Creation of a CRUD J2EE Web application using Spring MVC and Hibernate

Author: Diana ALLAM

For this lab you need to :
- have a DAO level for CRUD access on an SQL database following steps indicated in tutorial 1
- have an Apache Tomcat installed
- verify that you have all the needed dependencies in your pom.xml file for this project, your pom file should be similar to the next one:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/m
aven-v4_0_0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.spring</groupId>
	<artifactId>SpringMVC</artifactId>
	<packaging>war</packaging>
	<version>0.0.1-SNAPSHOT</version>
	<name>SpringMVC Maven Webapp</name>
	<url>http://maven.apache.org</url>

	<properties>
		<java-version>1.7</java-version>
		<org.springframework-version>4.2.6.RELEASE</org.springframework-
version>
	</properties>

	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
			<scope>test</scope>
		</dependency>

		<dependency>
			<groupId>javax.servlet</groupId>
			<artifactId>javax.servlet-api</artifactId>
			<version>3.1.0</version>
		</dependency>

		<dependency>
```

```
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>

        <!-- http://mvnrepository.com/artifact/org.springframework/spring-context-
support -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>4.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>4.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>4.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>4.2.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.36</version>
        </dependency>

        <dependency>
            <groupId>org.codehaus.jackson</groupId>
            <artifactId>jackson-mapper-asl</artifactId>
            <version>1.9.10</version>
        </dependency>

        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.3</version>
        </dependency>

        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>4.1.0.Final</version>
        </dependency>

        <dependency>
            <groupId>org.hibernate.javax.persistence</groupId>
            <artifactId>hibernate-jpa-2.0-api</artifactId>
            <version>1.0.0.Final</version>
        </dependency>

    </dependencies>
    <build>
        <finalName>SpringMVC</finalName>
    </build>
</project>
```

Exercice 1) To use Spring MVC in order to display a Hello message on the Web view

Step 1) Configuration of src/main/webapp/WEB-INF/web.xml file

The web.xml file is your front-end controller which will redirect a call on a url to a specific servlet. It is a configuration file for web application in java. It instructs the servlet container **(Tomcat)** which class to load.

- •Filters could be defined in it.
- •Welcome file list could be defined in it.
- •Error pages could be defined in it.

Web.xml file configration parameters are:

- •**Init-param:** It is a static parameter, which is defined within servlet tag.It is a servlet level scope. At the servlet level you can get value of this.
- •**Context-param:** It is a static parameter, which is defined in web.xml file. It is an application level scope parameter. If the data does not change frequently you can store in it.
- •**Servlet:** The servlet element contains the declarative data of a servlet.
- •**Servlet-Mapping:** The servlet-mapping element defines a mapping between a servlet and a URL pattern.
- •**Listener:** The Listener element defines an event of the web application.

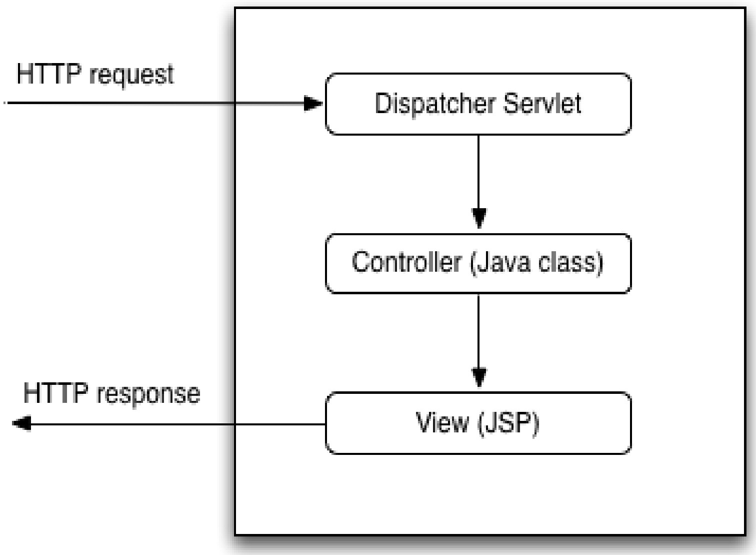
For our exercice, we will use "DispatcherServlet", "ContextLoaderListener" and "RequestContextListener" from spring web framework.

The DispatcherServlet takes an incoming URI and find the right combination of handlers (generally methods on *Controller* classes) and views (generally JSPs) that combine to form the page or resource that's supposed to be found at that location.
So if you have

- a file /WEB-INF/jsp/pages/Home.jsp
- and a *method* on a class

```
@RequestMapping(value="/pages/Home.html")
private ModelMap buildHome() {
        return somestuff;
}
```

The *Dispatcher servlet* is the bit that "knows" to call that method when a browser requests the page, and to combine its results with the matching JSP file to make an html document.
This principle is sumarized in the picture bellow



The DispatcherServlet uses *ContextLoaderListener* to listen requests and *RequestContextListener* to respond requests.

Your web.xml file should be as follows:

```
<web-app id="WebApp_ID" 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/xml/ns/j2ee/web-app_2_4.xsd">

        <display-name>Archetype Created Web Application</display-name>

        <context-param>
                <param-name>ApplicationContext</param-name>
                <param-value>/WEB-INF/applicationContext.xml</param-value>
```

```
	</context-param>

	<servlet>
		<servlet-name>SpringMVC</servlet-name>
		<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>1</load-on-startup>
	</servlet>

	<servlet-mapping>
		<servlet-name>SpringMVC</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>

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

	<listener>
		<listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
	</listener>

</web-app>
```

Step 2) Creation of applicationContext.xml

In the Web.xml file, we declared to use an applicationContext.xml file as a context-param for the servlet. Why we use application context file?

This fie is used when Spring configuration is provided to Application by application context file. It defines beans at root webapp context.

- By **<context:annotation-config/>** element, we tells application to activate beans already registered in application context. It gives general support for annotation Like @Required, @Autowired, @PostConstructor

- By **<context:component-scan base-package=”com”/>** element, we tells application to activate beans already registered in application context and scan package to find, register beans in application context.

- By < **mvc:annotation-driven** /> element, we gives application to support for annotation-driven MVC Controllers Like @RequestMapping, @Controller, @RequestBody, @ResponseBody

- By < **mvc:resources mapping=”/resources/**” location=”/resources/”** /> element,

- **<mvc:annotation-driven** /> is to tell that you can define spring beans dependencies without defining much elements in xml or implements interface or extends a class.

•**@Repository** is to tell that Class is a Dao without having to extend JpaDaoSupport or SubClass DaoSupport.

•**@Controller** is to tell that Class having all methods to handle HTTP request without implements Controller interface or extends Sub class of Controllers.

•**<context:component-scan base-package=”com” />** is to tell that your spring application will search base package for scanning classes with @Controller, @Services, @Repository, or @Component

•**<mvc:resources mapping=”/resources/**” location=”/resources/” />** is tell that spring application needs to define path location and mapping to static resources like CSS, Images or JavaScripts.

•**<bean>** is a java bean. Java Bean is a simple class with private fields and setter, getter methods. In XML we can defined it for usage. So here in this file three of beans tags are used with properties. Each property is like Class Private Field.

•**DataSource Bean:** There are two attribute defined in this bean.

  •**ID** is defined for Unique Identifier for this bean as “dataSource”
  •**CLASS** is defined for reference type of “org.apache.commons.dbcp.BasicDataSource”.

•**BasicDataSource** of Apache Commos is used for interaction with Relational Database. It helps for creating a new connection for an application.

•Properties of BasicDataSource class:

  •**driverClassName:** The JDBC Driver Class Name to be used.
  •**url:** The connection URL to be passed to JDBC driver to establish a connection.
  •**username:** The connection username to be passed to JDBC driver to establish a connection.
  •**password:** The connection password to be passed to JDBC driver to establish a connection.

•**SessionFactory Bean:** There are two attribute defined in this bean.

  •**ID** Is defined for Unique Identifier for this bean as “sessionFactory”
  •**CLASS** is defined for reference type of “org.springframework.orm.hibernate4.LocalSessionFactoryBean”.

•Properties of LocalSessionFactoryBean class:

  •**dataSource:** SessionFactory uses dataSource properties. There is ref a attribute for referring bean by its ID.
  •**packagesToScan:** To specify packages to search for autodetection of entity classes in the classpath.
  •**hibernateProperties :** It sets hibernate properties.

- •**hibernate.dialect:** To specify correct database Dialect. Each Database has different SQL code generator.
- •**hibernate.show_sql:** Write all SQL statements to CONSOLE
- •**hibernate.default_schema:** Qualifies unqualified Table names with Given TableName and Schema Name.
- •**format_sql:** Pretty print the SQL in Console and LOG.
- •**use_sql_comments:** Hibernate will generate Comments inside SQL.

•**TransactionManager Bean:** There are two attribute defined in this bean.

- •**ID** Is defined for Unique Identifier for this bean as "transactionManager"
- •**CLASS** is defined for reference type of "org.springframework.orm.hibernate4.HibernateTransactionManager".

•**Properties of HibernateTransactionManager Class:**

This transaction manager is appropriate for applications that use a single Hibernate SessionFactory for transactional data access

In tutorial 1, we created the applicationContext.xml file in order to declare the SessionFactory bean responsible about creating a session on the data base using Hibernate and sending SQL queries to MySQL database.
The applicationContext.xml file should be located at **src/main/webapp/WEB-INF/applicationContext.xml** and it should be as follows :

```
<?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:tx="http://www.springframework.org/schema/tx"
	xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

	<!-- Enable autowire -->
	<context:annotation-config />
	<context:component-scan base-package="com" />

	<mvc:annotation-driven />

	<mvc:resources mapping="/resources/**" location="/resources/" />

	<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/tests" />
		<property name="username" value="root" />
		<property name="password" value="12345" />
	</bean>

	<!-- Session Factory Declaration -->
	<bean id="sessionFactory"
		class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
		<property name="dataSource" ref="dataSource" />
		<property name="packagesToScan" value="com.entities" />
		<property name="hibernateProperties">
			<props>
				<prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
				<prop key="hibernate.show_sql">true</prop>
				<prop key="hibernate.enable_lazy_load_no_trans">true</prop>
				<prop key="hibernate.default_schema">test</prop>
				<prop key="format_sql">true</prop>
				<prop key="use_sql_comments">true</prop>
				<!-- <prop key="hibernate.hbm2ddl.auto">create</prop> -->
			</props>
		</property>
	</bean>

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

	<bean id="transactionManager"
		class="org.springframework.orm.hibernate4.HibernateTransactionManager">
		<property name="sessionFactory" ref="sessionFactory" />
	</bean>
</beans>
```

In this file, you specify the base package as "com", so you should create a package "com" under src/main/java directory.
In the *com* package, you should have the *dao* package which contains the dao interfaces, a *daoImpl* package wich contains the implementation of the interfaces in *dao* package and an entities package which contains the mapped classes to you database tables as you did by following tutorial 1.

In the following steps, we will add more packages for the service level and controllers.

Step 3) Creation of **SpringMVC-servlet.xml**

It is a servlet level configuration file. Each application context can have multiple servlet context files. It holds all the configurations for the whole servlet application. In this file, we will declare usinf an *InternalResourceViewResolver Bean*. By this resolver bean, we tell the servlet application about path of JSP views under the WEB-INF. Controllers always access them. It takes prefix for the name path and suffix for the file name.

The SpringMVC-servlet.xml file should be created under the WEB-INF folder and it should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

</beans>
```

Create then the view folder under WEB-INF, where you will put all the JSP files which describes your dynamic web pages.

Step 4) Creation of a controller

We want to create a java method which should be called when we call the URI: localhost:8080/users/page in order to display a Hello message.

- In your com package, create a controllers pakage where you will create all your controller classes.
- Create a UserController class
- Add the spring annotaion @Controller before "public Class UserController "
- Add the spring annotation @RequestMapping(value="users") in order to guide spring framework to this class when "users" appears in requested URI
- Add the spring annotation @RequestMapping(value="/page", method = RequestMethod.Get) before the method you want it to be executed when you specify "page" after "users" in your URI.
- Create a method getPage which should redirect the user to a hello.jsp page which simply display a hello message. To do that, you should use org.springframework.web.servlet.ModelAndView class from the spring framework. The ModelAndView(String jspFileName) constructor, will create a ModelAndView page which refers to the jspFileName in input parameter.

So your class will be as follows:

```
@Controller
@RequestMapping("users")
public class UsersController {

        @RequestMapping(value = "/page", method = RequestMethod.GET)
        public ModelAndView getPage() {
                ModelAndView view = new ModelAndView("hello");
                return view;
        }
```

hello.jsp file will be created in the WEB-INF/views folder as follows:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
```

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Hello Page</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.2/jquery.min.js"></script>
</head>
<body>
	<h3>Hello!</h3>
</body>
</html>
```

Step 5) Run the project on Tomcat

- Right click on your project
- Run on Server
- choose your Tomcat server
- an web browser is opened and you are redirected by default to the index.jsp file, in order to call the hello.jsp file, you should complete your URI by /users/page to see the Hello! message.

Exercice 2) To use Spring MVC in order to create a CRUD Web application

Step1) Creation of a java service interface

- in the com package, create a "services" package where you will put the interface describing the CRUD service
- create an interface UserServices as follows

```
package com.servicesapi;

import java.util.List;
import com.entities.Users;

public interface UsersService {
	public boolean saveOrUpdate(Users users);

	public List<Users> list();

	public boolean delete(Users users);
}
```

Step 2) Creation of a users view

In WEB-INF/views create a file users.jsp which calls the CRUD service using jQuery. Iquery will create a JSON message wichi specify the name of the method and the input parameters. These jsons are mapped to objects in Java thanks to GSON library which we added as dependency to the project.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
	pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Users</title>
```

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.2/jquery.min.js"></script>
</head>
<body onload="load();">

		<input type="hidden" id="user_id">
		Name: <input type="text" id="name" required="required"
name="user_name"><br>
		Email: <input type="email" id="email" required="required"
name="email"><br>
		<button onclick="submit();">Submit</button>




		<table id="table" border=1>
			<tr> <th> Name </th> <th> Email </th> <th> Edit </th> <th>
Delete </th> </tr>

		</table>



	<script type="text/javascript">
	data = "";
	submit = function(){

			$.ajax({
				url:'saveOrUpdate',
				type:'POST',
				data:{user_id:$("#user_id").val(),user_name:$('#name').val(),email:
$('#email').val()},
				success: function(response){
						alert(response.message);
						load();
				}
			});
	}

	delete_ = function(id){
		 $.ajax({
			url:'delete',
			type:'POST',
			data:{user_id:id},
			success: function(response){
					alert(response.message);
					load();
			}
		});
}


	edit = function (index){
		$("#user_id").val(data[index].user_id);
		$("#name").val(data[index].user_name);
		$("#email").val(data[index].email);
```

```
        }


        load = function(){
                $.ajax({
                        url:'list',
                        type:'POST',
                        success: function(response){
                                        data = response.data;
                                        $('.tr').remove();
                                        for(i=0; i<response.data.length; i++){
                                                $("#table").append("<tr class='tr'> <td>
"+response.data[i].user_name+" </td> <td> "+response.data[i].email+" </td> <td>
<a href='#' onclick= edit("+i+");> Edit </a>  </td> </td> <td> <a href='#'
onclick='delete_("+response.data[i].user_id+");'> Delete </a>  </td> </tr>");
                                        }
                        }
                });

        }

        </script>

</body>
</html>
```

Step 3) Creation of a java service implementation

- in the com package, create "servicesImpl" package where you will put the service implementation class of the UsersService interface
- create a class UserServicesImpl which uses the DAO classes to access the NoSQL database. The methods should return a JSON as an object of Map<String, Object>:

```java
package com.controllers;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.servlet.ModelAndView;

import com.entities.Users;
import com.servicesapi.UsersService;

@Controller
```

```java
@RequestMapping("users")
public class UsersController {

	@Autowired
	UsersService userServices;

	@RequestMapping(value = "/page", method = RequestMethod.GET)
	public ModelAndView getPage() {
		ModelAndView view = new ModelAndView("users");
		return view;
	}

	@RequestMapping(value = "/saveOrUpdate", method =
RequestMethod.POST)
	public @ResponseBody Map<String, Object> getSaved(Users users) {
		Map<String, Object> map = new HashMap<String, Object>();

		if (userServices.saveOrUpdate(users)) {
			map.put("status", "200");
			map.put("message", "Your record have been saved
successfully");
		}

		return map;
	}

	@RequestMapping(value = "/list", method = RequestMethod.POST)
	public @ResponseBody Map<String, Object> getAll(Users users) {
		Map<String, Object> map = new HashMap<String, Object>();

		List<Users> list = userServices.list();

		if (list != null) {
			map.put("status", "200");
			map.put("message", "Data found");
			map.put("data", list);
		} else {
			map.put("status", "404");
			map.put("message", "Data not found");

		}

		return map;
	}

	@RequestMapping(value = "/delete", method = RequestMethod.POST)
	public @ResponseBody Map<String, Object> delete(Users users) {
		Map<String, Object> map = new HashMap<String, Object>();

		if (userServices.delete(users)) {
```

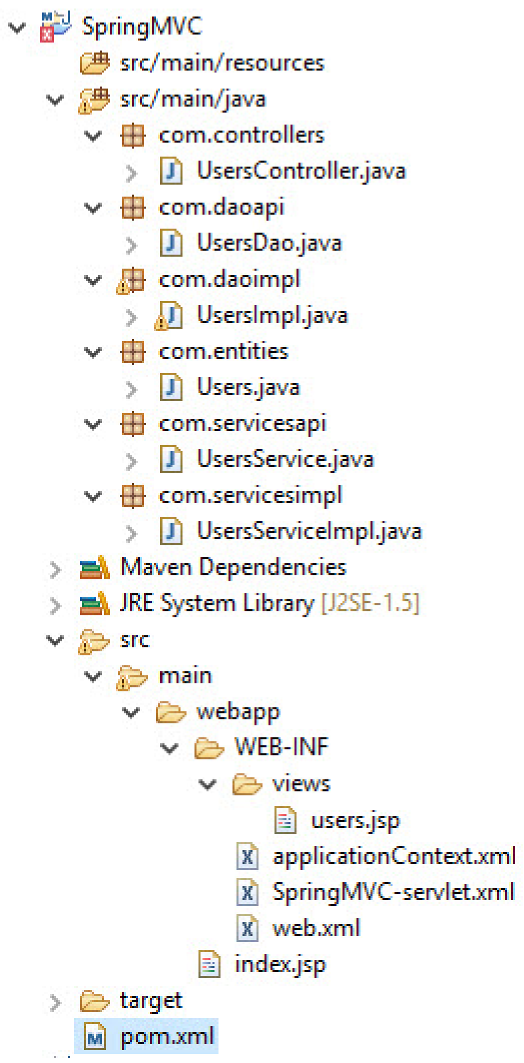
```
                        map.put("status", "200");
                        map.put("message", "Your record have been deleted
successfully");
                }

                return map;
        }
}
```

Step 4) Testing the CRUD web application on Tomcat

- Right click on your project > Run on server > choose Tomcat
- a Web browser will be opened, go to /users/page to access the content of users.jsp file
- try the different CRUD methods

Your project structure should be as follows:



```
SpringMVC
    src/main/resources
    src/main/java
        com.controllers
            UsersController.java
        com.daoapi
            UsersDao.java
        com.daoimpl
            UsersImpl.java
        com.entities
            Users.java
        com.servicesapi
            UsersService.java
        com.servicesimpl
            UsersServiceImpl.java
    Maven Dependencies
    JRE System Library [J2SE-1.5]
    src
        main
            webapp
                WEB-INF
                    views
                        users.jsp
                    applicationContext.xml
                    SpringMVC-servlet.xml
                    web.xml
                index.jsp
    target
    pom.xml
```