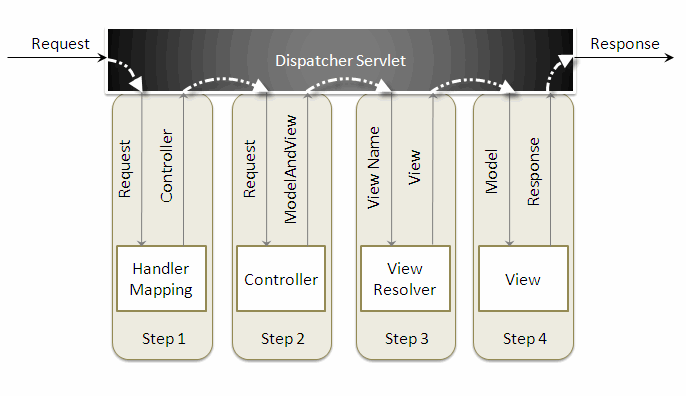
**Spring MVC Framework Tutorial**

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.

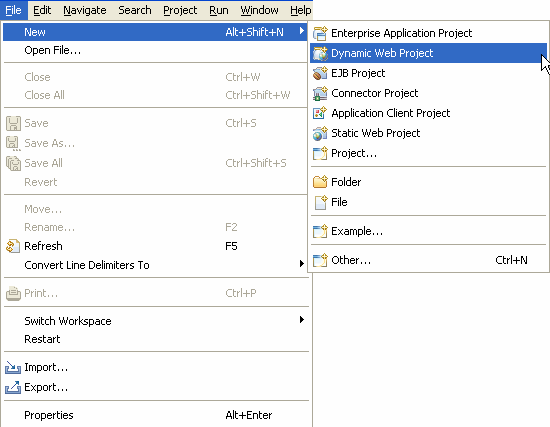


When a request is sent to the Spring MVC Framework the following sequence of events happen.

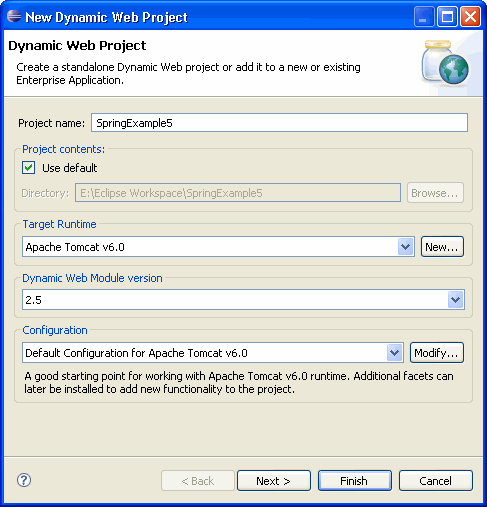
* The *DispatcherServlet* first receives the request.
* The *DispatcherServlet* consults the *HandlerMapping* and invokes the *Controller* associated with the request.
* The *Controller* process the request by calling the appropriate service methods and returns a*ModeAndView* object to the *DispatcherServlet*. 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.

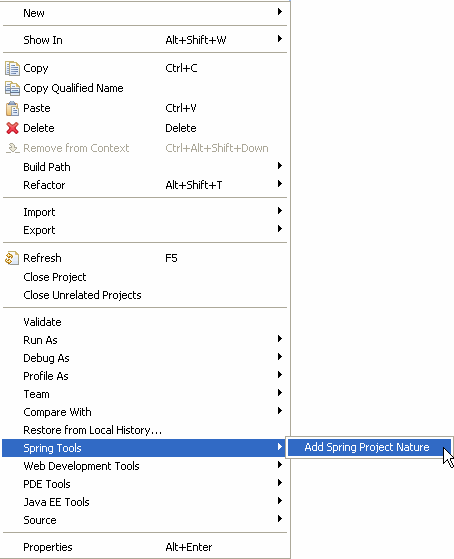
Go to*File -> New -> Dynamic Web Project*, to create a web project.



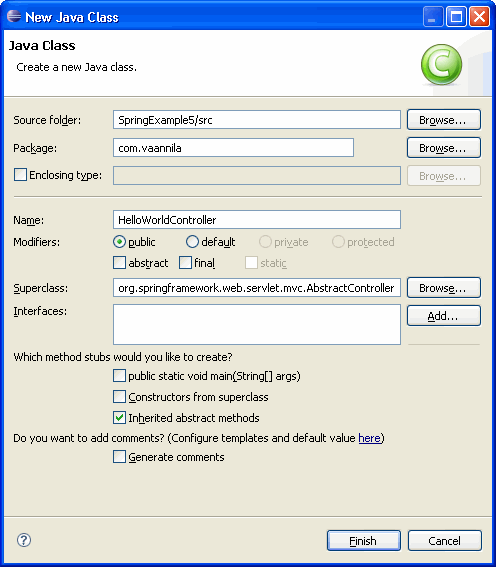
 Enter the project name and click the *Finish* button.



 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.vaannila* 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.

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

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

public class HelloWorldController extends AbstractController {

private String message;

@Override

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

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

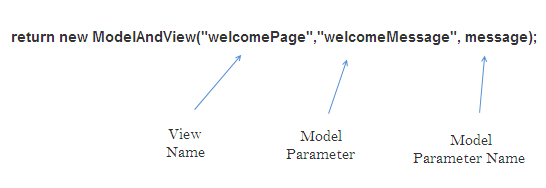
}

public void setMessage(String message) {

this.message = message;

}

}



 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.

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

<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">

<servlet>

<servlet-name>dispatcher</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

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

</servlet-mapping>

<welcome-file-list>

<welcome-file>redirect.jsp</welcome-file>

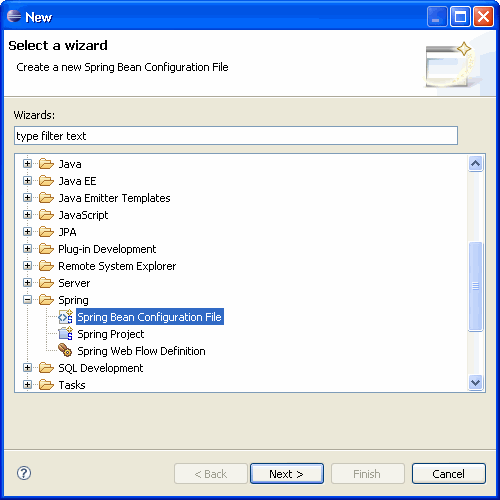
</welcome-file-list>

</web-app>

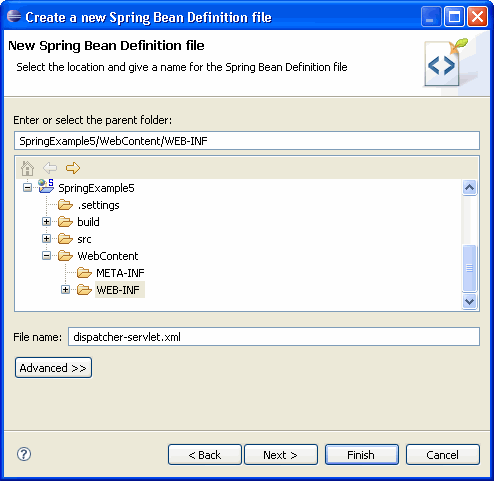
Here the servlet name is *dispatcher*. By default the *DispatcherServlet* will look for a file name*dispatcher-servlet.xml* to load the Spring MVC configuration. This file name is formed by concatenating the servlet name ("*dispatcher*") with "*-servlet.xml*". Here we user the the *url-pattern*as "*.htm*"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

<bean id="viewResolver"

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

<property name="prefix">

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

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >

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

</bean>

</beans>

First let's understand how to configure the controller.

<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >

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

</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.

<bean id="viewResolver"

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

<property name="prefix">

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

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</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.

antlr-runtime-3.0

commons-logging-1.0.4

org.springframework.asm-3.0.0.M3

org.springframework.beans-3.0.0.M3

org.springframework.context-3.0.0.M3

org.springframework.context.support-3.0.0.M3

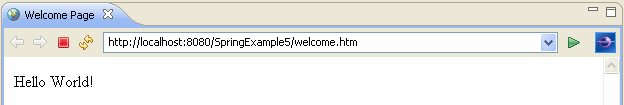
org.springframework.core-3.0.0.M3

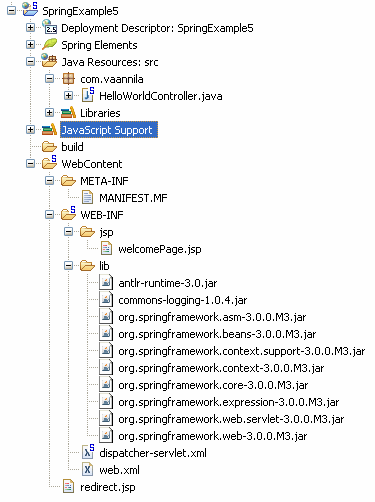
org.springframework.expression-3.0.0.M3

org.springframework.web-3.0.0.M3

org.springframework.web.servlet-3.0.0.M3

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





 You can download and try the Spring MVC example by clicking the Download link below.

Source:  [Download](https://dzone.com/sites/all/files/SpringExample5.zip)

War:  [Download](https://dzone.com/sites/all/files/SpringExample5.war)

Topics:

Let's be friends:

**Struts 2 UI Tags Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (0)**

**Save**

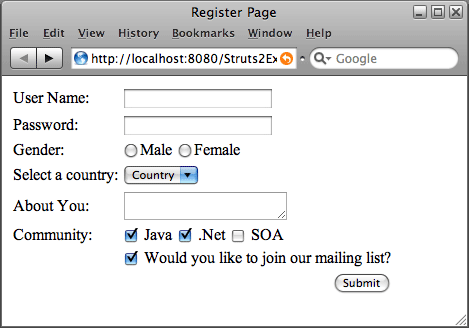
[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**104.0k Views**

Struts 2 UI Tags are simple and easy to use. You need not write any HTML code, the UI tags will automatically generate them for you based on the theme you select. By default the XHTML theme is used. The XHTML theme uses tables to position the form elements.

In this example you wil see how to create a registration page using Struts 2 UI tags. You will also learn how to pre populate the form fields, set default values to it and to retrive the values back in the jsp page.

The register.jsp looks like this



The following code is used to create the register.jsp page

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

pageEncoding="ISO-8859-1"%>

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

<%@taglib uri="/struts-tags" prefix="s"%>

<html>

<head>

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

<title>Register Page</title>

</head>

<body>

<s:form action="Register">

<s:textfield name="userName" label="User Name" />

<s:password name="password" label="Password" />

<s:radio name="gender" label="Gender" list="{'Male','Female'}" />

<s:select name="country" list="countryList" listKey="countryId" listValue="countryName" headerKey="0" headerValue="Country" label="Select a country" />

<s:textarea name="about" label="About You" />

<s:checkboxlist list="communityList" name="community" label="Community" />

<s:checkbox name="mailingList"

label="Would you like to join our mailing list?" />

<s:submit />

</s:form>

</body>

</html>

If you view the source of this page you can see the HTML codes generated based on the XHTML theme.

**Struts 2 ValueStack**

Now lets understand how the UI tags work. In Struts 2 *ValueStack* is the place where the data associated with processing of the request is stored. So all the form properties will be stored on the*ValueStack*. The *name* attribute of the UI tag is the one which links the property on the *ValueStack*.

The next important attribute of the UI tag that you need to understand is the *value* attribute. If you like to populate some default value for a specific field then you need to set that *value* attribute to that value.

The following code will by default set the value of the textfield to "*Eswar*"

<s:textfield name="userName" label="User Name" value="Eswar"/>

Here we directly specify the value in the jsp page, instead if you want to set it throught Action then, you can have a property like *defaultName*and set its value to the desired name. In this case the code will look like this.

<s:textfield name="userName" label="User Name" value="defaultName"/>

The property *defaultName* is stored on the ValueStack so its value will be set to the textfield. If you think you don't need a seperate form property to do this, then you can set the *userName* property itself to the desired value. In this case you need not specify the *value* attribute seperately. In this example we populate the *community* in this way.

The value set in the *label* attribute of the UI tags will be used to render the label for that particular field while generating the HTML code.

Now lets see the flow of the example. First the index.jsp page will be invoked by the framework.

index.jsp

~~~~~~~~~

<META HTTP-EQUIV="Refresh" CONTENT="0;URL=populateRegister.action">

Here we forward the request to the *populateRegister* URL. Based on the mapping done in the*struts.xml* file the *populate()* method in the *RegisterAction* class will be called. Here the mapping is done using the dynamic method invocation feature of Struts 2. The struts.xml file contains the following mapping.

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"

"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

<package name="default" extends="struts-default">

<action name="\*Register" method="{1}" class="vaannila.RegisterAction">

<result name="populate">/register.jsp</result>

<result name="input">/register.jsp</result>

<result name="success">/success.jsp</result>

</action>

</package>

</struts>

The register action class contains the form properties and the corresponding getter and setter methods. It also contains the *execute()* and *populate()* methods. In the populate method we first populate the values and then set the default values for the form fields. The *RegisterAction* class contains the following code.

package vaannila;

import java.util.ArrayList;

import com.opensymphony.xwork2.ActionSupport;

public class RegisterAction extends ActionSupport {

private String userName;

private String password;

private String gender;

private String about;

private String country;

private ArrayList<Country> countryList;

private String[] community;

private ArrayList<String> communityList;

private Boolean mailingList;

public String populate() {

countryList = new ArrayList<Country>();

countryList.add(new Country(1, "India"));

countryList.add(new Country(2, "USA"));

countryList.add(new Country(3, "France"));

communityList = new ArrayList<String>();

communityList.add("Java");

communityList.add(".Net");

communityList.add("SOA");

community = new String[]{"Java",".Net"};

mailingList = true;

return "populate";

}

public String execute() {

return SUCCESS;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String getGender() {

return gender;

}

public void setGender(String gender) {

this.gender = gender;

}

public String getAbout() {

return about;

}

public void setAbout(String about) {

this.about = about;

}

public String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

public ArrayList<Country> getCountryList() {

return countryList;

}

public void setCountryList(ArrayList<Country> countryList) {

this.countryList = countryList;

}

public String[] getCommunity() {

return community;

}

public void setCommunity(String[] community) {

this.community = community;

}

public ArrayList<String> getCommunityList() {

return communityList;

}

public void setCommunityList(ArrayList<String> communityList) {

this.communityList = communityList;

}

public Boolean getMailingList() {

return mailingList;

}

public void setMailingList(Boolean mailingList) {

this.mailingList = mailingList;

}

}

**Textfield and Password Tags**

Now lets see each UI tag in detail. The *textfiled* tag is used to create a textfield and *password* tag is used to create a password field. These tags are simple and uses only the common attributes discussed before.

<s:textfield name="userName" label="User Name" />

<s:password name="password" label="Password" />

**Radio Tag**

To create radio buttons we use *radio* tag. The *list* attribute of the *radio* tag is used to specify the option values. The value of the *list* attribute can be a Collection, Map, Array or Iterator. Here we use Array.

<s:radio name="gender" label="Gender" list="{'Male','Female'}" />

**Select Tag**

We dispaly the country dropdown using the *select* tag. Here we specify the option values using the*countryList* property of the *RegisterAction* class. The *countryList* is of type ArrayList and contain values of type *Country*. The *Country* class has *countryId* and *countryName* attribute. The*countryName* holds the country value to be display in the frontend and the *countryId* holds the id value to store it in the backend. Here *countryId* is the key and the *countryName* is the value. We specify this in the select tag using the *listKey* and *listValue* attribute. The first value can be specified using the *headerValue* attribute and the corresponding key value is specified using the *headerKey*attribute.

<s:select name="country" list="countryList" listKey="countryId" listValue="countryName" headerKey="0" headerValue="Country"

label="Select a country" />

**Textarea Tag**

The *textarea* tag is used to create a textarea.

<s:textarea name="about" label="About You" />

**Checkboxlist Tag**

The *checkboxlist* tag is similar to that of the *select* tag, the only difference is that it displays boxes for each option instead of a dropdown. It returns an array of String values.

<s:checkboxlist list="communityList" name="community" label="Community" />

**Checkbox Tag**

The *checkbox* tag returns a boolean value. If the checkbox is checked then true is returned else false is returned.

<s:checkbox name="mailingList" label="Would you like to join our mailing list?" />

**Submit Tag**

The *submit* tag is used to create the Submit button

<s:submit />

Now lets enter the details and submit the form. The *execute()* method in the *RegisterAction* class will be invoked this time and the user will be forwarded to the *success.jsp* page.

success.jsp

-----------

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

pageEncoding="ISO-8859-1"%>

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

<%@taglib uri="/struts-tags" prefix="s"%>

<html>

<head>

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

<title>Details Page</title>

</head>

<body>

User Name: <s:property value="userName" /><br>

Gender: <s:property value="gender" /><br>

Country: <s:property value="country" /><br>

About You: <s:property value="about" /><br>

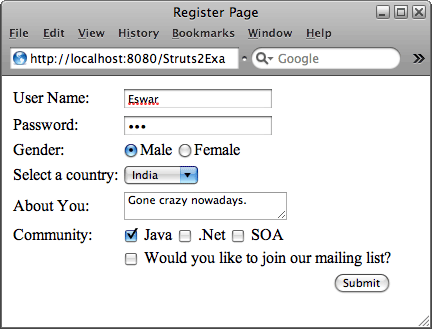
Community: <s:property value="community" /><br>

Mailing List: <s:property value="mailingList" />

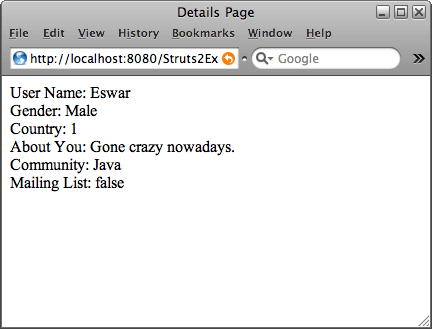
</body>

</html>

Now lets enter the following details and submit the form.



The following registration details will be displayed to the user.



You can download the Struts 2 UI Tags Example by clicking the Download link below.

|  |
| --- |
| Source :[Download](https://dzone.com/sites/all/files/Struts2Example7.zip) |
| Source + Lib :[Download](https://dzone.com/sites/all/files/Struts2Example77.war) |

Topics:

Partner Resources

Let's be friends:

**Struts DispatchAction Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (7)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**99.26k Views**

DispatchAction provides a mechanism for grouping a set of related functions into a single action, thus eliminating the need to create seperate actions for each functions. In this example we will see how to group a set of user related actions like add user, update user and delete user into a single action called UserAction.

The class UserAction extends org.apache.struts.actions.DispatchAction. This class does not provide an implementation of the execute() method as the normal Action class does. The DispatchAction uses the execute method to manage delegating the request to the individual methods based on the incoming request parameter. For example if the incoming parameter is "method=add", then the add method will be invoked. These methods should have similar signature as the execute method.

public class UserAction extends DispatchAction {

private final static String SUCCESS = "success";

public ActionForward add(ActionMapping mapping, ActionForm form,

HttpServletRequest request, HttpServletResponse response)

throws Exception {

UserForm userForm = (UserForm) form;

userForm.setMessage("Inside add user method.");

return mapping.findForward(SUCCESS);

}

public ActionForward update(ActionMapping mapping, ActionForm form,

HttpServletRequest request, HttpServletResponse response)

throws Exception {

UserForm userForm = (UserForm) form;

userForm.setMessage("Inside update user method.");

return mapping.findForward(SUCCESS);

}

public ActionForward delete(ActionMapping mapping, ActionForm form,

HttpServletRequest request, HttpServletResponse response)

throws Exception {

UserForm userForm = (UserForm) form;

userForm.setMessage("Inside delete user method.");

return mapping.findForward(SUCCESS);

}

}

If you notice the signature of the add, update and delete methods are similar to the execute method except the name. The next step is to create an action mapping for this action handler. The request parameter name is specified using the parameter attribute. Here the request parameter name is method.

<action-mappings>

<action input="/index.jsp" parameter="method" name="UserForm" path="/UserAction" scope="session" type="com.vaannila.UserAction">

<forward name="success" path="/index.jsp" />

</action>

</action-mappings>

Now lets see how to invoke a DispatchAction from jsp. We have a simple form with three buttons to add, update and delete a user. When each button is clicked a different method in UserAction class is invoked.

<html>

<head>

<script type="text/javascript">

function submitForm()

{

document.forms[0].action = "UserAction.do?method=add"

document.forms[0].submit();

}

</script>

</head>

<body>

<html:form action="UserAction" >

<table>

<tr>

<td>

<bean:write name="UserForm" property="message" />

</td>

</tr>

<tr>

<td>

<html:submit value="Add" onclick="submitForm()" />

</td>

</tr>

<tr>

<td>

<html:submit property="method" value="update" />

</td>

</tr>

<tr>

<td>

<html:submit property="method" >delete</html:submit>

</td>

</tr>

</table>

</html:form>

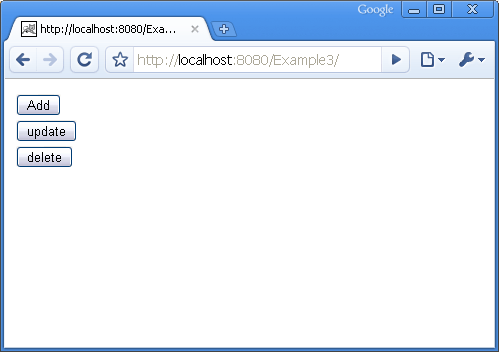
</body>

</html>

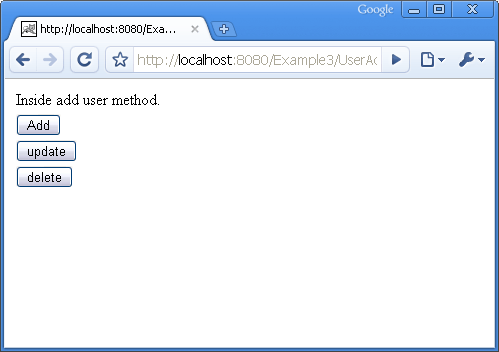
Now consider the update and the delete button. The request parameter name specified in the action handler is "method". So this should be specified as the property name for the submit button. The name of the method to be invoked and the value of the button should be the same. So when the button is clicked the corresponding method in the UserAction will be called. The delete button shows an alternate way to specify the value of the button.

Here the main constraint is the method name and the button name should be same. So we can't have an update button like this "Update". Inorder to avoid this you can call a javascript function on click of the button. Specify the action and submit the form from javascript. In this way we can have a different button name and method name. On click of the Add button the action value is set to "**UserAction.do?method=add**" and the form is submitted from javascript.

On executing the sample example the following page is displayed to the user.



After clicking the add button the following page is displayed.



You can download the source code of the DispatchAction example by clicking on the Download link below.

|  |
| --- |
| Source :[Download](https://dzone.com/sites/all/files/Example3.zip) |
| Source + Lib :[Download](https://dzone.com/sites/all/files/Exampleandlib3.zip) |

Topics:

Let's be friends:

**Struts 2 Hello World Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (1)**

**Save**

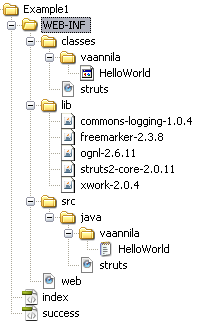
[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**91.05k Views**

In this tutorial we will see how to create a simpe Struts 2 Hello World Application. The following files are needed to create a Hello World Application.

* web.xml
* struts.xml
* HelloWorld.java
* index.jsp
* success.jsp

The following picture shows the directory structure of the Hello World application.



**web.xml**

web.xml is used to configure the servlet container properties of the hello world appliation. The filter and the filter-mapping elements are used to setup the Struts 2 FilterDispatcher. The filter is mapped to the URL pattern "*/\**". This means all the incoming request that targets to the Struts 2 action will be handled by FilterDispatcher class.

<filter>

<filter-name>struts2</filter-name>

<filter-class>org.apache.struts2.dispatcher.FilterDispatcher </filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

The gateway for our hello world application is index.jsp file. The index.jsp file should be mentioned in web.xml as shown above.

**struts.xml**

The entry point to the XML declarative architecture is struts.xml file. The struts.xml file contains the following action mapping.

<struts>

<package name="default" extends="struts-default">

<action name="HelloWorld" class="vaannila.HelloWorld">

<result name="SUCCESS">/success.jsp</result>

</action>

</package>

</struts>

**index.jsp**

The Struts 2 UI tags are simple and powerful. To use the struts tags in the jsp page the following taglib directive should be included.

<%@taglib uri="/struts-tags" prefix="s" %>

<html>

<head>

<title>Hello World</title>

</head>

<body>

<s:form action="HelloWorld" >

<s:textfield name="userName" label="User Name" />

<s:submit />

</s:form>

</body>

</html>

**HelloWorld.java**

As you see the HelloWorld class is very simple. It contains two properties one for the user name and the other for displaying the message.

public class HelloWorld {

private String message;

private String userName;

public HelloWorld() {

}

public String execute() {

setMessage("Hello " + getUserName());

return "SUCCESS";

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

}

In the execute() method of the HelloWorld action we compose the message to be displayed. Note we need not have a seperate form bean like struts 1 to access the form data. We can have a simple java class as action. The action need not extend any class or implement any interface. The only obligation is that you need to have an execute() method which returns a String and has a public scope.

**success.jsp**

In the success page we display the "*Hello User*" message using the property tag.

<%@taglib uri="/struts-tags" prefix="s" %>

<html>

<head>

<title>Hello World</title>

</head>

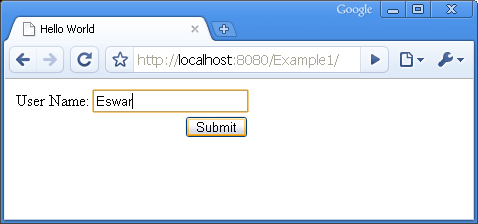
<body>

<h1><s:property value="message" /></h1>

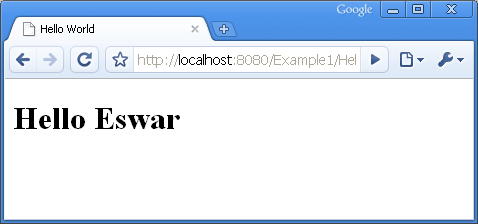
</body>

</html>

Extract the downloaded files into the webapps folder of Tomcat. Start the Tomcat server. Type the following url in the browser "**http://localhost:8080/Example1/index.jsp**". The index page will be displayed.



Enter the user name and submit the form. Hello user name message will be displayed.



You can download the Struts 2 Hello World example by clicking the download link.

|  |
| --- |
| Source :[Download](https://dzone.com/sites/all/files/Example1.zip) |
| Source + Lib :[Download](https://dzone.com/sites/all/files/Example1_1.zip) |

Topics:

Let's be friends:

**Struts Login Application Using Action Form Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(1)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**230.0k Views**

In this example we will see how to create a login application using ActionForm. The following files are required for the login application.

* login.jsp
* success.jsp
* failure.jsp
* web.xml
* struts-config.xml
* LoginAction.java
* LoginForm.java
* ApplicationResource.properties

**web.xml**

The first page that will be called in the login application is the login.jsp page. This configuration should be done in web.xml as shown below.

<welcome-file-list>

<welcome-file>login.jsp</welcome-file>

</welcome-file-list>

**login.jsp**

We use Struts HTML Tags to create login page. The form has one text field to get the user name and one password field to get the password. The form also has one submit button, which when clicked calls the login action. **<html:errors />** tag is used to display the error messages to the user.

<%@taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html>

<head>

<title>Login Page</title>

</head>

<body>

<div style="color:red">

<html:errors />

</div>

<html:form action="/Login" >

User Name :<html:text name="LoginForm" property="userName" />

Password :<html:password name="LoginForm" property="password" />

<html:submit value="Login" />

</html:form>

</body>

</html>

The user enters the user name and password and clicks the login button. The login action is invoked.

**struts-config.xml**

The validate method in the LoginForm class is called when the Form is submitted. If any errors are found then the control is returned back to the input page where the errors are displayed to the user. The input page is configured in the action tag of strut-config file. <html:errors /> tag is used to display the errors in the jsp page.

<struts-config>

<form-beans>

<form-bean name="LoginForm" type="com.vaannila.LoginForm"/>

</form-beans>

<action-mappings>

<action input="/login.jsp" name="LoginForm" path="/Login" scope="session" type="com.vaannila.LoginAction">

<forward name="success" path="/success.jsp" />

<forward name="failure" path="/failure.jsp" />

</action>

</action-mappings>

</struts-config>

Here the action is "*/Login*" , the input page is "*login.jsp*" and the corresponding action class is LoginAction.java. Now the validate method in the LoginForm class will be invoked.

**LoginForm.java**

Inside the validate method, we check whether the user name and password is entered. If not the corresponding error message is displayed to the user. The error messages are configured in the ApplicationResource.properties file.

public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {

ActionErrors errors = new ActionErrors();

if (userName == null || userName.length() < 1) {

errors.add("userName", new ActionMessage("error.userName.required"));

}

if (password == null || password.length() < 1) {

errors.add("password", new ActionMessage("error.password.required"));

}

return errors;

}

**ApplicationResource.properties**

The ApplicationResource.properties file contains the error messages. The key "*error.userName.required*" is used in the validate function to add a new error. Since the error messages are configured in a seperate properties file they can be changed anytime without making any changes to the java files or the jsp pages.

error.userName.required = User Name is required.

error.password.required = Password is required.

If either user name or password is not entered then the corresponding error message will be added to the ActionErrors. If any errors are found then the control is returned back to the input jsp page, where the error messages are displayed using the <html:errors /> tag. The validate method is used to perform the client-side validations. Once when the input data is valid the execute method in the LoginAction class is called.

**LoginAction.java**

The execute method contains the business logic of the application. Here first we typecast the ActionForm object to LoginForm, so that we can access the form variables using the getter and setter methods. If the user name and password is same then we forward the user to the success page else we forward to the failure page.

public class LoginAction extends org.apache.struts.action.Action {

private final static String SUCCESS = "success";

private final static String FAILURE = "failure";

public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response) throws Exception {

LoginForm loginForm = (LoginForm) form;

if (loginForm.getUserName().equals(loginForm.getPassword())) {

return mapping.findForward(SUCCESS);

} else {

return mapping.findForward(FAILURE);

}

}

}

Lets enter the user names and password as "*Eswar*". Since the user name and password is same the execute method will return an ActionForward "*success*". The corresponding result associated with the name "*success*" will be shown to the user. This configuration is done in struts-config.xml file.

<action-mappings>

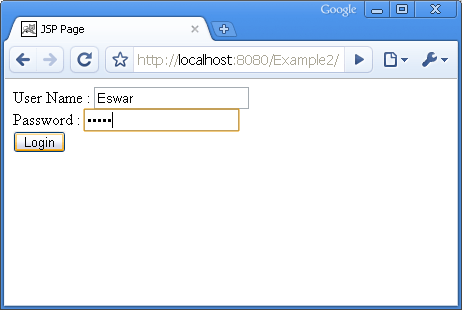
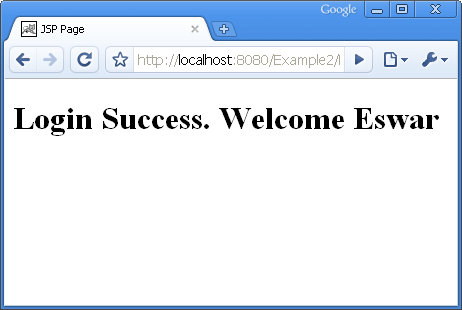
<action input="/login.jsp" name="LoginForm" path="/Login" scope="session" type="com.vaannila.LoginAction">

<forward name="success" path="/success.jsp" />

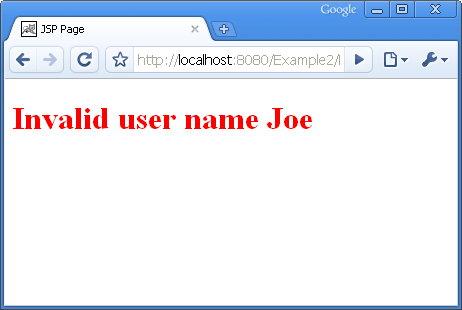
<forward name="failure" path="/failure.jsp" />

</action>

</action-mappings>

So according to the configuration in struts-config.xml the user will be forwarded to success.jsp page.  
  
  


If the user name and password did not match the user will be forwarded to the failure page. Lets try entering "*Joe*" as the user name and "*Eswar*" as the password, the following page will be displayed to the user.



You can download the source code of the Struts Login Application example by clicking on the Download link below.

Source :[Download](https://dzone.com/sites/all/files/Example2.zip)  
Source + Lib :[Download](https://dzone.com/sites/all/files/Exampleandlib2.zip)

Topics:

Let's be friends:

**Struts 2 Annotation Tutorial 2**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (1)**

**Save**

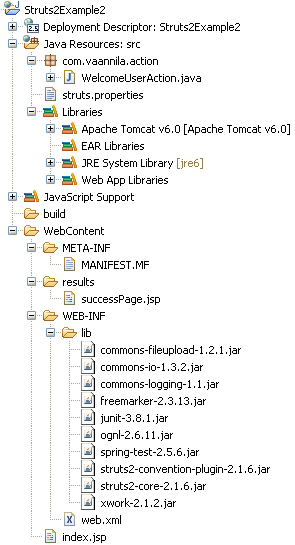
[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**29.17k Views**

This example is the continuation of the previous annotation example. If you are new to Struts 2 annotations then go through that example first ([Struts 2 Annotations - Part 1](http://vaannila.com/struts-2/struts-2-example/struts-2-annotation-example-1.html)). Here we will see the same hello user example with the following changes.

* Our Action class ends with the world **Action** and does not implement com.opensymphony.xwork2.Action.
* We use **/results** directory for storing our result pages instead of WEB-INF/content.

The directory structure of the hello user example is shown below.



Our WelcomeUserAction class is a simple pojo class. The important thing to note here is that our Action classs name ends with the word **Action**.

import org.apache.struts2.convention.annotation.Action;

import org.apache.struts2.convention.annotation.Result;

public class WelcomeUserAction {

private String userName;

private String message;

@Action(value="/welcome",results={ @Result(name="success",location="/results/successPage.jsp")})

public String execute() {

message = "Welcome " + userName + " !";

return "success";

}

public void setUserName(String userName) {

this.userName = userName;

}

public void setMessage(String message) {

this.message = message;

}

public String getUserName() {

return userName;

}

public String getMessage() {

return message;

}

}

Here we use Action and Result annotations just to show you how to use them, for simple example like this you can use the intelligent defaults provided by the Convention plug-in.

The Convention plug-in uses the Action class name to map the action URL. The Convention plug-in first removes the world Action at the end of the class name and then converts the camel case name to dashes. So by default our *WelcomeUserAction* will be invoked for the request URL *welcome-user*. But if you want the Action to be invoked for a different URL then you can do this by using the **Action annotation**.

The value of the Action annotation is *"/welcome*", this means that the action will be invoked for the request URL "*/welcome*". You can change the default action and URL mapping using the Action annotation.

Now based on the result code from action the Convention plug-in will look for the result name**welcome-resultcode** in the directory WEB-INF/content. You can change this to a different location by setting the property **struts.convention.result.path** to a new value in the Struts properties file. In this example we store the result pages in /results directory.

struts.properties file

-----------------------

struts.convention.result.path=/results

Our result page name is *successPage.jsp*, the Convention plug-in will look for a page like welcome.jsp ( the file can even be a freemaker or velocity file ) since our URL is "*/welcome*". In this case it will give an error, if we are not specifying which result it should invoke when the result is "*success*". To do this we use the Result annotation.

The **Result annotation** maps the result code with the result page. Here the result code "*success*" is mapped to the result "*/results/successPage.jsp*".

The annotations needs to be specified only when you are not using the default naming conventions, if you use them you can keep writing action classes and result pages without any configuration and the framework know exactly when to invoke them.

You can download the Struts 2 annotation example by clicking the download link below.

Source :[Download](https://dzone.com/sites/all/files/Struts2Example2.zip)

Source + Lib :[Download](https://dzone.com/sites/all/files/Struts2Example2.war)

Topics:

Let's be friends:

**Struts 2 Annotation Tutorial 1**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (2)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**40.63k Views**

We will learn annotations in struts 2 using the hello user example. In this example we will get the user name and display a welcome message to the user. There are two versions of this example, in the first one we will see how to do this by using the intelligent defaults provided by the struts 2 framework. We will not do any configuration in this example except the deployment descriptor.

The example is created using ecilpse. The war file of this example is also provided at the end of this tutorial so that you can try it yourself.

So lets start, you need to have the following jar files in the WEB-INF/lib directory.

commons-fileupload-1.2.1

commons-io-1.3.2

commons-logging-1.1

freemarker-2.3.13

junit-3.8.1

ognl-2.6.11

spring-test-2.5.6

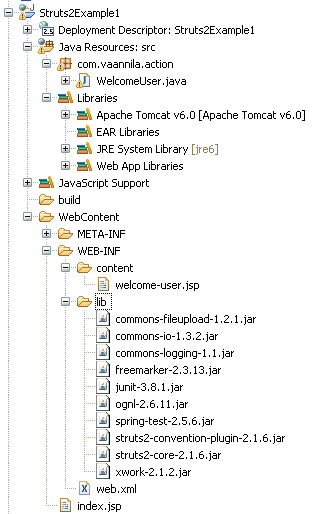
struts2-convention-plugin-2.1.6

struts2-core-2.1.6

xwork-2.1.2

You can definitely save a lot of time by having the correct versions of these jar files in the lib directory. The *struts2-convention-plugin-2.1.6* jar file is needed if your are using annotations.

This is the directory structure of the hello user example.



Now we will create the index page. This page is simple, we use the struts tags to create the page. The*textfield* tag is used to create the textfiled and the *submit* tag is used to create the submit button. The index.jsp page contains the following code.

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

pageEncoding="ISO-8859-1"%>

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

<%@taglib uri="/struts-tags" prefix="s" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Hello World</title>

</head>

<body>

<s:form action="welcome-user" >

<s:textfield name="userName" label="User Name" />

<s:submit />

</s:form>

</body>

</html>

Note the URL value of the action attribute in the form tag. In the end we will see how everything relates together.

We compose the welcome message in the *execute() method* of the WelcomeUser class and we return "*success*". The WelcomeUser class contains the following code.

package com.vaannila.action;

import com.opensymphony.xwork2.ActionSupport;

public class WelcomeUser extends ActionSupport{

private String userName;

private String message;

public String execute() {

message = "Welcome " + userName;

return SUCCESS;

}

public void setUserName(String userName) {

this.userName = userName;

}

public void setMessage(String message) {

this.message = message;

}

public String getUserName() {

return userName;

}

public String getMessage() {

return message;

}

}

Note the class name, can you find any similarities between the action URL and the class name? if yes got the concept, if no don't worry you will learn what it is in the coming pages.

We display the welcome message to the user using the *welcome-user.jsp* page. The content of the page is very simple, we just display the message. The important thing to note here is the name of the page.

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

pageEncoding="ISO-8859-1"%>

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

<html>

<head>

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

<title>Welcome User</title>

</head>

<body>

<h1>${message}</h1>

</body>

</html>

Now we will configure the web.xml for the struts 2 framework. We need to specify the filter and the filter mapping here. Except this there is no need to have any other XML configuration files.

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

<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">

<display-name>Struts2Example1</display-name>

<filter>

<filter-name>struts2</filter-name>

<filter-class>

org.apache.struts2.dispatcher.ng.filter. StrutsPrepareAndExecuteFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>struts2</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

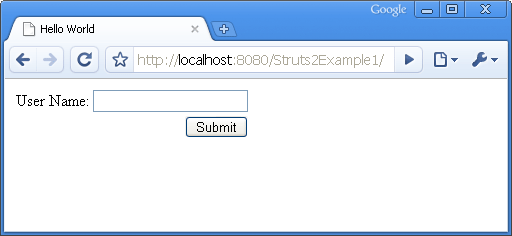
<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

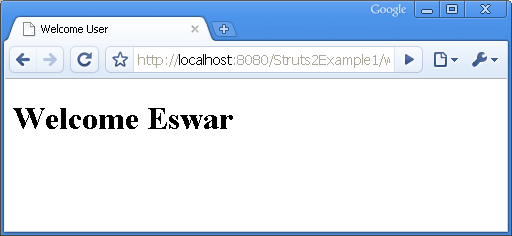
</welcome-file-list>

</web-app>

Now the coding part is complete. You can run the example by using the following URL "*http://localhost:8080/Struts2Example1/*"



Enter a user name and submit the form, you will see the following welcome-user.jsp page.



The example works fine. Now lets see how the example works.

The **Convention plug-in** is the one which does everything in the background. The Convention plug-in does the following things.

* By default the Convention plug-in looks for the action classes inside the following packages**strut, struts2, action or actions**. Here our package name is com.vaannila.**action**. Any package that matches these names will be considered as the root package for the Convention plug-in.
* The action class should either implement **com.opensymphony.xwork2.Action** interface or the name of the action class should end with **Action**. Here we extend our WelcomeUser class from **com.opensymphony.xwork2.ActionSupport** which inturn implements com.opensymphony.xwork2.Action.
* The Convention plug-in uses the action class name to map the action URL. Here our action class name is *WelcomeUser* and the URL is *welcome-user*. The plug-in converts the camel case class name to dashes to get the request URL.
* Now the Convention plug-in knows which Action class to call for a particular request. The next step is to find which result to forward based on the return value of the execute method. By default the Convention plug-in will look for result pages in **WEB-INF/content** directory.
* Now the Convention plug-in knows where to look for results, but it doesn't know which file to display inside the content directory. The Convention plug-in finds this with the help of the result code returned by the Action class. If "*success*" is returned then the Convention plug-in will look for a file name *welcome-user-success.jsp or welcome-user.jsp . It need not be a jsp file it can be even a velocity or freemaker files.* If the result value is "input" it will look for a file name*welcome-user-input.jsp or welcome-user-input.vm or welcome-user-input.ftl.*
* For more Struts 2 annotations example refer here ( [Struts 2 Annotations - Part 2](http://vaannila.com/struts-2/struts-2-example/struts-2-annotations-example-1.html) ).

You can download the Struts 2 annotation example by clicking the download link below.

Source :[Download](https://dzone.com/sites/all/files/Struts2Example1.zip)

Source + Lib :[Download](https://dzone.com/sites/all/files/Struts2Example1_0.zip)

Topics:

Let's be friends:

**Struts 2 Framework Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**46.80k Views**

A framework tries to automate the common tasks and provides a platform for the users to build applications quickly.

Struts 2 is based on the OpenSymphony**Web Works Framework**.

Struts 2 framework implements the Model-View-Controller (**MVC**) design pattern.

In Struts 2 the model, view and controller are implemented by the **action**, **result** and**FilterDispatcher** respectively.

The controller's job is to map the user request to appropriate action.

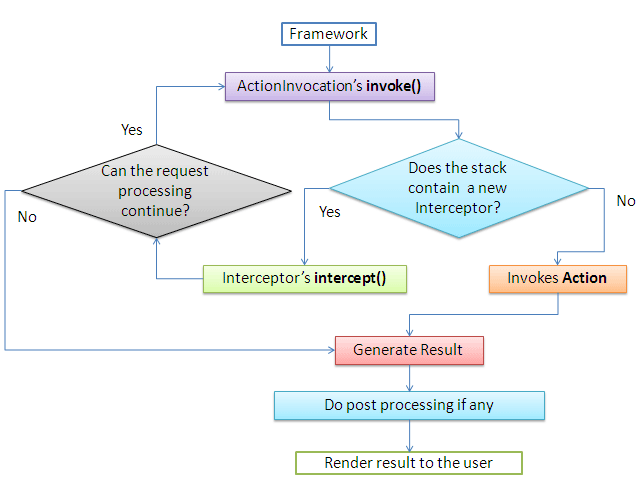
In Struts 2 FilterDispatcher does the job of Controller.

Model contains the data and the business logic.

In Struts 2 the model is implemented by the Action component.

View is the presentation component of the MVC Pattern.

In Struts 2 the view is commonly implemented using JSP, Velocity Template, Freemaker or some other presentation-layer technology.



The controller receives the user request and determine which Struts 2 action to invoke.

The framework creates an instance of this action and associate it with the newly created instance of the ActionInvocation.

In Struts 2 the invocation of action should pass through a series of interceptors as defined in the application's XML file.

The framework calls the ActionInvocations invoke() method to start the execution of the action.

Each time the invoke() method is called, ActionInvocation consults its state and executes whichever interceptor comes next.

ActionInvocation hands control over to the interceptor in the stack by calling the interceptors intercept() method.

The intercept() method of the interceptor inturn calls the invoke() method of the ActionInvocation till all the interceptors are invoked, in the end the action itself will be called and the corresponding result will be returned back to the user.

Some interceptor do work before the action is executed and some do work after the action is executed. It's not necessary that it should do something each time it is invoked.

These interceptors are invoke both before and after the action.

First all the interceptors are executed in the order they are defined in the stack.

Then the action is invoked and the result is generated.

Again all the interceptors present in the stack are invoked in the reverse order.

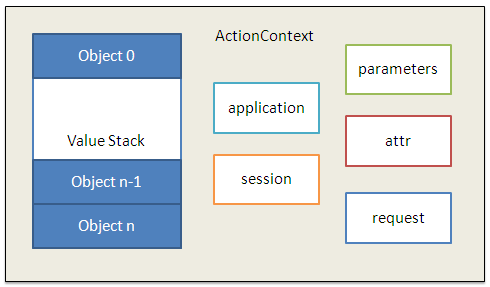
The other important features of Struts 2 are **OGNL** and **ValueStack**.

Object-Graph Navigation Language (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack.

OGNL help in **data transfer** and **type conversion**.

OGNL expression language provides simplified stytax to reference java objects.

OGNL is used to bind the java-side data properties to the string-based view layer.



In Struts 2 the action resides on the ValueStack which is a part of the **ActionContext**. ActionContext is a global storage area that holds all the data associated with the processing of a request.

When a request comes the **params** interceptor helps in moving the request data to the ValueStack.

Now the OGNL does the job of converting the string based form data to their corresponding java types. OGNL does this by using the set of available**built-in type converters**.

Again when the results are generated the OGNL converts the java types of the property on the ValueStack to the string-based HTML output.

ActionContext is thread local which means that the values stored in the ActionContext are unique per thread, this makes the Struts 2 actions **thread safe**.

Topics:

Let's be friends:

**Spring Constructor Injection**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**60.31k Views**

In this example you will learn how to set a bean property via constructor injection. Consider the following *User* bean class.

package com.vaannila;

public class User {

private String name;

private int age;

private String country;

User(String name, int age, String country)

{

this.name=name;

this.age=age;

this.country=country;

}

public String toString() {

return name + " is " + age + " years old, living in " + country;

}

}

 The *User* bean class has three attributes viz.*name, age and country*. All the three attributes are set thru constructor injection. The *toString()* method of the *User* bean class is overridden to display the user object.

Here the *beans.xml* file is used to do spring bean configuration. The following code shows how to set a property value thru constructor injection.

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

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

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.xsd">

<bean id="user" class="com.vaannila.User" >

<constructor-arg value="Eswar" />

<constructor-arg value="24"/>

<constructor-arg value="India"/>

</bean>

</beans>

 The *constructor-arg* element within the *bean* element is used to set the property value thru constructor injection. Since there is only one constructor in the *User* bean class, this code will work fine. When there is more than one constructor with the same number of arguments, then the following ambiguities will occur. Conside the following code.

<bean id="user" class="com.vaannila.User" >

<constructor-arg value="24"/>

<constructor-arg value="India"/>

</bean>

</pre>

<p>

Now which constructor do you think will be invoked? The first one with the int and the String argument, right? But for your surprise it will call the second constructor with both String arguments. Though we know the first argument is of type int and the second argument is of type String, spring interprets both as String arguments. To avoid this confusion you need to specify the <em>type</em> attribute of the <em>constructor-arg</em> element. Now with the following bean configuration, the first constructor will be invoked. </p>

<pre class="brush: xml; toolbar: false;">

<bean id="user" class="com.vaannila.User" >

<constructor-arg type="int" value="24"/>

<constructor-arg type="java.lang.String" value="India"/>

</bean>

 The *constructor-arg* element within the *bean* element is used to set the property value thru constructor injection. Since there is only one constructor in the *User* bean class, this code will work fine. When there is more than one constructor with the same number of arguments, then the following ambiguities will occur. Conside the following code.

User(String name, int age)

{

this.name=name;

this.age=age;

}

User( int age, String country)

{

this.age=age;

this.country=country;

}

 The bean configuration file.

<bean id="user" class="com.vaannila.User" >

<constructor-arg type="int" value="24"/>

<constructor-arg type="java.lang.String" value="India"/>

</bean>

 Now which constructor do you think will be invoked? The first one with the int and the String argument, right? But for your surprise it will call the second constructor with both String arguments. Though we know the first argument is of type int and the second argument is of type String, spring interprets both as String arguments. To avoid this confusion you need to specify the *type* attribute of the *constructor-arg* element. Now with the following bean configuration, the first constructor will be invoked.

<bean id="user" class="com.vaannila.User" >

<constructor-arg type="int" value="24"/>

<constructor-arg type="java.lang.String" value="India"/>

</bean>

Now which constructor do you think will be called? The second constructor, right? But again for your surprise the first constructor will be called, this is because the order in which the arguments appear in the bean configuration file will not be considered while invoking the constructor. To solve this problem you can use the *index* attribute to specify the constructor argument index.

Here is the bean configuration file after adding the index attribute.

<bean id="user" class="com.vaannila.User" >

<constructor-arg index="0" type="int" value="24"/>

<constructor-arg index="1" type="java.lang.String" value="India"/>

</bean>

Now as expected, the second constructor will be invoked.

You can download and try the constructor injection example by clicking the Download link below.

Source:  [Download](https://dzone.com/sites/all/files/SpringExample4.zip)

Source + Lib:  [Download](https://dzone.com/sites/all/files/SpringExample4_0.zip)

Topics:

Let's be friends:

**Struts Hello World Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (9)**

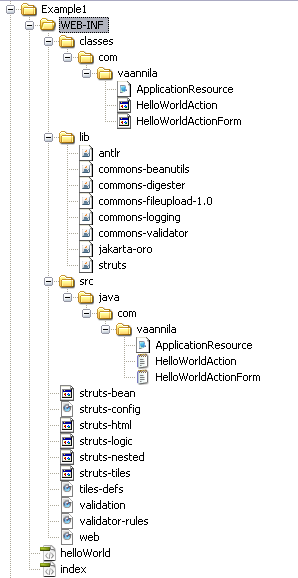
**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**89.60k Views**

Lets say a quick hello to struts. Struts follows MVC 2 pattern. The following files are needed to create a hello world application.

* index.jsp
* helloWorld.jsp
* web.xml
* struts-config.xml
* HelloWorldAction.java
* HelloWorldActionForm.java



**web.xml**

web.xml is used to configure the servlet container properties of the hello world appliation.

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

</welcome-file-list>

The gateway for our hello world application is index.jsp file. The index.jsp file should be mentioned in web.xml as shown above.

**index.jsp**

In the hello world example the index.jsp page simply forwards the request to the hello world action.

<jsp:forward page="HelloWorld.do"/>

**struts-config.xml**

struts-config.xml file is used to configure the struts framework for the hello world application. This file contains the details regarding the form bean and the action mapping.

<struts-config>

<form-beans>

<form-bean name="HelloWorldActionForm"

type="com.vaannila.HelloWorldActionForm"/>

</form-beans>

<action-mappings>

<action input="/index.jsp" name="HelloWorldActionForm" path="/HelloWorld" scope="session" type="com.vaannila.HelloWorldAction">

<forward name="success" path="/helloWorld.jsp" />

</action>

</action-mappings>

</struts-config>

**HelloWorldActionForm.java**

HelloWorldActionForm extends org.apache.struts.action.ActionForm. HelloWorldActionForm class has one String variable message and the corresponding getter and setter methods.

public class HelloWorldActionForm extends

org.apache.struts.action.ActionForm {

private String message;

public HelloWorldActionForm() {

super();

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

**HelloWorldAction.java**

HelloWorldAction class extends org.apache.struts.action.Action. The action class contains an execute method which contains the business logic of the application. To access the HelloWorldActionForm variables in the Action we need to type caste the form object to HelloWorldActionForm. Then we can access the variables using the getter and setter methods. The execute method returns a value of type ActionForward, based on its value the corresponding view will be called. This configuration is done in struts-config.xml file.

public class HelloWorldAction extends org.apache.struts.action.Action {

private final static String SUCCESS = "success";

public ActionForward execute(ActionMapping mapping,ActionForm form,

HttpServletRequest request,HttpServletResponse response) throws Exception {

HelloWorldActionForm helloWorldForm = (HelloWorldActionForm) form;

helloWorldForm.setMessage("Hello World!");

return mapping.findForward(SUCCESS);

}

}

<action-mappings>

<action input="/index.jsp" name="HelloWorldActionForm" path="/HelloWorld"

scope="session" type="com.vaannila.HelloWorldAction">

<forward name="success" path="/helloWorld.jsp" />

</action>

</action-mappings>

The name "success" is mapped to the view helloWorld.jsp. So when the execute method in the action returns "success" the request will be forwarded to the helloWold.jsp page.

**helloWorld.jsp**

In helloWorld.jsp we get the value of the form variable message and display it. We use struts bean tag to do this. The name attribute of the bean tag hold the value of the action form and the property attribute holds the value of the variable to be displayed.

<%@taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html>

<head>

<title>Hello World</title>

</head>

<body>

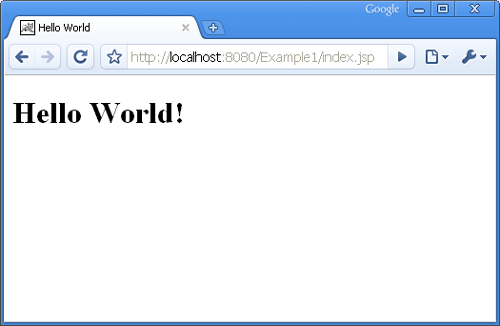
<h1>

<bean:write name="HelloWorldActionForm" property="message" />

</h1>

</body>

</html>

Extract the downloaded files into the webapps folder of the Tomcat server. Start the Tomcat server. Type the following url in the browser "**http://localhost:8080/Example1/index.jsp**". There you go, you have your first struts program up and running.  
  


You can download the source code of the Hello World example by clicking on the Download link below.

|  |
| --- |
| Source :[Download](https://dzone.com/sites/all/files/Example1_0.zip) |
| Source + Lib :[Download](https://dzone.com/sites/all/files/Exampleandlib1.zip) |

Topics:

Let's be friends:

**Maven - In Real Time**

[**[](https://dzone.com/users/995833/kkarthikeyan.html)**](https://dzone.com/users/995833/kkarthikeyan.html)**by**

[**Karthikeyan Kanagaraj**](https://dzone.com/users/995833/kkarthikeyan.html)

**·**

**Jun 14, 12 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (3)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**5,767 Views**

A perfect dependency management tool that most of us have used in our projects.   
  
I would like to share some real-time problem scenarios that Maven is able to address pretty well.  
  
Mr.Yan is working as a build management engineer. He takes care of the build and deploy environment in his company on a daily basis.  
  
After doing is routine tasks, he was frustated due to the increasing work load because of multiple OS and Java versions. His entire day was gone in editing the property files and pom files.

Finally, he started searching for a tool that would help make his work easier and atlast founfd it to be Maven... After going through the documentation, he realized "Profiles" in maven can do it.

What a Maven Profile can do for us?  
  
Maven Profile is used for the following:  
  
   \* Package the Build in different formats  
   \* Dynamically inject the dependencies  
   \* choose which source files or resources to be inculded/excluded for build (optional java files includes & excludes)  
   \* Activation based on various JDK version/OS version/Property(only System Property)  
  
Lets see a quick snapshot of Pom, 

<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/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>MyTest</groupId>  
 <artifactId>MyTest</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <profiles>  
 <profile>  
 <id>yan1</id>  
 <activation>  
 <activeByDefault>false</activeByDefault>  
 <jdk>1.5</jdk> <!--this profile will be activated if the JDK is 1.5 -->  
 </activation>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <configuration>  
 <source>1.5</source>  
 <target>1.5</target>  
 <includes> <!--can include/Exclude the sources based on the profile -->  
 <include>\*\*/MyTest1.java</include>  
 <include>\*\*/service/\*\*</include>  
 </includes>  
 </configuration>  
 </plugin>  
 </plugins>  
 </build>  
 <dependencies><!-- we can tell the dependencies for this profile alone--></dependencies>  
 </profile>  
 <profile>  
 <id>yan2</id>  
 <activation>  
 <activeByDefault>false</activeByDefault>  
 <property><!--this profile will be activated when the system property hq.version is 2 -->  
 <name>hq.version</name>  
 <value>2</value>  
 </property>  
 </activation>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <configuration>  
 <source>1.5</source>  
 <target>1.5</target>  
 <excludes><!--can include/Exclude the sources based on the profile -->  
 <exclude>\*\*/MyTest2.java</exclude>  
 </excludes>  
 </configuration>  
 </plugin>  
 </plugins>  
 </build>  
 <dependencies><!-- we can tell the dependencies for this profile alone--></dependencies>  
 </profile>  
 </profiles>  
</project>

This made his life easier... Now Yan wants to activate a profile and he types a simple command  
  
mvn -P profileName  
  
and he wants to call using properties   
  
mvn -Dprop=value  
  
After doing all these changes, now he is enjoying his life without any manual work in build managment :) :)

Topics:

Let's be friends:

**Hibernate Examples**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 ·**[**Database Zone**](https://dzone.com/database-sql-nosql-tutorials-tools-news)

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (1)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**28.54k Views**

|  |
| --- |
|  |
| |  |  | | --- | --- | | **Example** | **Source** | | [Hibernate Tools Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-tools-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample1.zip) | | [Hibernate Annotations Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample3.zip) | | [Hibernate Many-To-One Mapping Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-many-to-one-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample5.zip) | | [Hibernate One-To-One Mapping Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-one-to-one-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample6.zip) | | [Hibernate One-To-Many Mapping Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-one-to-many-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample7.zip) | | [Hibernate Many-To-Many Mapping Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-many-to-many-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample2.zip) | | [Hibernate Component Mapping Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-component-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample4.zip) | | [Hibernate Many-To-One Mapping Using Annotations Example](https://dzone.com/tutorial/hibernate/hibernate-example/hibernate-mapping-many-to-one-using-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample8.zip) | | [Hibernate One-To-One Mapping Using Annotations Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-one-to-one-using-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample9.zip) | | [Hibernate One-To-Many Mapping Using Annotations Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-one-to-many-using-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample10.zip) | | [Hibernate Many-To-Many Mapping Using Annotations Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-many-to-many-using-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample12.zip) | | [Hibernate Component Mapping Using Annotations Example](https://dzone.com/tutorials/java/hibernate/hibernate-example/hibernate-mapping-component-using-annotations-1.html) | [Download](https://dzone.com/sites/all/files/HibernateExample11.zip) | |

Topics:

Let's be friends:

**Spring Setter Injection**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**30.78k Views**

In this example you will learn how to set a bean property via setter injection. Consider the following*User* bean class.

package com.vaannila;

public class User {

private String name;

private int age;

private String country;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

public String toString() {

return name + " is " + age + " years old, living in " + country;

}

}

 The *User* bean class has three attributes viz.*name, age and country*. All the three attributes are set using the setter injection. The *toString()* method of the *User* bean class is overridden to display the user object.

Here the *beans.xml* file is used to do spring bean configuration. The following code shows how to set a property value thru setter injection.

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

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

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.xsd">

<bean id="user" class="com.vaannila.User" >

<property name="name" value="Eswar" />

<property name="age" value="24"/>

<property name="country" value="India"/>

</bean>

</beans>

The *id* attribute of the *bean* element is used to specify the bean name and the *class* attribute is used to specify the fully qualified class name of the bean. The *property* element with in the *bean* element is used to inject property value via setter injection. The *name* attribute of the *property* element represents the bean attribute and the *value* attribute specifies the corresponding property value.

Here we set "*Eswar*", "*24*" and "*India*" for the *User* bean properties *name, age* and *country*respectively. The following *Main* class is used to get the *User* bean from the Spring IoC container and dispaly its value it to the user.

package com.vaannila;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support. ClassPathXmlApplicationContext;

public class Main {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

User user = (User)context.getBean("user");

System.out.println(user);

}

}

 On executing the *Main* class the following message gets displayed on the console.

Eswar is 24 years old, living in India

You can download and try the Spring setter injection example by clicking the Download link below.

Source:  [Download](https://dzone.com/sites/all/files/SpringExample3.zip)

Source + Lib:  [Download](https://dzone.com/sites/all/files/SpringExample3_0.zip)

Topics:

Let's be friends:

**Hibernate Component Mapping Using Annotations Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 ·**[**Database Zone**](https://dzone.com/database-sql-nosql-tutorials-tools-news)

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

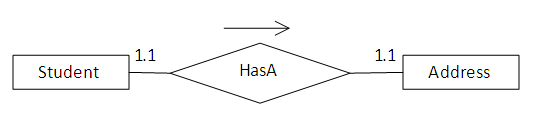
**Comment (0)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

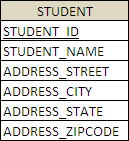
**37.15k Views**

In this example you will learn how to map components using Hibernate Annotations. Consider the following relationship between *Student* and *Address* entity.



According to the relationship each student should have a unique address.

Since the *Student* and *Address* entities are strongly related (composition relation), it is better to store them in a single table. The relational model is shown below.



*Student* class is used to create the *STUDENT* table.

package com.vaannila.student;

import javax.persistence.Column;

import javax.persistence.Embedded;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "STUDENT")

public class Student {

private long studentId;

private String studentName;

private Address studentAddress;

public Student() {

}

public Student(String studentName, Address studentAddress) {

this.studentName = studentName;

this.studentAddress = studentAddress;

}

@Id

@GeneratedValue

@Column(name = "STUDENT\_ID")

public long getStudentId() {

return this.studentId;

}

public void setStudentId(long studentId) {

this.studentId = studentId;

}

@Column(name = "STUDENT\_NAME", nullable = false, length = 100)

public String getStudentName() {

return this.studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

@Embedded

public Address getStudentAddress() {

return this.studentAddress;

}

public void setStudentAddress(Address studentAddress) {

this.studentAddress = studentAddress;

}

}

The *@Embedded* annotation is used to specify the *Address* entity should be stored in the *STUDENT*table as a component.

*@Embeddable* annotation is used to specify the *Address* class will be used as a component. The*Address* class cannot have a primary key of its own, it uses the enclosing class primary key.

package com.vaannila.student;

import javax.persistence.Column;

import javax.persistence.Embeddable;

@Embeddable

public class Address {

private String street;

private String city;

private String state;

private String zipcode;

public Address() {

}

public Address(String street, String city, String state, String zipcode) {

this.street = street;

this.city = city;

this.state = state;

this.zipcode = zipcode;

}

@Column(name = "ADDRESS\_STREET", nullable = false, length=250)

public String getStreet() {

return this.street;

}

public void setStreet(String street) {

this.street = street;

}

@Column(name = "ADDRESS\_CITY", nullable = false, length=50)

public String getCity() {

return this.city;

}

public void setCity(String city) {

this.city = city;

}

@Column(name = "ADDRESS\_STATE", nullable = false, length=50)

public String getState() {

return this.state;

}

public void setState(String state) {

this.state = state;

}

@Column(name = "ADDRESS\_ZIPCODE", nullable = false, length=10)

public String getZipcode() {

return this.zipcode;

}

public void setZipcode(String zipcode) {

this.zipcode = zipcode;

}

}

Now create the hibernate configuration file and add the *Student* and *Address* classes.

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

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class"> org.hsqldb.jdbcDriver</property>

<property name="hibernate.connection.url"> jdbc:hsqldb:hsql://localhost</property>

<property name="hibernate.connection.username">sa</property>

<property name="connection.password"></property>

<property name="connection.pool\_size">1</property>

<property name="hibernate.dialect"> org.hibernate.dialect.HSQLDialect</property>

<property name="show\_sql">true</property>

<property name="hbm2ddl.auto">create-drop</property>

<mapping class="com.vaannila.student.Student" />

<mapping class="com.vaannila.student.Address" />

</session-factory>

</hibernate-configuration>

Create the *Main* class to run the example.

package com.vaannila.student;

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.Transaction;

import com.vaannila.util.HibernateUtil;

public class Main {

public static void main(String[] args) {

Session session = HibernateUtil.getSessionFactory().openSession();

Transaction transaction = null;

try {

transaction = session.beginTransaction();

Address address = new Address("OMR Road", "Chennai", "TN", "600097");

Student student = new Student("Eswar", address);

session.save(student);

transaction.commit();

} catch (HibernateException e) {

transaction.rollback();

e.printStackTrace();

} finally {

session.close();

}

}

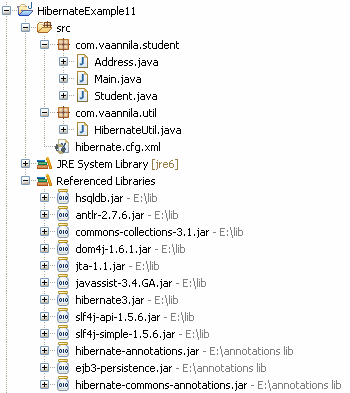
}

On executing the *Main* class you will see the following output.

http://cdn.dzone.com/static/images/vaannila/hibernate/hibernateComponentPic2.gif

Each student has one address and the values are stored in the same *STUDENT* table.

The folder structure of the example is shown below.



You can download the source code of this example here.

Source :[Download](https://dzone.com/sites/all/files/HibernateExample11.zip)

Topics:

Let's be friends:

**Spring IoC Tutorial**

[**[](https://dzone.com/users/1031353/eswar_vaannila.html)**](https://dzone.com/users/1031353/eswar_vaannila.html)**by**

[**Meyyappan Muthuraman**](https://dzone.com/users/1031353/eswar_vaannila.html)

**·**

**Jun 14, 12 · Zone**

**Like** [**(0)**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**Comment (10)**

**Save**

[**Tweet**](https://dzone.com/tutorials/java/spring/spring-ioc-1.html)

**83.23k Views**

In Spring, the *Inversion of Control (IoC)* principle is implemented using the *Dependency Injection (DI)* design pattern. Let's understand dependency injection with the help of an example. First we will see a java version of the example and later we will add spring functionalities to it. As far as the example go, its pretty simple. The *QuizMater* interface exposes the popQuestion() method. To keep things simple, our *QuizMaster* will generate only one question.

QuizMaster.java

----------------

package com.vaannila;

public interface QuizMaster {

public String popQuestion();

}

The *StrutsQuizMaster* and the *SpringQuizMaster* class implements *QuizMaster* interface and they generate questions related to struts and spring respectively.

StrutsQuizMaster.java

----------------------

package com.vaannila;

public class StrutsQuizMaster implements QuizMaster {

@Override

public String popQuestion() {

return "Are you new to Struts?";

}

}

SpringQuizMaster.java

----------------------

package com.vaannila;

public class SpringQuizMaster implements QuizMaster {

@Override

public String popQuestion() {

return "Are you new to Spring?";

}

}

We have a *QuizMasterService* class that displays the question to the user. The *QuizMasterService*class holds reference to the *QuizMaster*.

QuizMasterService.java

-----------------------

package com.vaannila;

public class QuizMasterService {

private QuizMaster quizMaster = new SpringQuizMaster();

public void askQuestion()

{

System.out.println(quizMaster.popQuestion());

}

}

Finally we create the *QuizProgram* class to conduct quiz.

QuizProgram.java

----------------

package com.vaannila;

public class QuizProgram {

public static void main(String[] args) {

QuizMasterService quizMasterService = new QuizMasterService();

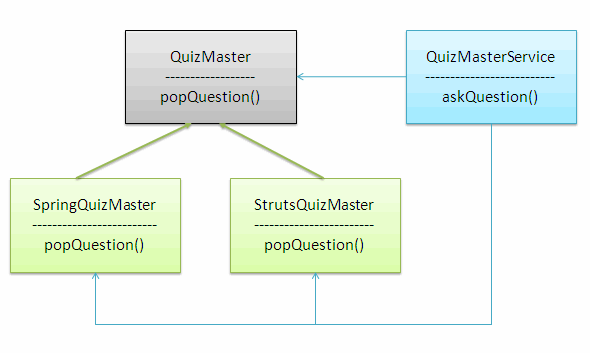
quizMasterService.askQuestion();

}

}

As you can see it is pretty simple, here we create an instance of the *QuizMasterService* class and call the *askQuestion()* method. When you run the program as expected "*Are you new to Spring?*" gets printed in the console.

Let's have a look at the class diagram of this example. The green arrows indicate generalization and the blue arrows indicates association.



As you can see this architecture is tightly coupled. We create an instance of the *QuizMaster* in the*QuizMasterService* class in the following way.

private QuizMaster quizMaster = new SpringQuizMaster();

To make our quiz master Struts genius we need to make modifications to the *QuizMasterService* class like this.

private QuizMaster quizMaster = new StrutsQuizMaster();

So it is tightly coupled. Now lets see how we can avoid this by using the *Dependency Injection* design pattern. The Spring framework provides prowerful container to manage the components. The container is based on the Inversion of Control (IoC) principle and can be implemented by using the Dependency Injection (DI) design pattern. Here the component only needs to choose a way to accept the resources and the container will deliver the resource to the components.

In this example instead of we, directly creating an object of the *QuizMaster* bean in the*QuizMasterService* class, we make use of the container to do this job for us. Instead of hard coding any values we will allow the container to inject the required dependancies.

We can inject the dependancies using the setter or constructor injection. Here we will see how we can do this using the setter injection.

QuizMasterService.java

-----------------------

package com.vaannila;

public class QuizMasterService {

QuizMaster quizMaster;

public void setQuizMaster(QuizMaster quizMaster) {

this.quizMaster = quizMaster;

}

public void askQuestion()

{

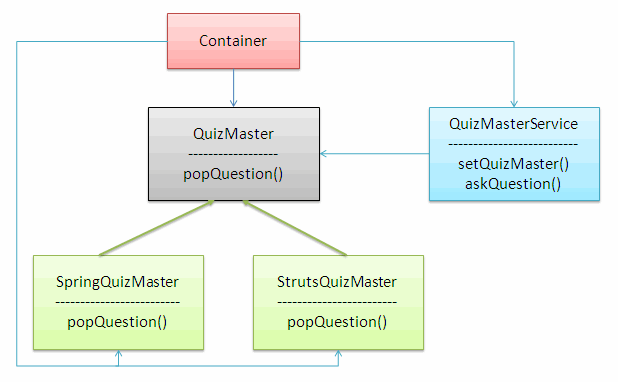
System.out.println(quizMaster.popQuestion());

}

}

The value for the *QuizMaster* will be set using the *setQuizMaster()* method. The QuizMaster object is never instantiated in the *QuizMasterService* class, but still we access it. Usually this will throw a*NullPointerException,* but here the container will instantiate the object for us, so it works fine.

After making all the changes, the class diagram of the example look like this.



The container comes into picture and it helps in injecting the dependancies.

The bean configuration is done in the *beans.xml* file.

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

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

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.xsd">

<bean id="springQuizMaster" class="com.vaannila.SpringQuizMaster"></bean>

<bean id="strutsQuizMaster" class="com.vaannila.StrutsQuizMaster"></bean>

<bean id="quizMasterService" class="com.vaannila.QuizMasterService">

<property name="quizMaster">

<ref local="springQuizMaster"/>

</property>

</bean>

</beans>

We define each bean using the *bean* tag. The *id* attribute of the bean tag gives a logical name to the bean and the *class* attribute represents the actual bean class. The *property* tag is used to refer the property of the bean. To inject a bean using the setter injection you need to use the *ref* tag.

Here a reference of *SpringQuizMaster* is injected to the *QuizMaster* bean. When we execute this example, "*Are you new to Spring?*" gets printed in the console.

To make our *QuizMaster* ask questions related to Struts, the only change we need to do is, to change the bean reference in the *ref* tag.

<bean id="quizMasterService" class="com.vaannila.QuizMasterService">

<property name="quizMaster">

<ref local="strutsQuizMaster"/>

</property>

</bean>

In this way the Dependency Injection helps in reducing the coupling between the components.

To execute this example add the following jar files to the classpath.

antlr-runtime-3.0

commons-logging-1.0.4

org.springframework.asm-3.0.0.M3

org.springframework.beans-3.0.0.M3

org.springframework.context-3.0.0.M3

org.springframework.context.support-3.0.0.M3

org.springframework.core-3.0.0.M3

org.springframework.expression-3.0.0.M3

You can download and try the DI example by clicking the download link below.

Source: [Download](https://dzone.com/sites/all/files/SpringExample2.zip)

Source + Lib: [Download](https://dzone.com/sites/all/files/SpringExample2_0.zip)