 5. Estructura de directorios en el STS.*

We learned how to integrate [Spring and Hibernate](https://www.digitalocean.com/community/tutorials/spring-hibernate-integration-example-tutorial) in our last tutorial. Today we will move forward and integrate Spring MVC and Hibernate frameworks in a web application CRUD example. Our final project structure looks like below image, we will look into each of the components one by one.Note that I am using Spring 4.0.3.Release and Hibernate 4.3.5.Final versions for our example, the same program is also compatible for Spring 4 and Hibernate 3, however you need to make small changes in spring bean configuration file discussed in the last tutorial.

### **Maven Dependencies**

Let’s look at all the maven dependencies are required for hibernate and spring MVC framework integration.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.journaldev.spring</groupId>

<artifactId>SpringMVCHibernate</artifactId>

<name>SpringMVCHibernate</name>

<packaging>war</packaging>

<version>1.0.0-BUILD-SNAPSHOT</version>

<properties>

<java-version>1.6</java-version>

<org.springframework-version>4.0.3.RELEASE</org.springframework-version>

<org.aspectj-version>1.7.4</org.aspectj-version>

<org.slf4j-version>1.7.5</org.slf4j-version>

<hibernate.version>4.3.5.Final</hibernate.version>

</properties>

<dependencies>

<!-- Spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${org.springframework-version}</version>

<exclusions>

<!-- Exclude Commons Logging in favor of SLF4j -->

<exclusion>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<!-- Hibernate -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>${hibernate.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>${hibernate.version}</version>

</dependency>

<!-- Apache Commons DBCP -->

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

<!-- Spring ORM -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<!-- AspectJ -->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>${org.aspectj-version}</version>

</dependency>

<!-- Logging -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${org.slf4j-version}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>${org.slf4j-version}</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>${org.slf4j-version}</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.15</version>

<exclusions>

<exclusion>

<groupId>javax.mail</groupId>

<artifactId>mail</artifactId>

</exclusion>

<exclusion>

<groupId>javax.jms</groupId>

<artifactId>jms</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jdmk</groupId>

<artifactId>jmxtools</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jmx</groupId>

<artifactId>jmxri</artifactId>

</exclusion>

</exclusions>

<scope>runtime</scope>

</dependency>

<!-- @Inject -->

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

<!-- Servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<!-- Test -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<artifactId>maven-eclipse-plugin</artifactId>

<version>2.9</version>

<configuration>

<additionalProjectnatures>

<projectnature>org.springframework.ide.eclipse.core.springnature</projectnature>

</additionalProjectnatures>

<additionalBuildcommands>

<buildcommand>org.springframework.ide.eclipse.core.springbuilder</buildcommand>

</additionalBuildcommands>

<downloadSources>true</downloadSources>

<downloadJavadocs>true</downloadJavadocs>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.5.1</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

<compilerArgument>-Xlint:all</compilerArgument>

<showWarnings>true</showWarnings>

<showDeprecation>true</showDeprecation>

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>exec-maven-plugin</artifactId>

<version>1.2.1</version>

<configuration>

<mainClass>org.test.int1.Main</mainClass>

</configuration>

</plugin>

</plugins>

<finalName>${project.artifactId}</finalName>

</build>

</project>

Some of the dependencies above are included by STS (Spring Tool Suite) when I create Spring MVC project. Important dependencies above are spring-context, spring-webmvc, spring-tx, hibernate-core, hibernate-entitymanager and spring-orm. I am using Apache Commons DBCP for connection pooling, but in real life situations, most probably you have connection pooling done by the container and all we need is to provide the JNDI reference details to use. NOTE: I noticed that some of the readers are getting database connection issues. Notice that in my pom.xml, there is no database driver. That works for me because I have MySQL driver in tomcat lib directory and some DataSource connections configured with it. For any database connection related issues, either put the database driver in container lib or include that in pom.xml dependencies.

### **Deployment Descriptor**

We need to plugin the spring framework in our web application, that is done by configuring Spring framework DispatcherServlet as the front controller. Our web.xml file looks like below.

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="https://java.sun.com/xml/ns/javaee"

xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://java.sun.com/xml/ns/javaee https://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- Processes application requests -->

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

Most of the part is boilerplate code, most important part is the spring context file location where we will configure our spring beans and services. If you want, you can change them according to your project requirements.

### **Hibernate Entity Bean**

We are using JPA annotations in our entity bean class, however, we can also have a simple java bean and mapping details in the XML file. In that case, we need to provide mapping file details while configuring Hibernate SessionFactory in spring bean configurations.

package com.journaldev.spring.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

/\*\*

\* Entity bean with JPA annotations

\* Hibernate provides JPA implementation

\* @author pankaj

\*

\*/

@Entity

@Table(name="PERSON")

public class Person {

@Id

@Column(name="id")

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int id;

private String name;

private String country;

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 String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

@Override

public String toString(){

return "id="+id+", name="+name+", country="+country;

}

}

Our entity bean maps to PERSON table in MySQL database, notice that I have not annotated “name” and “country” fields with @Column annotation because they are of the same name. Below SQL script shows the table details.

CREATE TABLE `Person` (

`id` int(11) unsigned NOT NULL AUTO\_INCREMENT,

`name` varchar(20) NOT NULL DEFAULT '',

`country` varchar(20) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 DEFAULT CHARSET=utf8;

### **Hibernate DAO Implementation**

We will create PersonDAO interface to declare the methods that we will use in our project. Next, we will provide hibernate specific implementation for it.

package com.journaldev.spring.dao;

import java.util.List;

import com.journaldev.spring.model.Person;

public interface PersonDAO {

public void addPerson(Person p);

public void updatePerson(Person p);

public List<Person> listPersons();

public Person getPersonById(int id);

public void removePerson(int id);

}

Hibernate-specific DAO implementation looks like below.

package com.journaldev.spring.dao;

import java.util.List;

import org.hibernate.Session;

import org.hibernate.SessionFactory;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Repository;

import com.journaldev.spring.model.Person;

@Repository

public class PersonDAOImpl implements PersonDAO {

private static final Logger logger = LoggerFactory.getLogger(PersonDAOImpl.class);

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sf){

this.sessionFactory = sf;

}

@Override

public void addPerson(Person p) {

Session session = this.sessionFactory.getCurrentSession();

session.persist(p);

logger.info("Person saved successfully, Person Details="+p);

}

@Override

public void updatePerson(Person p) {

Session session = this.sessionFactory.getCurrentSession();

session.update(p);

logger.info("Person updated successfully, Person Details="+p);

}

@SuppressWarnings("unchecked")

@Override

public List<Person> listPersons() {

Session session = this.sessionFactory.getCurrentSession();

List<Person> personsList = session.createQuery("from Person").list();

for(Person p : personsList){

logger.info("Person List::"+p);

}

return personsList;

}

@Override

public Person getPersonById(int id) {

Session session = this.sessionFactory.getCurrentSession();

Person p = (Person) session.load(Person.class, new Integer(id));

logger.info("Person loaded successfully, Person details="+p);

return p;

}

@Override

public void removePerson(int id) {

Session session = this.sessionFactory.getCurrentSession();

Person p = (Person) session.load(Person.class, new Integer(id));

if(null != p){

session.delete(p);

}

logger.info("Person deleted successfully, person details="+p);

}

}

Notice that I am not using Hibernate Transaction, that is because it will be taken care by Spring framework.

### **Spring Service Classes**

Here are our service classes that are using Hibernate DAO classes to work with Person objects.

package com.journaldev.spring.service;

import java.util.List;

import com.journaldev.spring.model.Person;

public interface PersonService {

public void addPerson(Person p);

public void updatePerson(Person p);

public List<Person> listPersons();

public Person getPersonById(int id);

public void removePerson(int id);

}

package com.journaldev.spring.service;

import java.util.List;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.journaldev.spring.dao.PersonDAO;

import com.journaldev.spring.model.Person;

@Service

public class PersonServiceImpl implements PersonService {

private PersonDAO personDAO;

public void setPersonDAO(PersonDAO personDAO) {

this.personDAO = personDAO;

}

@Override

@Transactional

public void addPerson(Person p) {

this.personDAO.addPerson(p);

}

@Override

@Transactional

public void updatePerson(Person p) {

this.personDAO.updatePerson(p);

}

@Override

@Transactional

public List<Person> listPersons() {

return this.personDAO.listPersons();

}

@Override

@Transactional

public Person getPersonById(int id) {

return this.personDAO.getPersonById(id);

}

@Override

@Transactional

public void removePerson(int id) {

this.personDAO.removePerson(id);

}

}

Notice that spring declarative transaction management is applied by using @Transactional annotation.

### **Spring Controller Class**

Our DAO and Service classes are ready, it’s time to write our controller class that will take care of client requests and uses service classes to perform database specific operations and then returns the view pages.

package com.journaldev.spring;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

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.journaldev.spring.model.Person;

import com.journaldev.spring.service.PersonService;

@Controller

public class PersonController {

private PersonService personService;

@Autowired(required=true)

@Qualifier(value="personService")

public void setPersonService(PersonService ps){

this.personService = ps;

}

@RequestMapping(value = "/persons", method = RequestMethod.GET)

public String listPersons(Model model) {

model.addAttribute("person", new Person());

model.addAttribute("listPersons", this.personService.listPersons());

return "person";

}

//For add and update person both

@RequestMapping(value= "/person/add", method = RequestMethod.POST)

public String addPerson(@ModelAttribute("person") Person p){

if(p.getId() == 0){

//new person, add it

this.personService.addPerson(p);

}else{

//existing person, call update

this.personService.updatePerson(p);

}

return "redirect:/persons";

}

@RequestMapping("/remove/{id}")

public String removePerson(@PathVariable("id") int id){

this.personService.removePerson(id);

return "redirect:/persons";

}

@RequestMapping("/edit/{id}")

public String editPerson(@PathVariable("id") int id, Model model){

model.addAttribute("person", this.personService.getPersonById(id));

model.addAttribute("listPersons", this.personService.listPersons());

return "person";

}

}

Notice that I am using @Controller annotation, so that Spring framework will treat it as a Controller class to handle client requests. Also I am using @Autowired and @Qualifier annotations for injecting PersonService, we could have done it in the spring context xml file too. Recommended Read: [Spring Bean Autowiring](https://www.digitalocean.com/community/tutorials/spring-autowired-annotation)

### **Spring Bean Configuration**

Our services are ready, all we need is to wire them through spring bean configurations. Our root-context.xml file is empty, so we will look only into the servlet-context.xml file.

<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns="https://www.springframework.org/schema/mvc"

xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:beans="https://www.springframework.org/schema/beans"

xmlns:context="https://www.springframework.org/schema/context" xmlns:tx="https://www.springframework.org/schema/tx"

xsi:schemaLocation="https://www.springframework.org/schema/mvc https://www.springframework.org/schema/mvc/spring-mvc.xsd

https://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd

https://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context.xsd

https://www.springframework.org/schema/tx https://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

<!-- DispatcherServlet Context: defines this servlet's request-processing

infrastructure -->

<!-- Enables the Spring MVC @Controller programming model -->

<annotation-driven />

<!-- Handles HTTP GET requests for /resources/\*\* by efficiently serving

up static resources in the ${webappRoot}/resources directory -->

<resources mapping="/resources/\*\*" location="/resources/" />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources

in the /WEB-INF/views directory -->

<beans:bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<beans:property name="prefix" value="/WEB-INF/views/" />

<beans:property name="suffix" value=".jsp" />

</beans:bean>

<beans:bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<beans:property name="driverClassName" value="com.mysql.jdbc.Driver" />

<beans:property name="url"

value="jdbc:mysql://localhost:3306/TestDB" />

<beans:property name="username" value="pankaj" />

<beans:property name="password" value="pankaj123" />

</beans:bean>

<!-- Hibernate 4 SessionFactory Bean definition -->

<beans:bean id="hibernate4AnnotatedSessionFactory"

class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

<beans:property name="dataSource" ref="dataSource" />

<beans:property name="annotatedClasses">

<beans:list>

<beans:value>com.journaldev.spring.model.Person</beans:value>

</beans:list>

</beans:property>

<beans:property name="hibernateProperties">

<beans:props>

<beans:prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect

</beans:prop>

<beans:prop key="hibernate.show\_sql">true</beans:prop>

</beans:props>

</beans:property>

</beans:bean>

<beans:bean id="personDAO" class="com.journaldev.spring.dao.PersonDAOImpl">

<beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />

</beans:bean>

<beans:bean id="personService" class="com.journaldev.spring.service.PersonServiceImpl">

<beans:property name="personDAO" ref="personDAO"></beans:property>

</beans:bean>

<context:component-scan base-package="com.journaldev.spring" />

<tx:annotation-driven transaction-manager="transactionManager"/>

<beans:bean id="transactionManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">

<beans:property name="sessionFactory" ref="hibernate4AnnotatedSessionFactory" />

</beans:bean>

</beans:beans>

dataSource bean is defined for org.apache.commons.dbcp.BasicDataSource class for basic connection pooling. org.springframework.orm.hibernate4.LocalSessionFactoryBean bean is used for Hibernate 4 SessionFactory. For Hibernate 3, you will find similar classes as org.springframework.orm.hibernate3.LocalSessionFactoryBean and org.springframework.orm.hibernate3.AnnotationSessionFactoryBean. One important point is that when we are depending on Spring framework for Hibernate Session management, we should not define hibernate.current\_session\_context\_class, otherwise, you will get a lot of session transaction-related issues. personDAO and personService beans are self understood. transactionManager bean definition for org.springframework.orm.hibernate4.HibernateTransactionManager is required for Spring ORM to support hibernate session transaction management. For Hibernate 3, you will find similar class as org.springframework.orm.hibernate3.HibernateTransactionManager. Spring uses AOP for transaction management, you can now relate it with @Transactional annotation. Recommended Read: [Spring AOP](https://www.digitalocean.com/community/tutorials/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations) and [Spring Transaction Management](https://www.digitalocean.com/community/tutorials/spring-transaction-management-jdbc-example)

### **View Page**

Our last part of the application is the view page, notice the attributes added to Model in Controller handler methods, we will use them to create our view page. We will also use [JSTL tags](https://www.digitalocean.com/community/tutorials/jstl-tutorial-jstl-tags-example), spring core and spring form tags.

<%@ taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="https://www.springframework.org/tags" prefix="spring" %>

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

<%@ page session="false" %>

<html>

<head>

<title>Person Page</title>

<style type="text/css">

.tg {border-collapse:collapse;border-spacing:0;border-color:#ccc;}

.tg td{font-family:Arial, sans-serif;font-size:14px;padding:10px 5px;border-style:solid;border-width:1px;overflow:hidden;word-break:normal;border-color:#ccc;color:#333;background-color:#fff;}

.tg th{font-family:Arial, sans-serif;font-size:14px;font-weight:normal;padding:10px 5px;border-style:solid;border-width:1px;overflow:hidden;word-break:normal;border-color:#ccc;color:#333;background-color:#f0f0f0;}

.tg .tg-4eph{background-color:#f9f9f9}

</style>

</head>

<body>

<h1>

Add a Person

</h1>

<c:url var="addAction" value="/person/add" ></c:url>

<form:form action="${addAction}" commandName="person">

<table>

<c:if test="${!empty person.name}">

<tr>

<td>

<form:label path="id">

<spring:message text="ID"/>

</form:label>

</td>

<td>

<form:input path="id" readonly="true" size="8" disabled="true" />

<form:hidden path="id" />

</td>

</tr>

</c:if>

<tr>

<td>

<form:label path="name">

<spring:message text="Name"/>

</form:label>

</td>

<td>

<form:input path="name" />

</td>

</tr>

<tr>

<td>

<form:label path="country">

<spring:message text="Country"/>

</form:label>

</td>

<td>

<form:input path="country" />

</td>

</tr>

<tr>

<td colspan="2">

<c:if test="${!empty person.name}">

<input type="submit"

value="<spring:message text="Edit Person"/>" />

</c:if>

<c:if test="${empty person.name}">

<input type="submit"

value="<spring:message text="Add Person"/>" />

</c:if>

</td>

</tr>

</table>

</form:form>

<br>

<h3>Persons List</h3>

<c:if test="${!empty listPersons}">

<table class="tg">

<tr>

<th width="80">Person ID</th>

<th width="120">Person Name</th>

<th width="120">Person Country</th>

<th width="60">Edit</th>

<th width="60">Delete</th>

</tr>

<c:forEach items="${listPersons}" var="person">

<tr>

<td>${person.id}</td>

<td>${person.name}</td>

<td>${person.country}</td>

<td><a href="<c:url value='/edit/${person.id}' />" >Edit</a></td>

<td><a href="<c:url value='/remove/${person.id}' />" >Delete</a></td>

</tr>

</c:forEach>

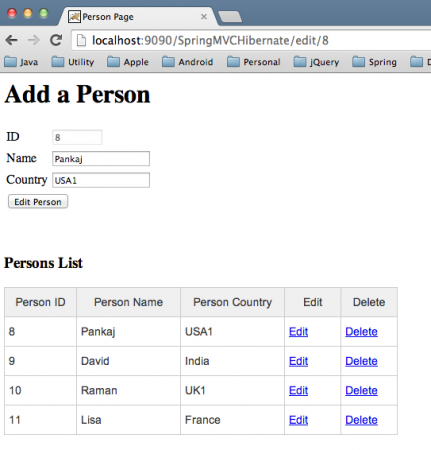
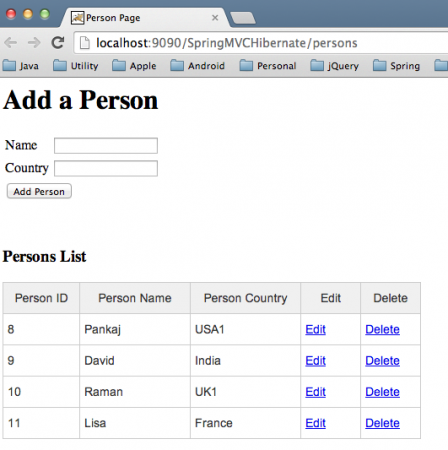
</table>

</c:if>

</body>

</html>

### **Spring MVC Hibernate Application Testing**

Just build and deploy the project into any servlet container of your choice, for example, Tomcat. Below screenshots shows the view pages for our application.You will also find similar logs in the server log file.

Hibernate: insert into PERSON (country, name) values (?, ?)

INFO : com.journaldev.spring.dao.PersonDAOImpl - Person saved successfully, Person Details=id=15, name=Pankaj, country=USA

Hibernate: select person0\_.id as id1\_0\_, person0\_.country as country2\_0\_, person0\_.name as name3\_0\_ from PERSON person0\_

INFO : com.journaldev.spring.dao.PersonDAOImpl - Person List::id=10, name=Raman, country=UK2

INFO : com.journaldev.spring.dao.PersonDAOImpl - Person List::id=11, name=Lisa, country=France

INFO : com.journaldev.spring.dao.PersonDAOImpl - Person List::id=15, name=Pankaj, country=USA

### **Summary**

This tutorial was aimed to provide sufficient details for you to getting started with Spring MVC and Hibernate integration, I hope you will find it useful. You can download the final project from below link and play around with it.