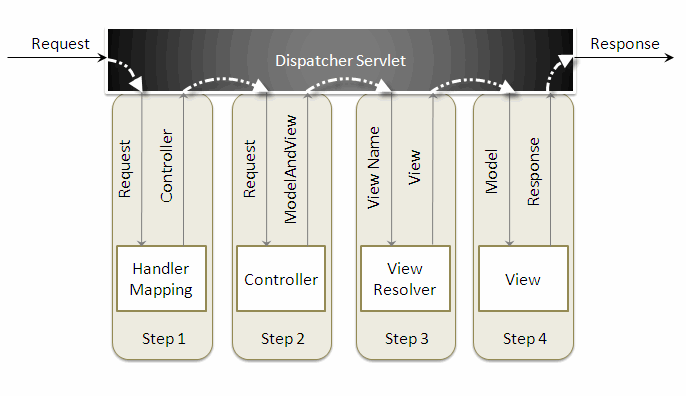
**Spring MVC Framework Tutorial**

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in seperating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

The figure below shows the flow of request in the Spring MVC Framework.

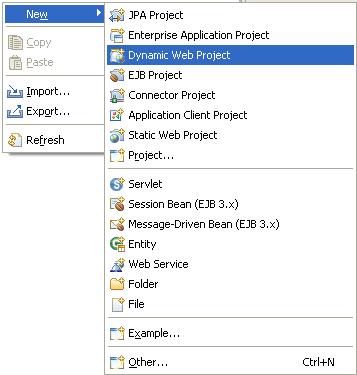


When a request is sent to the Spring MVC Framework the following sequence of events happen.

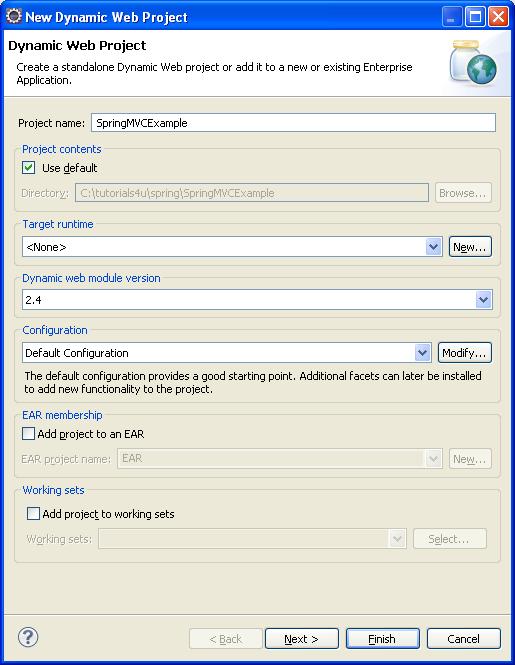
* The DispatcherServlet first receives the request.
* The DispatcherServlet consults the HandlerMapping and invokes the Controller associated with the request.
* The Controller process the request by calling the appropriate service methods and returns a ModeAndView object to the DispatcherServlet. The ModeAndView object contains the model data and the view name.
* The DispatcherServlet sends the view name to a ViewResolver to find the actual View to invoke.
* Now the DispatcherServlet will pass the model object to the View to render the result.
* The View with the help of the model data will render the result back to the user.

To understand the Spring MVC Framework we will now create a simple hello world example using the Eclipse IDE. I am using Exclipse IDE 3.4 , Spring IDE plugin, Tomcat 6.0 and Spring 3.0 to demonstrate this example.

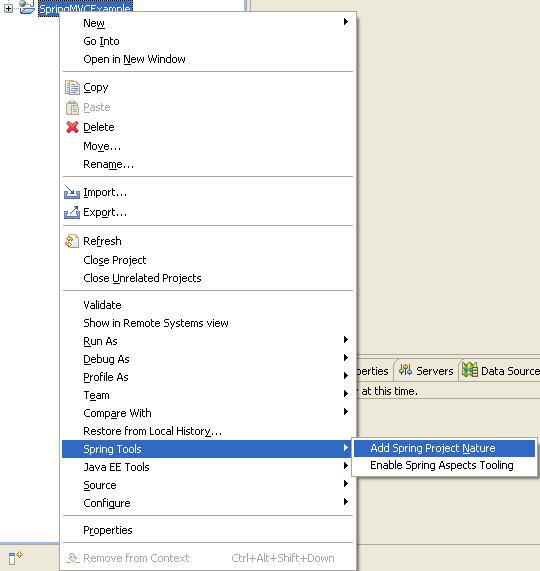
Go to File -> New -> Dynamic Web Project, to create a web project.



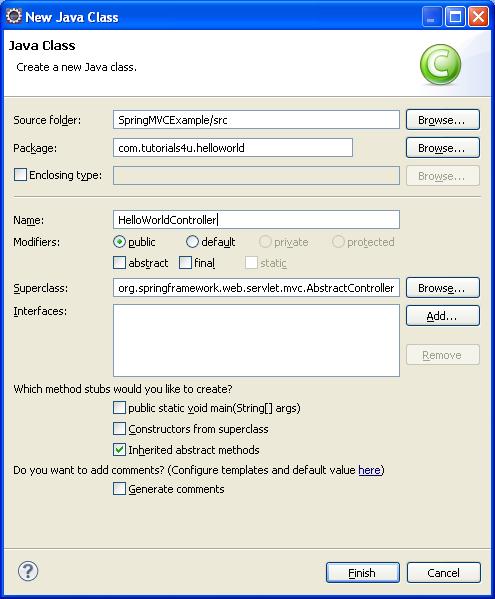
Enter the project name and click the Finish button.



Right click the project folder, and select Spring Tools -> Add Spring Project Nature, to add Spring capabilities to the web project. This feature will be available once you install the Spring IDE.



Create a new package com.tutorials4u.helloworld inside the src directory. The Spring controller class extends org.springframework.web.servlet.mvc.AbstractController class. To create a new controller class right click the src directory and create a new java class, enter the controller class name and super class name and the Finish button.



Copy the following code inside the HelloWorldController class.

01.import javax.servlet.http.HttpServletRequest;

02.import javax.servlet.http.HttpServletResponse;

03.

04.import org.springframework.web.servlet.ModelAndView;

05.import org.springframework.web.servlet.mvc.AbstractController;

06.

07.public class HelloWorldController extends AbstractController {

08.

09.    private String message;

10.

11.    @Override

12.    protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exception {

13.        return new ModelAndView("welcomePage","welcomeMessage", message);

14.    }

15.

16.    public void setMessage(String message) {

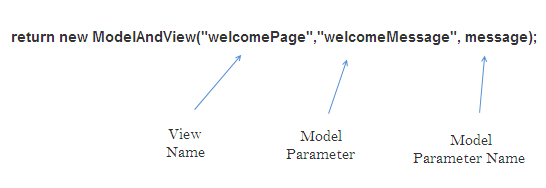
17.        this.message = message;

18.    }

19.

20.}

The HelloWorldController class has a message property that is set thru the setter injection. The HelloWorldController class should override the handleRequestInternal() method to process the request. After processing the request the handleRequestInternal() method returns a ModelAndView object back to the DispatcherServlet.



The DispatcherSevlet, as the name indicates, is a single servlet that manages the entire request-handling process. When a request is sent to the DispatcherServlet it delegates the job by invoking the appropriate controllers to process the request. Like any other servlet the DispatcherServlet need to be configured in the web deployment descriptor as shown.

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

02.<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="WebApp\_ID" version="2.5">

03.    <servlet>

04.

05.        <servlet-name>dispatcher</servlet-name>

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

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

08.    </servlet>

09.

10.    <servlet-mapping>

11.        <servlet-name>dispatcher</servlet-name>

12.        <url-pattern>\*.htm</url-pattern>

13.    </servlet-mapping>

14.    <welcome-file-list>

15.

16.        <welcome-file>welcome.htm</welcome-file>

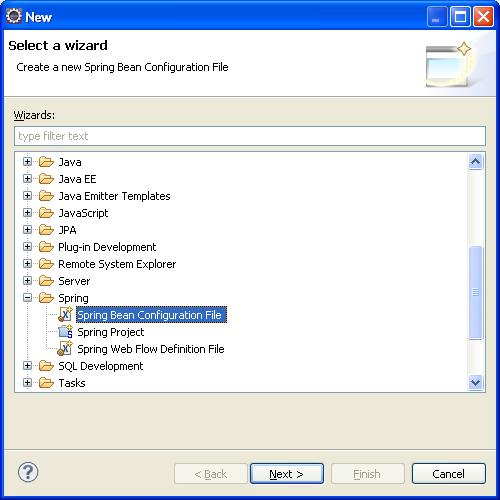
17.    </welcome-file-list>

18.</web-app>

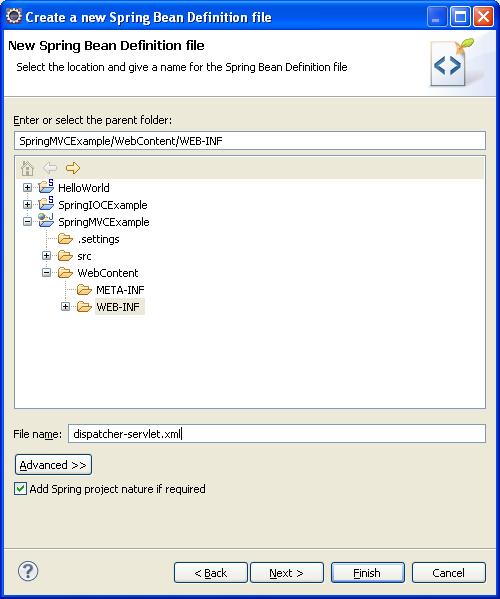
Here the servlet name is dispatcher. By default the DispatcherServlet will look for a file name dispatcher-servlet.xml to load the Spring MVC configuration. This file name is formed by concatenating the servlet name ("dispatcher") with "-servlet.xml". Here we user the the url-pattern as ".htm" inorder to hide the implementations technology to the users.

The redirect.jsp will be invoked first when we execute the Spring web application. This is the only jsp file outside the WEB-INF directory and it is here to provide a redirect to the DispatcherServlet. All the other views should be stored under the WEB-INF directory so that they can be invoked only through the controller process.

To create a bean configuration file right click the WebContent folder and select New -> Other. The following dialog box appears.



Select the Spring Bean Configuration file and click Next.



Enter the file name as "dispatcher-servlet.xml" and click the Finish button.

Now the Spring bean configuration file is created, we need to configure the Controller and the ViewResolver classes. The following code shows how to do this.

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

02.<beans xmlns="http://www.springframework.org/schema/beans"

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

04.xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

05.

06.    <bean id="viewResolver"

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

08.

09.        <property name="prefix">

10.            <value>/WEB-INF/jsp/</value>

11.        </property>

12.

13.        <property name="suffix">

14.            <value>.jsp</value>

15.

16.        </property>

17.    </bean>

18.

19.    <bean name="/welcome.htm" class="com.tutorials4u.helloworld.HelloWorldController" >

20.        <property name="message" value="Hello World!" />

21.    </bean>

22.

23.</beans>

First let's understand how to configure the controller.

1.<bean name="/welcome.htm" class="com.tutorials4u.helloworld.HelloWorldController" >

2.    <property name="message" value="Hello World!" />

3.</bean>

Here the name attribute of the bean element indicates the URL pattern to map the request. Since the id attribute can't contain special characters like "/" , we specify the URL pattern using the name attribute of the bean element. By default the DispatcherServlet uses the BeanNameUrlHandlerMapping to map the incoming request. The BeanNameUrlHandlerMapping uses the bean name as the URL pattern. Since BeanNameUrlHandlerMapping is used by default, you need not do any seperate configuration for this.

We set the message attribute of the HelloWorldController class thru setter injection. The HelloWorldController class is configured just like an another JavaBean class in the Spring application context, so like any other JavaBean we can set values to it through Dependency Injection(DI).

The redirect.jsp will redirect the request to the DispatcherServlet, which inturn consults with the BeanNameUrlHandlerMapping and invokes the HelloWorldController. The handleRequestInternal() method in the HelloWorldController class will be invoked. Here we return the message property under the name welcomeMessage and the view name welcomePage to the DispatcherServlet. As of now we only know the view name, and to find the actual view to invoke we need a ViewResolver.

The ViewResolver is configured using the following code.

01.<bean id="viewResolver"

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

03.    <property name="prefix">

04.        <value>/WEB-INF/jsp/</value>

05.    </property>

06.

07.    <property name="suffix">

08.        <value>.jsp</value>

09.    </property>

10.</bean>

Here the InternalResourceViewResolver is used to resolve the view name to the actual view. The prefix value + view name + suffix value will give the actual view location. Here the actual view location is /WEB-INF/jsp/welcomePage.jsp

The following library files are needed to run the example.

01.antlr-runtime-3.0

02.commons-logging-1.0.4

03.org.springframework.asm-3.0.0.M3

04.org.springframework.beans-3.0.0.M3

05.org.springframework.context-3.0.0.M3

06.org.springframework.context.support-3.0.0.M3

07.org.springframework.core-3.0.0.M3

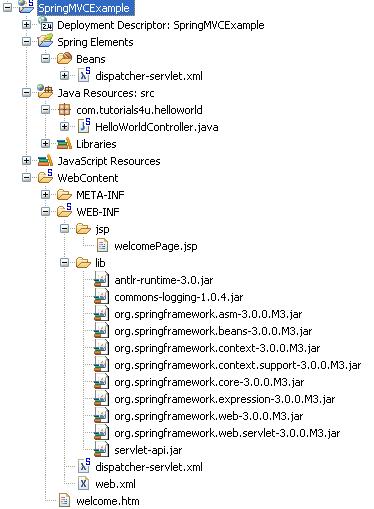
08.org.springframework.expression-3.0.0.M3

09.org.springframework.web-3.0.0.M3

10.org.springframework.web.servlet-3.0.0.M3

To execute the example run the redirect.jsp file. The following page will be displayed.

The directory structure of the example is shown below.



**Spring SimpleFormController Tutorial**

To handle forms in Spring you need to extend your controller class from SimpleFormController class. Here we will create a user registration form to understand how this works. The SimpleFormController is deprecated as of Spring 3.0 so if you are using Spring 3.0 or above use the annotate controllers instead.

01.package com.tutorials4u.web;

02.

03.import org.springframework.web.servlet.ModelAndView;

04.import org.springframework.web.servlet.mvc.SimpleFormController;

05.

06.import com.tutorials4u.domain.User;

07.import com.tutorials4u.service.UserService;

08.

09.@SuppressWarnings("deprecation")

10.public class UserController extends SimpleFormController {

11.

12.    private UserService userService;

13.

14.    public UserController() {

15.        setCommandClass(User.class);

16.        setCommandName("user");

17.    }

18.

19.    public void setUserService(UserService userService) {

20.        this.userService = userService;

21.    }

22.

23.    @Override

24.    protected ModelAndView onSubmit(Object command) throws Exception {

25.        User user = (User) command;

26.        userService.add(user);

27.        return new ModelAndView("success","user",user);

28.    }

29.

30.}

I am using Spring 3.0 so you see the SuppressWarnings annotation there. Here we extend the UserController from SimpleFormController, this makes the controller class capable of handling forms. Usually a form will be associated with a particular domain object, in our case it is the User class. In Spring this domain object is called command object by default. To refer the command object in the jsp page you need to set the command class using the setCommandClass() method in the constructor. Let say the User class has a name property, and to refer this in the jsp page you will use "command.name". You can also change this name by using the setCommandName() method. Here we set the name to user, so to access the user name in the jsp page we use "user.name".

You need to have a method to handle the form when the form is submitted, here the onSubmit() method is used for this purpose. The onSubmit() method has access to the command object, we first typecast the command object to User (our domain object) and then to register the user we call the add() method of the service class and finally return the ModelandView object.

All the forms field values will be submitted as Strings to the form controller. Spring has several pre registered property editors to convert the String values to common data types. Incase you have a custom data type you need to create custom property editors to handle them.

The User domain object has the following attributes.

01.package com.tutorials4u.domain;

02.

03.public class User {

04.

05.    private String name;

06.        private String email;

07.        private String profession;

08.        private String country;

09.        private String feedback;

10.        private String[] frameworks;

11.        private Boolean mailingList;

12.

13.        public String getName() {

14.            return name;

15.        }

16.

17.        public void setName(String name) {

18.            this.name = name;

19.        }

20.

21.        public String getEmail() {

22.            return email;

23.        }

24.

25.        public void setEmail(String email) {

26.            this.email = email;

27.        }

28.

29.        public String getProfession() {

30.            return profession;

31.        }

32.

33.        public void setProfession(String profession) {

34.            this.profession = profession;

35.        }

36.

37.        public String getCountry() {

38.            return country;

39.        }

40.

41.        public void setCountry(String country) {

42.            this.country = country;

43.        }

44.

45.        public String getFeedback() {

46.            return feedback;

47.        }

48.

49.        public void setFeedback(String feedback) {

50.            this.feedback = feedback;

51.        }

52.

53.        public String[] getFrameworks() {

54.            return frameworks;

55.        }

56.

57.        public void setFrameworks(String[] frameworks) {

58.            this.frameworks = frameworks;

59.        }

60.

61.        public Boolean getMailingList() {

62.            return mailingList;

63.        }

64.

65.        public void setMailingList(Boolean mailingList) {

66.            this.mailingList = mailingList;

67.    }

68.

69.}

Our User Service interface.

1.package com.tutorials4u.service;

2.

3.import com.tutorials4u.domain.User;

4.

5.public interface UserService {

6.

7.    public void add(User user);

8.}

Our User Service Implementation class.

01.package com.tutorials4u.service;

02.

03.import com.tutorials4u.domain.User;

04.

05.public class UserServiceImpl implements UserService {

06.

07.    @Override

08.    public void add(User user) {

09.        //Persist the user object here.

10.        System.out.println("Feedback submitted successfully");

11.

12.    }

13.

14.}

Let's now create the user feedback form "FeedbackForm.jsp" using the Spring form tags. To use the form tags you need to first import the Spring's form tag library.

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

02.<html>

03.<head>

04.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

05.<title>Feedback Page</title>

06.</head>

07.<body>

08.

09.<form:form method="POST" commandName="user">

10.    <table>

11.        <tr>

12.            <td>Name :</td>

13.            <td><form:input path="name" /></td>

14.        </tr>

15.        <tr>

16.            <td>Profession:</td>

17.            <td><form:radiobutton path="profession" value="Student" label="Student" />

18.                <form:radiobutton path="profession" value="Programmer" label="Programmer" /></td>

19.        </tr>

20.        <tr>

21.            <td>Email:</td>

22.            <td><form:input path="email" /></td>

23.        </tr>

24.        <tr>

25.            <td>Country :</td>

26.            <td><form:select path="country">

27.                <form:option value="0" label="Select" />

28.                <form:option value="India" label="India" />

29.                <form:option value="United States" label="United States" />

30.                <form:option value="United Kingdom" label="United Kingdom" />

31.                <form:option value="Europe" label="Europe" />

32.                <form:option value="Canada" label="Canada" />

33.            </form:select></td>

34.        </tr>

35.        <tr>

36.            <td>Feedback:</td>

37.            <td><form:textarea path="feedback" /></td>

38.        </tr>

39.        <tr>

40.            <td>Frameworks:</td>

41.            <td>

42.                <form:checkbox path="frameworks" value="Spring" label="Spring" />

43.                <form:checkbox path="frameworks" value="Hibernate" label="Hibernate" />

44.                <form:checkbox path="frameworks" value="Struts" label="Struts" />

45.                <form:checkbox path="frameworks" value="DOJO" label="DOJO" />

46.            </td>

47.        </tr>

48.        <tr>

49.            <td></td>

50.            <td><form:checkbox path="mailingList"

51.                label="Would you like to join our mailinglist?" /></td>

52.        </tr>

53.        <tr>

54.            <td colspan="2"><input type="submit" value="submit"></td>

55.        </tr>

56.    </table>

57.</form:form>

58.

59.</body>

60.</html>

Here the path attribute is used to bind the form fields to the domain object. Here we use the HTTP POST method to submit the form. Inorder to bind the form fields to the domain object successfully the command object should be set to the same name in the jsp page and the controller class. To set the command object name in the jsp page, use the commandName attribute of the form tag.

The web.xml file.

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

02.<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

03.xmlns="http://java.sun.com/xml/ns/javaee"

04.xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

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

06. id="WebApp\_ID" version="2.5">

07.  <display-name>Simple Form Controller Example</display-name>

08.  <servlet>

09.

10.        <servlet-name>dispatcher</servlet-name>

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

12.        </servlet-class>

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

14.    </servlet>

15.

16.    <servlet-mapping>

17.        <servlet-name>dispatcher</servlet-name>

18.        <url-pattern>\*.htm</url-pattern>

19.    </servlet-mapping>

20.    <welcome-file-list>

21.

22.        <welcome-file>feedback.htm</welcome-file>

23.    </welcome-file-list>

24.</web-app>

Next create the Spring Bean Configuration file.

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

02.

03.<beans xmlns="http://www.springframework.org/schema/beans"

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

05.xmlns:p="http://www.springframework.org/schema/p"

06.xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

07.

08.    <bean id="viewResolver"

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

10.    p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />

11.

12.    <bean id="userService" class="com.tutorials4u.service.UserServiceImpl" />

13.

14.    <bean name="/feedback.htm" class="com.tutorials4u.web.UserController" p:userService-ref="userService" p:formView="FeedbackForm" p:successView="success" />

15.

16.</beans>

As you can see, we use "p" namespace here. The "p" namespace is simple and easy to use. Using "p" namespace the properties can be supplied using attributes, rather than elements.

For injecting the simple types we use property name in the "p" namespace and for injecting references we add "-ref" suffix to it. For example we use p:formView for injecting the form view property and p:userService-ref for injecting the user service.

During the HTTP GET request the formView will be rendered. When the form is submitted (during the HTTP POST request) the onSubmit() method of the UserController class will be called, on successful execution of the method the successView will be rendered. Incase of any type conversion errors or validation errors the formView will be automatically displayed the user.

Run the example by executing the feedback.htm file. The feedback.htm file, redirect the request to "FeedbackForm.jsp".

1.<html>

2.    <head>

3.        <title>Feedback Form</title>

4.    </head>

5.    <body>

6.

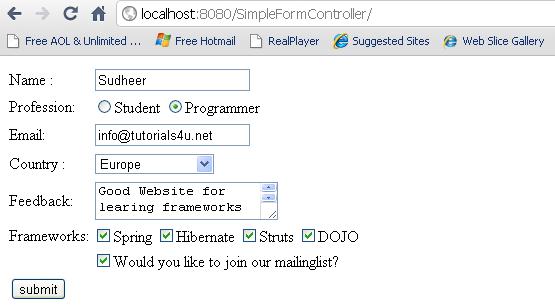
7.    </body>

8.</html>

The user feedback page will be displayed as below:



Fill the form and submit it.



The onSubmit() method of the UserController class will be called and the control will be transfered to the view "success". We use InternalResourceViewResolver here, so the success.jsp page will be dispalyed. In the success.jsp page we dispaly all the user details using the jstl tags.

01.<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

02.    pageEncoding="ISO-8859-1"%>

03.<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

04.<html>

05.<head>

06.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

07.<title>Success Page</title>

08.</head>

09.<body>

10.Details

11.<hr>

12.Name   : ${user.name} <br/>

13.Email   : ${user.email} <br/>

14.Profession      : ${user.profession} <br/>

15.Country     : ${user.country} <br/>

16.Feedback   : ${user.feedback} <br/>

17.Frameworks   : ${user.frameworks[0]},  ${user.frameworks[1]}, ${user.frameworks[2]}, ${user.frameworks[3]}<br/>

18.Mailing List: ${user.mailingList}

19.</body>

20.</html>

The success.jsp page.

