

Struts 2 Frame Work Tutorial

■ Raghupathi P

Struts 2 Framework Tutorial

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.

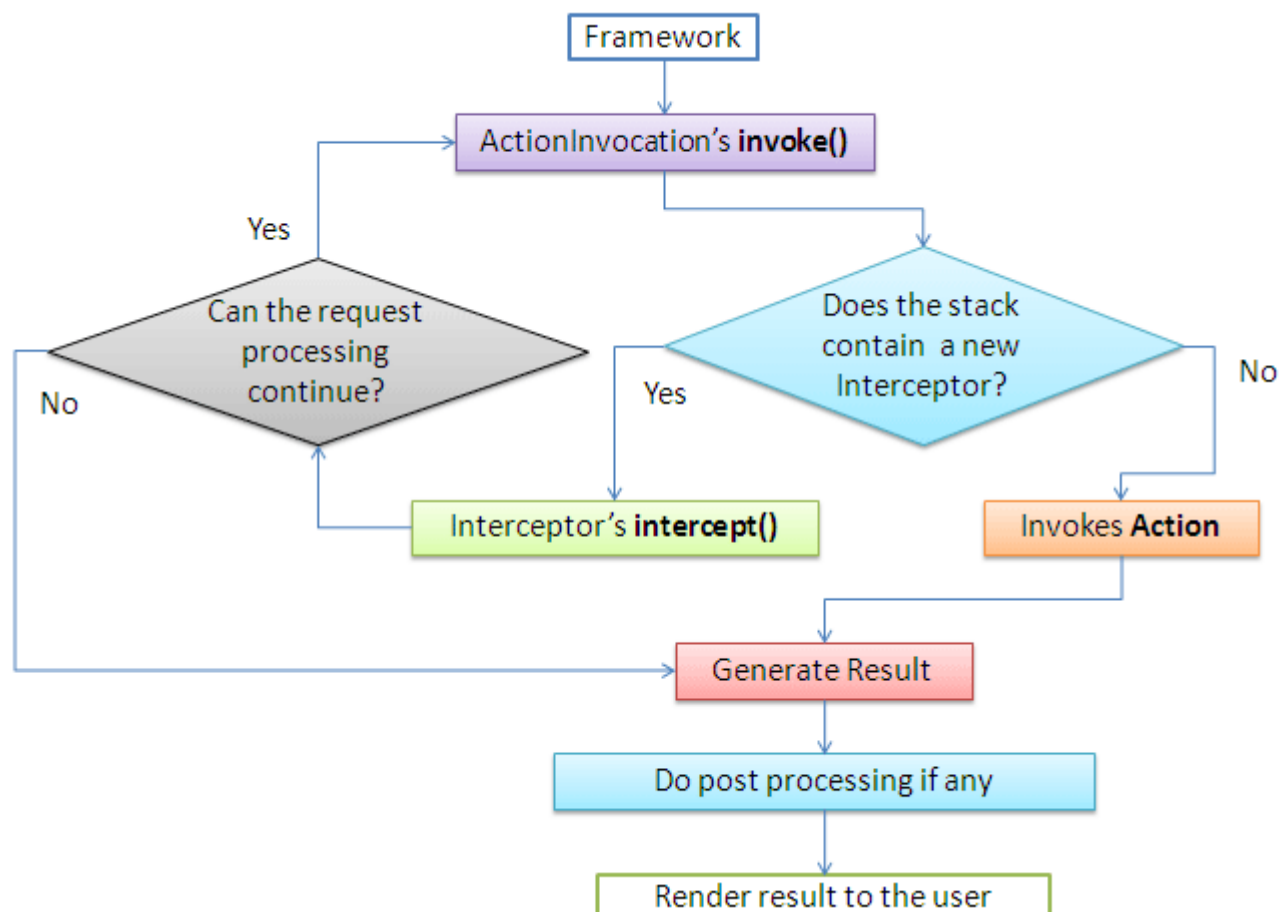
In Struts 2 FilterDispatcher does the job of Controller.

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.

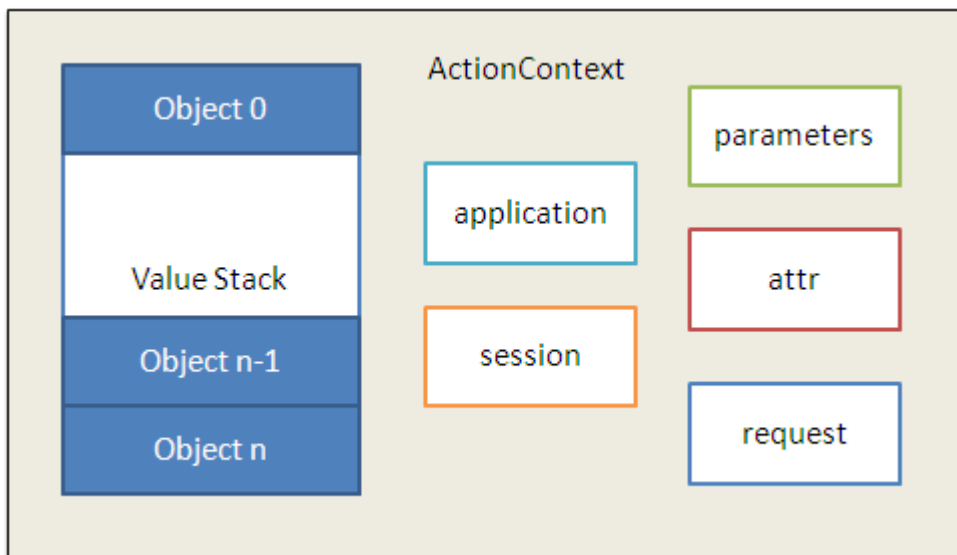
The other important features of Struts 2 are **OGNL** and **ValueStack**.

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 ApplicationContext are unique per thread, this makes the Struts 2 actions **thread safe**.

Struts 2 Annotations Example

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.

01.commons-fileupload-1.2.1

02.commons-io-1.3.2

03.commons-logging-1.1

04.freemarker-2.3.13

05.junit-3.8.1

06.ognl-2.6.11

07.spring-test-2.5.6

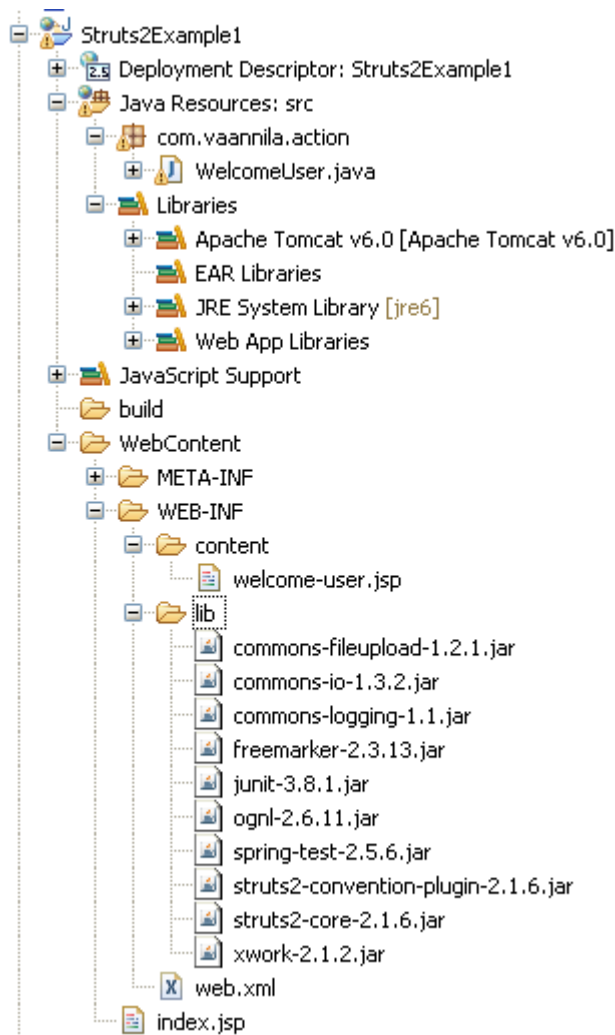
08.struts2-convention-plugin-2.1.6

09.struts2-core-2.1.6

10.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 textfield and the submit tag is used to create the submit button. The index.jsp page contains the following code.

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

```
05.<html>
```

```
06.<head>
```

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

```
08.<title>Hello World</title>
```

```
09.</head>
```

```
10.<body>
```

```
11.<s:form action="welcome-user" >
12.<s:textfield name="userName" label="User Name" />
13.<s:submit />
14.</s:form>
15.</body>
16.</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.

```
01.package com.vaannila.action;
02.
03.import com.opensymphony.xwork2.ActionSupport;
04.
05.public class WelcomeUser extends ActionSupport{
06.private String userName;
07.private String message;
08.
09.public String execute() {
10.message = "Welcome " + userName;
11.return SUCCESS;
12.}
13.
14.public void setUsername(String userName) {
15.this.userName = userName;
16.}
17.
```

```

18.public void setMessage(String message) {
19.this.message = message;
20.}

21.

22.public String getUserName() {

23.return userName;

24.}

25.

26.public String getMessage() {

27.return message;

28.}

29.}

```

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.

```

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>Welcome User</title>

08.</head>

09.<body>

10.

11.<h1>${message}</h1>

12.

```

13.</body>

14.</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.

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.su
n.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.

04.<display-name>Struts2Example1</display-name>

05.

06.<filter>

07.<filter-name>struts2</filter-name>

08.<filter-class>

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

10.</filter-class>

11.</filter>

12.

13.<filter-mapping>

14.<filter-name>struts2</filter-name>

15.<url-pattern>/*</url-pattern>

16.</filter-mapping>

17.

18.<welcome-file-list>

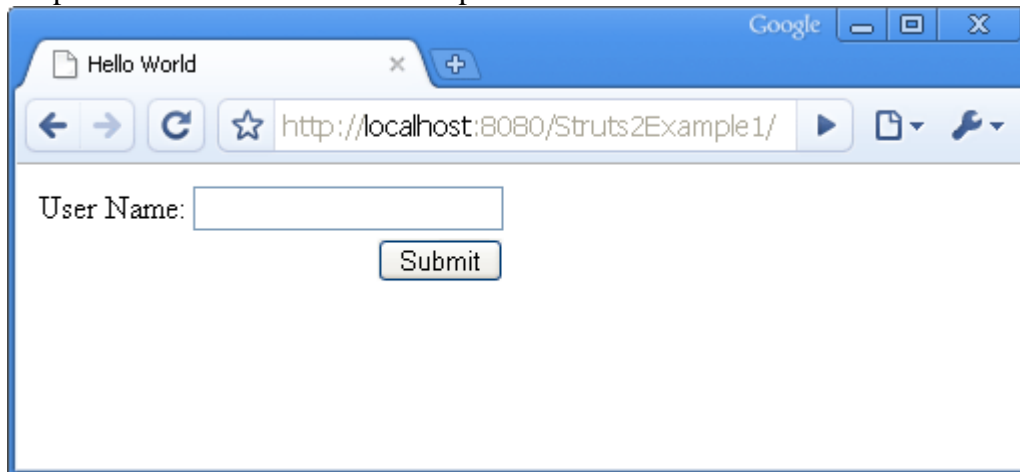
19.<welcome-file>index.jsp</welcome-file>

20.</welcome-file-list>

21.

22.</web-app>

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

A screenshot of a web browser window. The title bar says 'Hello World'. The address bar shows 'http://localhost:8080/Struts2Example1/'. The page content has a label 'User Name:' followed by a text input field. Below the input field is a 'Submit' button.

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



The example works fine. Now let's 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 in turn 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.

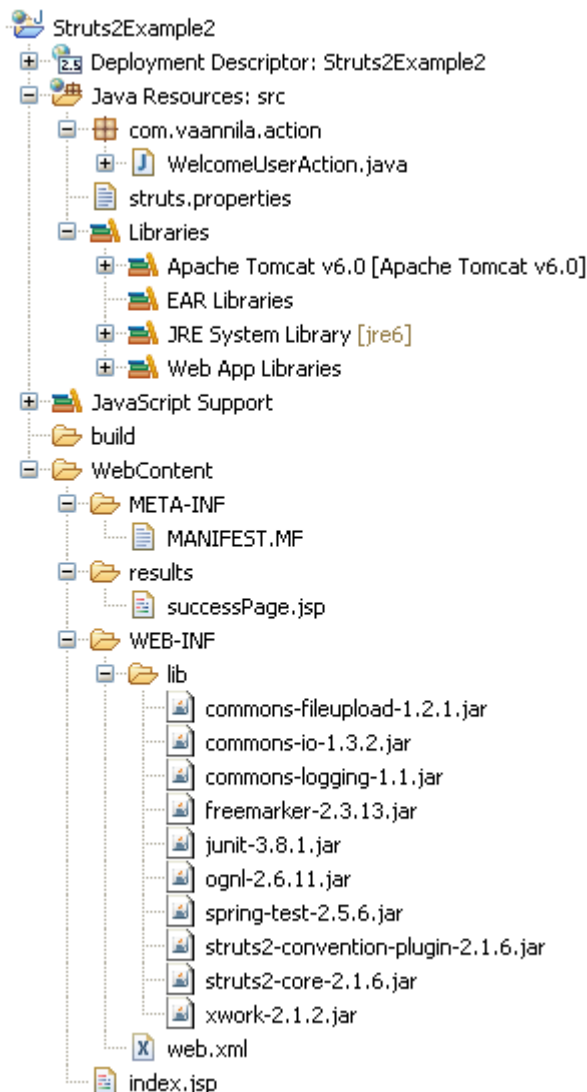
- Now the Convention plug-in knows where to look for results, but it doesn't know which file to display inside the content directory. The Convention plug-in finds this with the help of the result code returned by the Action class. If "success" is returned then the Convention plug-in will look for a file 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](#)).

Struts 2 Annotations Example

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](#)). Here we will see the same hello user example with the following changes.

- Our Action class ends with the word **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 class name ends with the word **Action**.

```
01.package com.vaannila.action;
02.
03.import org.apache.struts2.convention.annotation.Action;
04.import org.apache.struts2.convention.annotation.Result;
05.
06.public class WelcomeUserAction {
07.private String userName;
08.private String message;
09.
10.
11.@Action(value="/welcome",results={@Result(name="success",location="/results/successPage.jsp")})
12.public String execute() {
13.message = "Welcome " + userName + " !";
14.return "success";
15.}
16.
17.public void setUsername(String userName) {
18.this.userName = userName;
19.}
20.
21.public void setMessage(String message) {
22.this.message = message;
23.}
24.
25.public String getUsername() {
26.return userName;
27.}
28.
29.public String getMessage() {
30.return message;
31.}
32.}
```

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 word 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.

```
1.struts.properties file
2.-----
3.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"`.

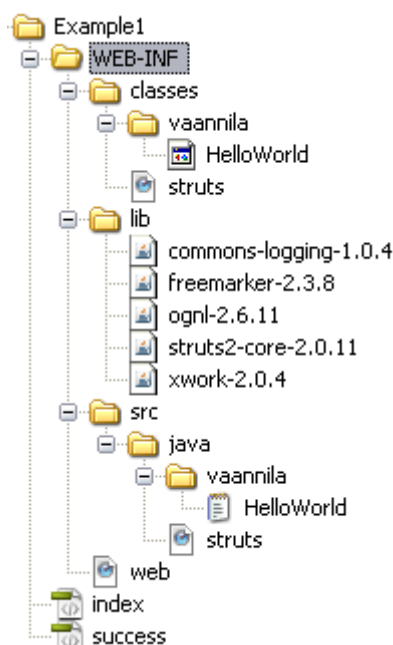
The annotations needs to be specified only when you are not using the default naming conventions, if you use them you can keep writing action classes and result pages without any configuration and the framework know exactly when to invoke them.

Hello World Application

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 application. 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.

```
01.<filter>

02.<filter-name>struts2</filter-name>

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

04.</filter>

05.<filter-mapping>

06.<filter-name>struts2</filter-name>

07.<url-pattern>/*</url-pattern>

08.</filter-mapping>

09.<welcome-file-list>

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

11.</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.

```
1.<struts>

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

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

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

5.</action>

6.</package>

7.</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.

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

02.

03.<html>

04.<head>

05.<title>Hello World</title>
```

```
06.</head>

07.<body>

08.<s:form action="HelloWorld" >

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

10.<s:submit />

11.</s:form>

12.</body>

13.</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.

```
01.public class HelloWorld {

02.

03.private String message;

04.

05.private String userName;

06.

07.public HelloWorld() {

08.}

09.

10.public String execute() {

11.setMessage("Hello " + getUserName());

12.return "SUCCESS";

13.}

14.

15.public String getMessage() {

16.return message;

17.}

18.}
```

```

19.public void setMessage(String message) {
20.this.message = message;
21.}
22.
23.public String getUser_name() {
24.return userName;
25.}
26.
27.public void setUser_name(String userName) {
28.this.userName = userName;
29.}
30.
31.}

```

In the execute() method of the HelloWorld action we compose the message to be displayed. Note we need not have a separate 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

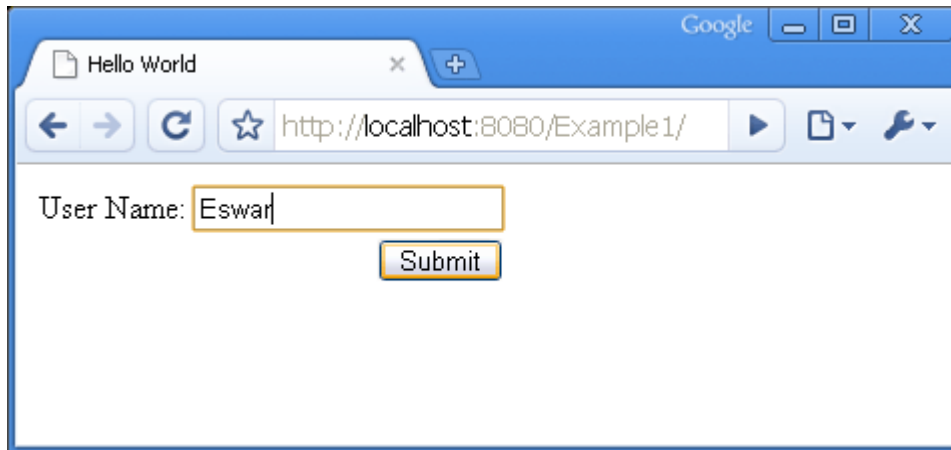
In the success page we display the "Hello User" message using the property tag.

```

01.<%@taglib uri="/struts-tags" prefix="s" %>
02.<html>
03.<head>
04.<title>Hello World</title>
05.</head>
06.<body>
07.<h1><s:property value="message" /></h1>
08.</body>
09.</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.



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 will 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 retrieve the values back in the jsp page.

The register.jsp looks like this

Register Page

File Edit View History Bookmarks Window Help

http://localhost:8080/Struts2Ex Google

User Name:

Password:

Gender: ☐ Male ☐ Female

Select a country:

About You:

Community: ☒ Java ☒ .Net ☐ SOA

☒ Would you like to join our mailing list?

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

```
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.<%@taglib uri="/struts-tags" prefix="s"%>
05.<html>
06.<head>
07.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
08.<title>Register Page</title>
09.</head>
10.<body>
11.<s:form action="Register">
12.<s:textfield name="userName" label="User Name" />
13.<s:password name="password" label="Password" />
14.<s:radio name="gender" label="Gender" list="{ 'Male', 'Female' }" />
15.<s:select name="country" list="countryList" listKey="countryId" listValue=
"countryName" headerKey="0" headerValue="Country" label="Select a country" />
```

```

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

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

18.<s:checkbox name="mailingList"

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

20.<s:submit />

21.</s:form>

22.</body>

23.</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 let's 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"

```
1.<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 through 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.

```
1.<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 separate 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 separately. 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 let's see the flow of the example. First the index.jsp page will be invoked by the framework.

```
1.index.jsp
```

```
2.~~~~~
```

```
3.<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.

```

01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.<struts>
06.<package name="default" extends="struts-default">
07.<action name="*Register" method="{1}" class="vaannila.RegisterAction">
08.<result name="populate">/register.jsp</result>
09.<result name="input">/register.jsp</result>
10.<result name="success">/success.jsp</result>
11.</action>
12.</package>
13.</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.

```

001.package vaannila;
002.
003.import java.util.ArrayList;
004.
005.import com.opensymphony.xwork2.ActionSupport;
006.
007.public class RegisterAction extends ActionSupport {
008.
009.private String userName;
010.
011.private String password;
012.

```

```
013.private String gender;

014.

015.private String about;

016.

017.private String country;

018.

019.private ArrayList<Country> countryList;

020.

021.private String[] community;

022.

023.private ArrayList<String> communityList;

024.

025.private Boolean mailingList;

026.

027.public String populate() {

028.

029.countryList = new ArrayList<Country>();

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

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

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

033.

034.communityList = new ArrayList<String>();

035.communityList.add("Java");

036.communityList.add(".Net");

037.communityList.add("SOA");

038.

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

```
040.mailingList = true;

041.

042.return "populate";

043.}

044.

045.public String execute() {

046.return SUCCESS;

047.}

048.

049.public String getUsername() {

050.return userName;

051.}

052.

053.public void setUsername(String userName) {

054.this.userName = userName;

055.}

056.

057.public String getPassword() {

058.return password;

059.}

060.

061.public void setPassword(String password) {

062.this.password = password;

063.}

064.

065.public String getGender() {

066.return gender;

067.}
```

```
068.

069.public void setGender(String gender) {
070.this.gender = gender;
071.}

072.

073.public String getAbout() {
074.return about;
075.}

076.

077.public void setAbout(String about) {
078.this.about = about;
079.}

080.

081.public String getCountry() {
082.return country;
083.}

084.

085.public void setCountry(String country) {
086.this.country = country;
087.}

088.

089.public ArrayList<Country> getCountryList() {
090.return countryList;
091.}

092.

093.public void setCountryList(ArrayList<Country> countryList) {
094.this.countryList = countryList;
095.}
```

```
096.

097.public String[] getCommunity() {

098.return community;

099.}

100.

101.public void setCommunity(String[] community) {

102.this.community = community;

103.}

104.

105.public ArrayList<String> getCommunityList() {

106.return communityList;

107.}

108.

109.public void setCommunityList(ArrayList<String> communityList) {

110.this.communityList = communityList;

111.}

112.

113.public Boolean getMailingList() {

114.return mailingList;

115.}

116.

117.public void setMailingList(Boolean mailingList) {

118.this.mailingList = mailingList;

119.}

120.

121.}
```

Textfield and Password Tags

Now lets see each UI tag in detail. The textfield 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.

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

```
2.<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.

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

Select Tag

We display 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 contains values of type Country. The Country class has countryId and countryName attribute. The countryName holds the country value to be displayed 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.

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

```
2.label="Select a country" />
```

Textarea Tag

The textarea tag is used to create a textarea.

```
1.<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.

```
1.<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.

```
1.<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

```
1.<s:submit />
```

Now let's 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.


```
01.success.jsp
02.-----
03.<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
04.pageEncoding="ISO-8859-1"%>
05.<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
06.<%@taglib uri="/struts-tags" prefix="s"%>
07.<html>
08.<head>
09.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10.<title>Details Page</title>
11.</head>
12.<body>
13.User Name: <s:property value="userName" /><br>
14.Gender: <s:property value="gender" /><br>
15.Country: <s:property value="country" /><br>
16.About You: <s:property value="about" /><br>
17.Community: <s:property value="community" /><br>
18.Mailing List: <s:property value="mailingList" />
19.</body>
20.</html>
```

Now lets enter the following details and submit the form.

Register Page

File Edit View History Bookmarks Window Help

http://localhost:8080/Struts2Exa Google

User Name:

Password:

Gender: ☒ Male ☐ Female

Select a country:

About You:

Community: ☒ Java ☐ .Net ☐ SOA

☐ Would you like to join our mailing list?

Submit

The following registration details will be displayed to the user.

Details Page

File Edit View History Bookmarks Window Help

http://localhost:8080/Struts2Ex Google

User Name: Eswar

Gender: Male

Country: 1

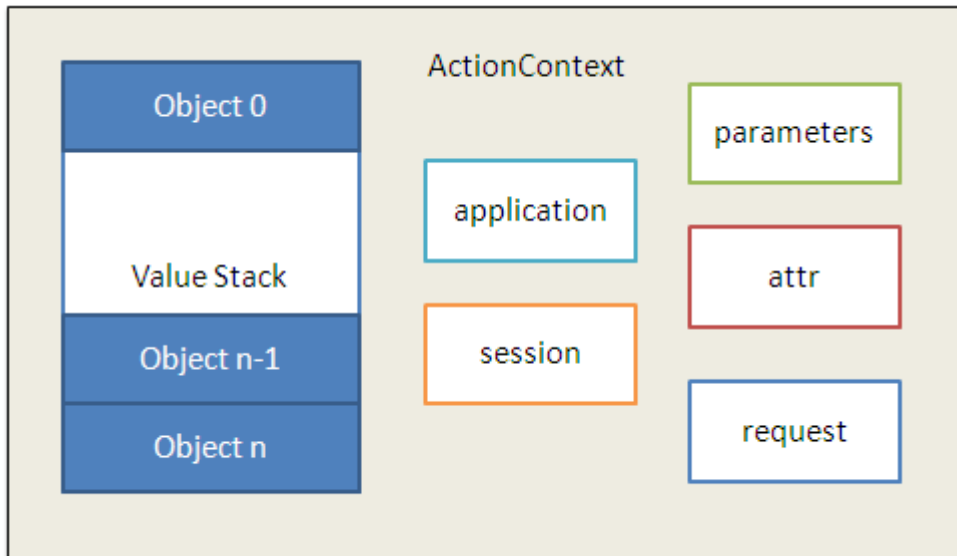
About You: Gone crazy nowadays.

Community: Java

Mailing List: false

Struts 2 Data Tags Example

In this example you will learn how to use the property tag, the set tag and the push tag. These tags are part of the Struts 2 Data Tags. Before we see the syntax of each tag you need to know what an ActionContext and a ValueStack is.



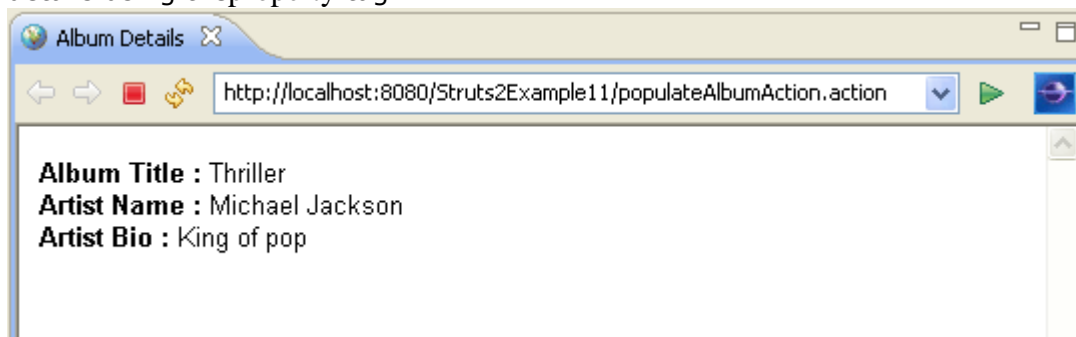
The ActionContext is a global storage area that holds all the data associated with the processing of a request.

The ActionContext is thread local this makes the Struts 2 actions thread safe.

The ValueStack is the part of the ActionContext. In Struts 2 actions resides on the ValueStack.

The Property Tag

The property tag is used to retrieve the data from the ValueStack or some other object in the ActionContext like application or session. Let's see how to display the following details using the property tag.



Our Action class AlbumInfoAction contains the following piece of code.

```
01.package vaannila;
02.
03.public class AlbumInfoAction{
04.
05.private String title;
06.private Artist artist;
07.
08.public String populate()
09.{
10.title = "Thriller";
11.artist = new Artist("Michael Jackson","King of pop");
12.return "populate";
13.}
```

```
14.
15.public String execute()
16.{
17.return "success";
18.}
19.
20.public String getTitle() {
21.return title;
22.}
23.public void setTitle(String title) {
24.this.title = title;
25.}
26.public Artist getArtist() {
27.return artist;
28.}
29.public void setArtist(Artist artist) {
30.this.artist = artist;
31.}
32.
33.}
```

Struts 2 Property Tag Example

Our Artist data class contains the following code.

```
01.package vaannila;

02.

03.public class Artist {

04.

05.private String name;

06.private String bio;

07.

08.Artist(String name, String bio)

09.{

10.this.name = name;

11.this.bio = bio;

12.}

13.public String getName() {

14.return name;

15.}
```

```
16.public void setName(String name) {  
17.this.name = name;  
18.}  
  
19.public String getBio() {  
20.return bio;  
21.}  
  
22.public void setBio(String bio) {  
23.this.bio = bio;  
24.}  
  
25.  
26.}
```

Let's see how we can access the action class attributes using the property tag in the jsp page. The albumDetails.jsp page contains the following code.

```
01.<%@taglib uri="/struts-tags" prefix="s"%>  
  
02.<html>  
  
03.<head>  
  
04.<s:head />  
  
05.<style type="text/css">  
06.@import url(style.css);  
07.</style>  
  
08.<title>Album Details</title>  
09.</head>  
  
10.<body>  
  
11.<div class="content">  
12.<b>Album Title :</b>  
13.<s:property value="title" /> <br>  
14.<b>Artist Name :</b>  
15.<s:property value="artist.name" />  
16.<br>
```

```

17.<b>Artist Bio :</b>

18.<s:property value="artist.bio" />

19.<br>

20.</div>

21.</body>

22.</html>

```

As you can see title is the property of the AlbumInfoAction so we can access it directly. But name and bio are properties of the Artist class so to access them we need to go one step deeper. We need to use a second-level OGNL expression to access them.

Struts 2 Set Tag Example

The `set` tag is used to assign a property value to another name. This helps in accessing the property in a faster and easier way. To access the artist name we need to go one level deeper and fetch it when we used the `property` tag, instead you can assign the value **to another property in the ActionContext and access it directly**. The following code shows how to do this.

```

1.<s:set name="artistName" value="artist.name" />

2.<s:set name="artistBio" value="artist.bio" />

3.<b>Album Title :</b> <s:property value="title" /> <br>

4.<b>Artist Name :</b> <s:property value="#artistName" /> <br>

5.<b>Artist Bio :</b> <s:property value="#artistBio" /> <br>

```

The property `artistName` and `artistBio` will now be stored in the `ActionContext`. To refer then you need to use the following syntax **#objectName**.

You can also place the property value in the session map in the following way. Now the value `artistName` and `artistBio` will persist throughout the session.

```

1.<s:set name="artistName" value="artist.name" scope="session" />

2.<s:set name="artistBio" value="artist.bio" scope="session" />

3.<b>Album Title :</b> <s:property value="title" /> <br>

4.<b>Artist Name :</b> <s:property value="#session['artistName']" /> <br>

5.<b>Artist Bio :</b> <s:property value="#session['artistBio']" /> <br>

```

In the same way you can also store the values in other maps available in the `ActionContext`.

Struts 2 Push Tag Example

You can **push a value into the ValueStack** using the `push` tag. The value we pushed using push tag will be on top of the ValueStack, so it can be easily referenced using the first-level OGNL expression instead of a deeper reference. The following code show how to do this.

```
1.<b>Album Title :</b> <s:property value="title" /> <br>

2.<s:push value="artist">

3.<b>Artist Name :</b> <s:property value="name" /> <br>

4.<b>Artist Bio :</b> <s:property value="bio" /> <br>

5.</s:push>
```

Struts 2 Bean Tag Example

We will see how the bean tag works using a currency converter example. In this example we will convert dollars to rupees. We do this using the `CurrencyConverter` JavaBeans class.

The `CurrencyConverter` class.

```
01.package vaannila;

02.

03.public class CurrencyConverter {

04.

05.private float rupees;

06.private float dollars;

07.

08.public float getRupees() {

09.return dollars * 50;

10.}

11.public void setRupees(float rupees) {

12.this.rupees = rupees;

13.}

14.public float getDollars() {

15.return rupees/50 ;
```

```

16.}

17.public void setDollars(float dollars) {

18.this.dollars = dollars;

19.}

20.

21.}

```

The next step is to create an instance of the `CurrencyConverter` bean in the jsp page using the bean tag. We can either use the bean tag to push the value onto the `ValueStack` or we can set a top-level reference to it in the `ActionContext`. Let's see one by one.

First we will see how we can do this by pushing the value onto the `ValueStack`. The `index.jsp` page contains the following code.

```

01.<html>

02.<head>

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

04.<title>Bean Tag Example</title>

05.</head>

06.<body>

07.<s:bean name="vaannila.CurrencyConverter">

08.<s:param name="dollars" value="100" />

09.100 Dollars = <s:property value="rupees" /> Rupees

10.</s:bean>

11.

12.</body>

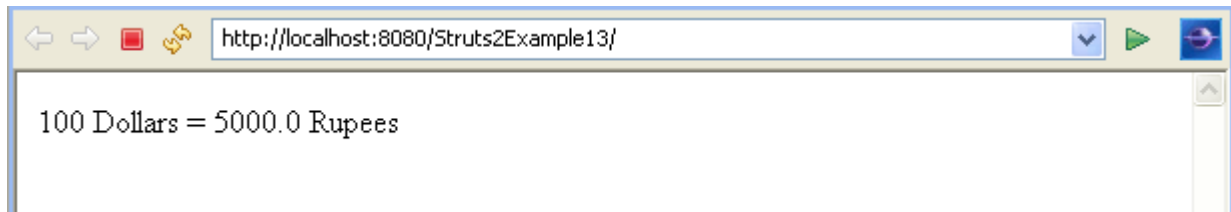
13.</html>

```

The `name` attribute of the bean tag hold the fully qualified class name of the `JavaBean`. The `param` tag is used to set the dollar value. We now use the `property` tag to retrieve the equivalent value in rupees.

The `CurrencyConverter` bean will reside on the `ValueStack` till the duration of the bean tag. So it's important that we use the property tag within the bean tag.

When you execute the example the following page is displayed.



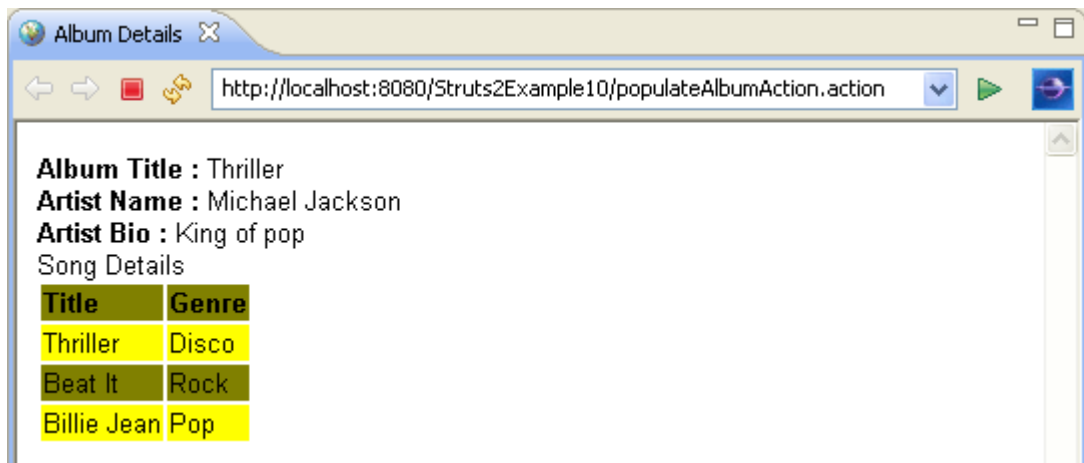
Now we will see how we can do the same by creating a top-level reference to the bean in the `ActionContext`. We do this using the following code.

```
01.<html>
02.<head>
03.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
04.<title>Bean Tag Example</title>
05.</head>
06.<body>
07.<s:bean name="vaannila.CurrencyConverter" var="converter">
08.<s:param name="dollars" value="100"></s:param>
09.</s:bean>
10.100 Dollars = <s:property value="#converter.rupees" /> Rupees
11.
12.</body>
13.</html>
```

To create an instance of the bean in the `ActionContext` we need to specify the instance name using the `var` attribute of the bean tag. Here our instance name is `converter`. Once we do this we can access the bean values outside the bean tag. Since the value is in the `ActionContext` we use the `#` operator to refer it.

Struts 2 Control Tags Example

In this example you will see how display the following details using the Struts 2 iterator tag and the if and else tags.



In our `AlbumInfoAction` class we populate the artist and the song details using the following code.

```
01.package vaannila;
```

```
02.
```

```
03.import java.util.ArrayList;
```

```
04.import java.util.List;
```

```
05.
```

```
06.
```

```
07.public class AlbumInfoAction{
```

```
08.
```

```
09.private String title;
```

```
10.private Artist artist;
```

```
11.private static List<Song> songs = new ArrayList<Song>();
```

```
12.
```

```
13.static {
```

```
14.songs.add(new Song("Thriller","Disco"));
```

```
15.songs.add(new Song("Beat It","Rock"));
```

```
16.songs.add(new Song("Billie Jean","Pop"));
```

```
17.}
```

```
18.
```

```
19.public String populate()
```

```
20.{
21.title = "Thriller";
22.artist = new Artist("Michael Jackson","King of pop");
23.return "populate";
24.}
25.
26.public String execute()
27.{
28.return "success";
29.}
30.
31.public String getTitle() {
32.return title;
33.}
34.public void setTitle(String title) {
35.this.title = title;
36.}
37.public Artist getArtist() {
38.return artist;
39.}
40.public void setArtist(Artist artist) {
41.this.artist = artist;
42.}
43.
44.public List<Song> getSongs() {
45.return songs;
46.}
47.
```

```
48.}
```

The song class contains the title and the genre attributes.

```
01.package vaannila;
```

```
02.
```

```
03.public class Song {
```

```
04.
```

```
05.private String title;
```

```
06.private String genre;
```

```
07.
```

```
08.Song(String title, String genre)
```

```
09.{
```

```
10.this.title = title;
```

```
11.this.genre = genre;
```

```
12.}
```

```
13.public String getTitle() {
```

```
14.return title;
```

```
15.}
```

```
16.public void setTitle(String title) {
```

```
17.this.title = title;
```

```
18.}
```

```
19.public String getGenre() {
```

```
20.return genre;
```

```
21.}
```

```
22.public void setGenre(String genre) {
```

```
23.this.genre = genre;
```

```
24.}
```

```
25.}
```

Struts 2 Iterator Tag Example

In Struts 2 the iterator tag is used to loop over a collection of objects. The iterator tag can iterate over any Collection, Map, Enumeration, Iterator, or array.

```
01.<table class="songTable">
02.<tr class="even">
03.<td><b>Title</b></td>
04.<td><b>Genre</b></td>
05.</tr>
06.<s:iterator value="songs" status="songStatus">
07.<tr
08.class="<s:if test="#songStatus.odd == true ">odd</s:if><s:else>even</s:else>">
09.<td><s:property value="title" /></td>
10.<td><s:property value="genre" /></td>
11.</tr>
12.</s:iterator>
13.</table>
```

We use the iterator tag to iterate over the collection of songs. Here the Song property is of type ArrayList. To know more about the iterator status we can create an instance of the `IteratorStatus` object by specifying a value to the status attribute of the iterator tag. The newly created instance will be placed in the ActionContext which can be referred using the following OGNL expression `#statusName`.

The table shows the different properties of the `IteratorStatus` object.

Name	Return Type	Description
index	int	zero-based index value.
count	int	index + 1
first	boolean	returns true if it is the first element
last	boolean	returns true if it is the last element

even	boolean	returns true if the count is an even number.
odd	boolean	returns true if the count is an odd number.
modulus	int	takes an int value and returns the modulus of count.

Struts 2 If and Else Tags Example

We use the if and else tags to highlight the even and odd rows in different colors. We use the `IteratorStatus` class methods to find whether the row is even or odd. The following code shows how to do this.

```

1.<s:iterator value="songs" status="songStatus">

2.<tr

3.class="<s:if test="#songStatus.odd == true
">odd</s:if><s:else>even</s:else>">

4.<td><s:property value="title" /></td>

5.<td><s:property value="genre" /></td>

6.</tr>

7.</s:iterator>

```

The `elseif` tag is also available. You can use it if there are multiple conditions to check.

Struts 2 OGNL Expression Language Example

In this example you will learn the different syntaxes for using the **Object-Graph Navigation Language (OGNL)**. OGNL expression language is simple and powerful. OGNL expression language helps in accessing the values stored on the `ValueStack` and in the `ActionContext`.

First let's see how to access an array of String variables using OGNL. In the action class we create a string array and initialize it as shown below.

```

01.package vaannila;

02.

03.public class SampleAction {

04.

05.private String[] sampleArray;

06.{

07.sampleArray = new String[]{"item1","item2","item3"};

08.}

09.public String execute()

```

```

10.{
11.return "success";
12.}

13.public String[] getSampleArray() {
14.return sampleArray;
15.}

16.public void setSampleArray(String[] sampleArray) {
17.this.sampleArray = sampleArray;
18.}
19.}

```

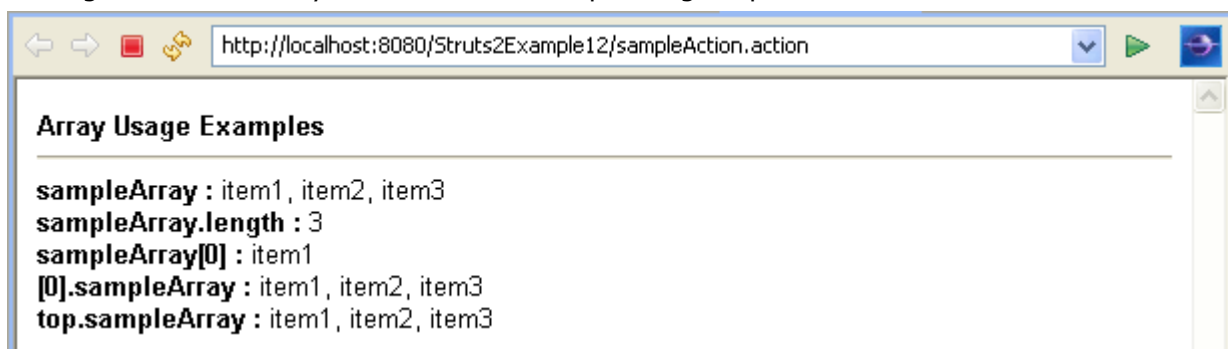
You can access the array values in the jsp page using the OGNL expression language in the following way.

```

1.<b>Array Usage Examples</b>
2.<br><hr>
3.<b>sampleArray :</b> <s:property value="sampleArray"/> <br>
4.<b>sampleArray.length :</b> <s:property value="sampleArray.length"/> <br>
5.<b>sampleArray[0] :</b> <s:property value="sampleArray[0]"/> <br>
6.<b>[0].sampleArray :</b> <s:property value="[0].sampleArray"/> <br>
7.<b>top.sampleArray :</b> <s:property value="top.sampleArray"/> <br>

```

The figure shows the syntax and the corresponding output.



Since our object is on top of the ValueStack we can access it using [0] notation. If our object was in the second position from the top, we will access it using the [1] notation.

We can also do this using the `top` keyword. `top` returns the value on top of the ValueStack.

Now let's see how to access an ArrayList using the OGNL expression language. In the action class we create and initialize the ArrayList as shown below.

```

01.package vaannila;

02.

03.import java.util.ArrayList;

04.import java.util.List;

05.

06.public class SampleAction {

07.

08.private List<String> sampleList = new ArrayList<String>();

09.{

10.sampleList.add("listItem1");

11.sampleList.add("listItem2");

12.sampleList.add("listItem3");

13.}

14.public String execute()

15.{

16.return "success";

17.}

18.public List<String> getSampleList() {

19.return sampleList;

20.}

21.public void setSampleList(List<String> sampleList) {

22.this.sampleList = sampleList;

23.}

24.}

```

You can access the ArrayList values in the jsp page using the following syntax.

1.List Usage Examples

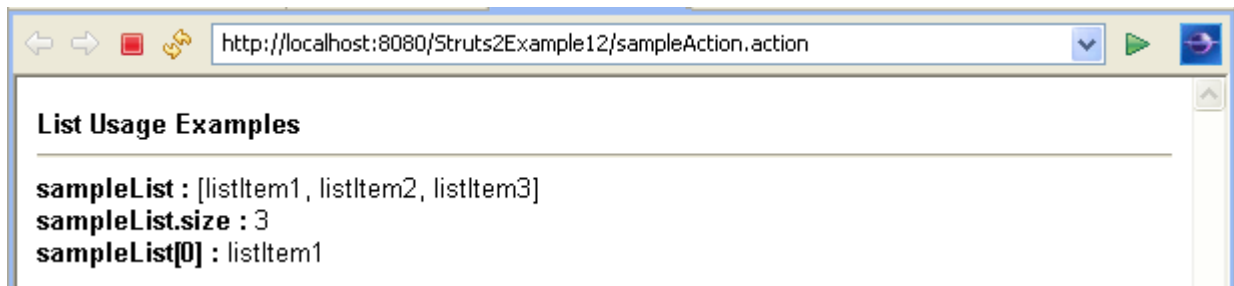
2.
<hr>

3.sampleList : <s:property value="sampleList"/>

4.sampleList.size : <s:property value="sampleList.size"/>

5.sampleList[0] : <s:property value="sampleList[0]"/>

The figure shows the syntax and the corresponding output.



Now let's see how to access a Map using the OGNL expression language. In the action class we create and initialize the HashMap as shown below.

```
01.package vaannila;

02.

03.import java.util.HashMap;

04.import java.util.Map;

05.

06.public class SampleAction {

07.

08.private Map<Integer,String> sampleMap = new HashMap<Integer,String>();

09.private String carMake;

10.{

11.sampleMap.put(new Integer(1), "one");

12.sampleMap.put(new Integer(2), "two");

13.sampleMap.put(new Integer(3), "three");

14.}

15.public String execute()

16.{

17.return "success";

18.}

19.public Map<Integer, String> getSampleMap() {
```

```

20.return sampleMap;
21.}

22.public void setSampleMap(Map<Integer, String> sampleMap) {
23.this.sampleMap = sampleMap;
24.}

25.public String getCarMake() {
26.return carMake;
27.}

28.public void setCarMake(String carMake) {
29.this.carMake = carMake;
30.}

31.
32.}

```

You can access the Map values in the jsp page using the following OGNL expression language syntax.

```

1.<b>Map Usage Examples</b>
2.<br><hr>
3.<b>sampleMap[1] :</b> <s:property value="sampleMap[1]"/> <br>
4.<b>sampleMap.size :</b> <s:property value="sampleMap.size"/> <br>

```

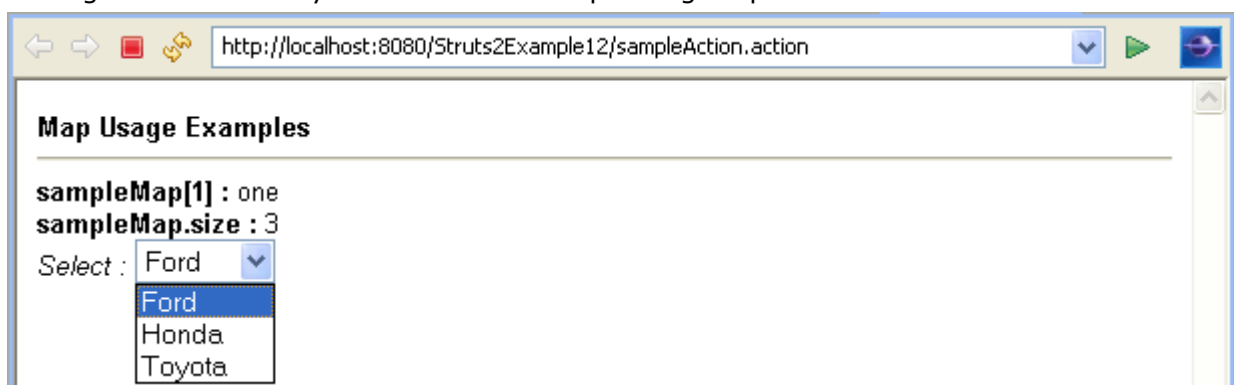
You can also create a map in the jsp page using the following syntax.

```

1.<s:select list="#{'make1':'Ford', 'make2':'Honda',
'make3':'Toyota'}"name="carMake" label="Select "></s:select>

```

The figure shows the syntax and the corresponding output.



Now let's see how to access the `name` property of the `User` object in the Action class using the OGNL expression language. The `SampleAction` contains the following code.

```

01.package vaannila;
02.
03.public class SampleAction {
04.
05.private User user = new User();
06.{
07.user.setName("Eswar");
08.}
09.public String execute()
10.{
11.return "success";
12.}
13.
14.public String getQuote()
15.{
16.return "Don't think, just do";
17.}
18.public User getUser() {
19.return user;
20.}
21.public void setUser(User user) {
22.this.user = user;
23.}
24.
25.}

```

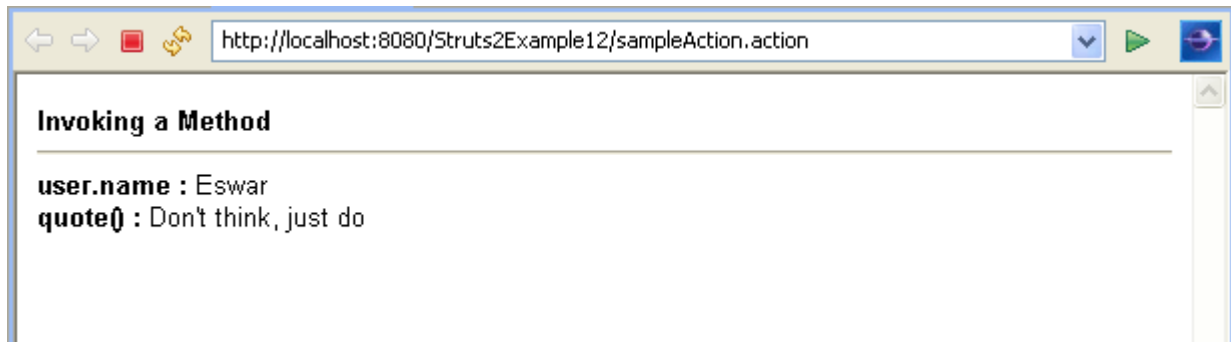
You need to use the second-level OGNL expression language to access the user name property.

```
1.<b>user.name :</b> <s:property value="user.name"/> <br>
```

You can also invoke a method in the action class in the following way. This will invoke the `quote()` method in the action class.

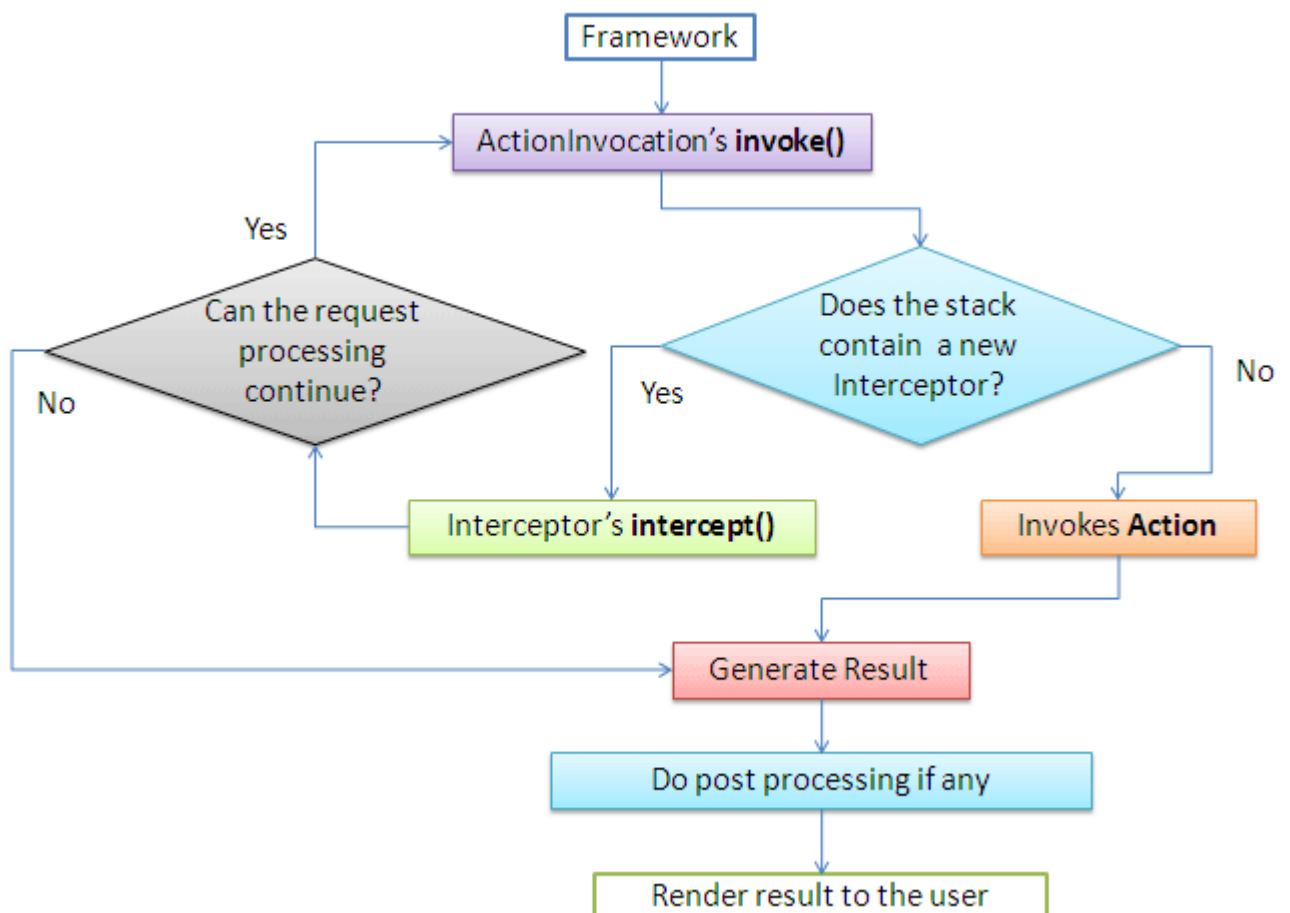
1.quote() : <s:property value="quote()" />

The figure shows the syntax and the corresponding output.



Struts 2 Interceptors Example

In this example you will see how the interceptors are invoked both before and after the execution of the action and how the results are rendered back to the user. Let's understand this with the help of the following diagram.



The following actions happen in a sequence when a request comes to the Struts 2 framework.

Framework first finds which Action class to invoke for this request and discovers the interceptors associated with the action mapping.

Now the Framework creates an instance of ActionInvocation and calls its invoke() method. At this point the Framework hands the control over to the ActionInvocation for further processing of the request.

ActionInvocation is the one which encapsulates the action and the associated interceptors. ActionInvocation knows in which sequence the interceptors should be invoked.

ActionInvocation now invokes the intercept() method of the first interceptor in the stack. We will understand this with the help of an example. Our example is very simple it uses only one interceptor for logging details.

The LoggingInterceptor's intercept() method contains the following code.

```
01.public String intercept(ActionInvocation invocation) throws Exception
02.{
03.//Pre processing
04.logMessage(invocation, START_MESSAGE);
05.
06.String result = invocation.invoke();
07.
08.//Post processing
09.logMessage(invocation, FINISH_MESSAGE);
10.
11.return result;
12.}
```

As you can see, first the logMessage() method is called and the message is logged, this is the pre processing done by the logger interceptor, then the invoke() method of the ActionInvocation is again called, this time the ActionInvocation will call the next interceptor in the stack and this cycle will continue till the last interceptor in the stack.

After the execution of all the interceptors the action class will be invoked. Finally a result string will be returned and the corresponding view will be rendered. This is the normal flow of events.

But what if an validation error occurs, in this case the request processing will be stopped. No further interceptors will be invoked. Action will not be executed. The control flow changes, now the interceptors executed so far will be invoked in the reverse order to do the post processing if any and finally the result will be rendered to the user.

Let's come back to the normal flow. In our case the logger interceptor is the only interceptor in the stack, so after logging the "START_MESSAGE", the ActionInvocation's invoke() method will invoke the action. Our action simply returns "success", then again the logger interceptor will be invoked to do the post processing, this time the "FINISH_MESSAGE" is logged and the result is returned. Based on the result the corresponding view will be rendered to the user.

We get the following benefits by using the interceptors.

- Extremely flexible.
- Cleaner and focused Action classes.
- Provides code readability and code reuse.
- Testing process becomes easier.

We can add only the interceptors we need to the stack and customising the action processing for each request.

Now let's see the flow of the example. In the index.jsp page we forward the request to the "TestLogger" URL.

```
1.<META HTTP-EQUIV="Refresh" CONTENT="0;URL=TestLogger.action">
```

The TestLogger URL is mapped to the TestLoggerAction class in the struts.xml file.

```
01.<!DOCTYPE struts PUBLIC
```

```
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
```

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

```
04.
```

```
05.<struts>
```

```
06.<package name="test" extends="struts-default">
```

```
07.<action name="TestLogger" class="vaannila.TestLoggerAction">
```

```
08.<interceptor-ref name="logger" />
```

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

```
10.</action>
```

```
11.</package>
```

```
12.</struts>
```

The interceptor-ref element is used to add a interceptor reference to the action. All the interceptors are defined in the struts-default package of the struts-default.xml file.

Now the execute() method of the TestLoggerAction class will be invoked. The execute() method just prints a statement and returns success.

```
01.package vaannila;
```

```
02.
```

```
03.public class TestLoggerAction {
```

```

04.
05.public String execute()
06.{
07.System.out.println("Inside Action");
08.return "success";
09.}
10.}

```

Based on the mapping in the XML configuration file the user will be forwarded to the success page.

The following log messages are logged in the console.

```

1.INFO: Starting execution stack for action //TestLogger
2.Inside Action

```

```

3.INFO: Finishing execution stack for action //TestLogger

```

Struts 2 Interceptors Tutorial

In this tutorial you will see different ways to create you own interceptor stack and associate it with the action class.

Struts 2 comes with a set of pre defined interceptors and interceptor stacks which you can use out of the box. The `struts-default.xml` file contains the `struts-default` package which defines all the interceptors and the interceptor stacks. You can use the stack that meets your need.

When you extend your package from the `struts-default` package by default the `defaultStack` will be used for all the actions in your package. This is configured in the `struts-default.xml` file in the following way.

```

1.<default-interceptor-ref name="defaultStack"/>

```

Let's now create our own interceptor stack. The `interceptor-stack` element is used to create an interceptor stack. A stack contains a group of interceptors. Each interceptor in the stack is defined using the `interceptor-ref` element. In this example we will create a stack similar to the `defaultStack` and customise the validation interceptor according to our need.

We have three methods in our `SampleAction` class, `populate()`, `execute()` and `validate()`. Since we extend our class from `ActionSupport` which inturn implements the `Validateable` interface, the `validate()` method of the action class will be called by the workflow interceptor. By default the `validate()` method will be called during the execution of both `populate()` and `execute()` methods but we need to validate only when the `execute()` method is invoked.

We do this by specifying the `populate` method in the `excludeMethods` parameter of the validation interceptor.

The `struts.xml` file contains the following code.

```

01.<!DOCTYPE struts PUBLIC

```

```
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.<struts>
06.<package name="default" extends="struts-default">
07.<interceptors>
08.<interceptor-stack name="exampleStack">
09.<interceptor-ref name="exception" />
10.<interceptor-ref name="alias" />
11.<interceptor-ref name="servletConfig" />
12.<interceptor-ref name="prepare" />
13.<interceptor-ref name="i18n" />
14.<interceptor-ref name="chain" />
15.<interceptor-ref name="debugging" />
16.<interceptor-ref name="profiling" />
17.<interceptor-ref name="scopedModelDriven" />
18.<interceptor-ref name="modelDriven" />
19.<interceptor-ref name="fileUpload" />
20.<interceptor-ref name="checkbox" />
21.<interceptor-ref name="staticParams" />
22.<interceptor-ref name="actionMappingParams" />
23.<interceptor-ref name="params">
24.<param name="excludeParams"> dojo\...*,^struts\...*</param>
25.</interceptor-ref>
26.<interceptor-ref name="conversionError" />
```



```

27.<interceptor-ref name="validation">

28.<param name="excludeMethods">populate</param>

29.</interceptor-ref>

30.<interceptor-ref name="workflow">

31.<param name="excludeMethods"> input,back, cancel,browse</param>

32.</interceptor-ref>

33.</interceptor-stack>

34.</interceptors>

35.<action name="*Sample" method="{1}" class="vaannila.SampleAction">

36.<interceptor-ref name="exampleStack" />

37.<result name="populate">/first.jsp</result>

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

39.</action>

40.</package>

41.</struts>

```

If you see our `exampleStack` the only change that we have done is, we have changed the `excludeMethods` of the validation interceptor, rest all is similar to the `defaultStack`. This is just to show you how to create your own interceptor stack, you can also achieve the same in a much simpler way.

You can extend your stack from the `defaultStack` and override the `excludeMethods` parameter of the validation interceptor in the following way to achieve the same result.

```

01.<!DOCTYPE struts PUBLIC

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

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

04.

05.<struts>

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

07.<action name="*Sample" method="{1}" class="vaannila.SampleAction">

08.<interceptor-ref name="defaultStack" >

```

```
09.<param name="validation.excludeMethods"> populate</param>
10.</interceptor-ref>
11.<result name="populate">/first.jsp</result>
12.<result name="success">/success.jsp</result>
13.</action>
14.</package>
15.</struts>
```

Our SampleAction class contains the following code.

```
01.package vaannila;

02.

03.import com.opensymphony.xwork2.ActionSupport;

04.

05.public class SampleAction extends ActionSupport{

06.

07.private static final long serialVersionUID = 1L;

08.

09.public void validate()

10.{

11.System.out.println("validate() method called");

12.}

13.

14.public String populate()

15.{

16.System.out.println("populate() method called");

17.return "populate";

18.}

19.
```

```

20.public String execute()
21.{
22.System.out.println("execute() method called");
23.return SUCCESS;
24.}
25.}

```

When you run the code using the `defaultStack` without any changes. The following messages gets printed in the console.

```

1.validate() method called
2.populate() method called
3.validate() method called
4.execute() method called

```

When you run the code the using the `exampleStack` we just created. The follwing messages gets printed in the console.

```

1.populate() method called
2.validate() method called
3.execute() method called

```

As you can see the `validate()` method is not invoked during `populate`. In this way you can customise the stack base on your requirement.

DispatchAction functionality in Struts 2

In Struts 1 `DispatchAction` helps us in grouping a set of related functions into a single action. In Struts 2 all the Actions by default provide this functionality. To use this functionality we need to create different methods with the similar signature of the `execute()` method, only the name of the method changes.

In our example the `UserAction` class contains all the functions related to a User like `addUser()`, `updateUser()` and `deleteUser()`.

```

01.package vaannila;
02.
03.import com.opensymphony.xwork2.ActionSupport;
04.
05.public class UserAction extends ActionSupport{
06.
07.private String message;

```

```
08.

09.public String execute()

10.{

11.message = "Inside execute method";

12.return SUCCESS;

13.}

14.

15.public String add()

16.{

17.message = "Inside add method";

18.return SUCCESS;

19.}

20.

21.public String update()

22.{

23.message = "Inside update method";

24.return SUCCESS;

25.}

26.

27.public String delete()

28.{

29.message = "Inside delete method";

30.return SUCCESS;

31.}

32.

33.public String getMessage() {

34.return message;

35.}
```

36.

```
37.public void setMessage(String message) {
```

```
38.this.message = message;
```

```
39.}
```

40.

```
41.}
```

Unlike Struts 1 you can even have the `execute()` method along with the other methods in the Action class. We need to specify which method in the action class will be called while configuring the action mapping. A separate action mapping needs to be created for each method in the action class. The following action mapping is done in the `struts.xml` file.

```
01.<!DOCTYPE struts PUBLIC
```

```
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
```

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

04.

```
05.<struts>
```

```
06.<package name="default" extends="struts-default">
```

```
07.<action name="User" class="vaannila.UserAction">
```

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

```
09.</action>
```

```
10.<action name="addUser" method="add" class="vaannila.UserAction">
```

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

```
12.</action>
```

```
13.<action name="updateUser" method="update" class="vaannila.UserAction">
```

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

```
15.</action>
```

```
16.<action name="deleteUser" method="delete" class="vaannila.UserAction">
```

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

```
18.</action>
```

```
19.</package>
```

```
20.</struts>
```

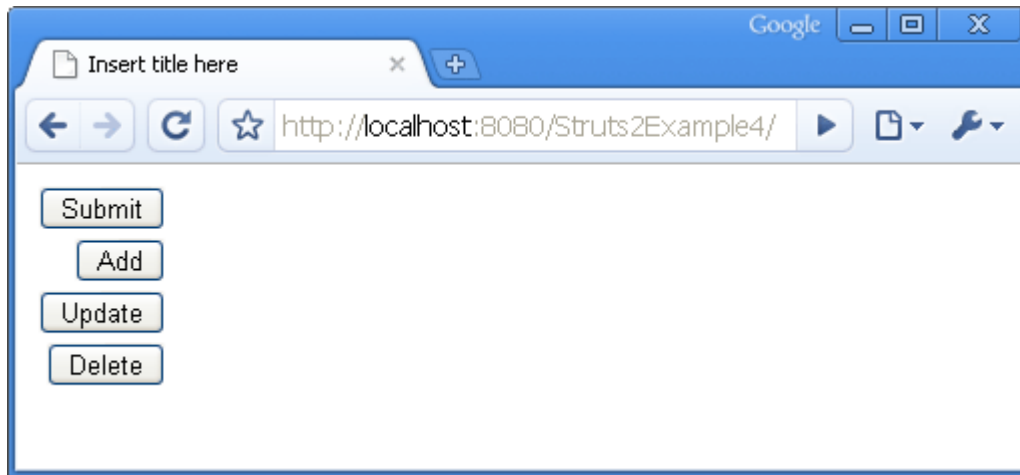
Note that we use the same action class in all the action mappings. When the request URL is "User" the execute() method in the UserAction class will be invoked. When the request URL is "addUser" the add() method in the UserAction class will be invoked, this is specified using the method attribute of the action tag. Similarly for update and delete request the updateUser() and deleteUser() methods will be invoked respectively.

Configuring a separate action mapping for each method in the action class can be avoided by using another feature of Struts 2 called Dynamic Method Invocation. The next example explains how to do this. ([Dynamic Method Invocation Example](#))

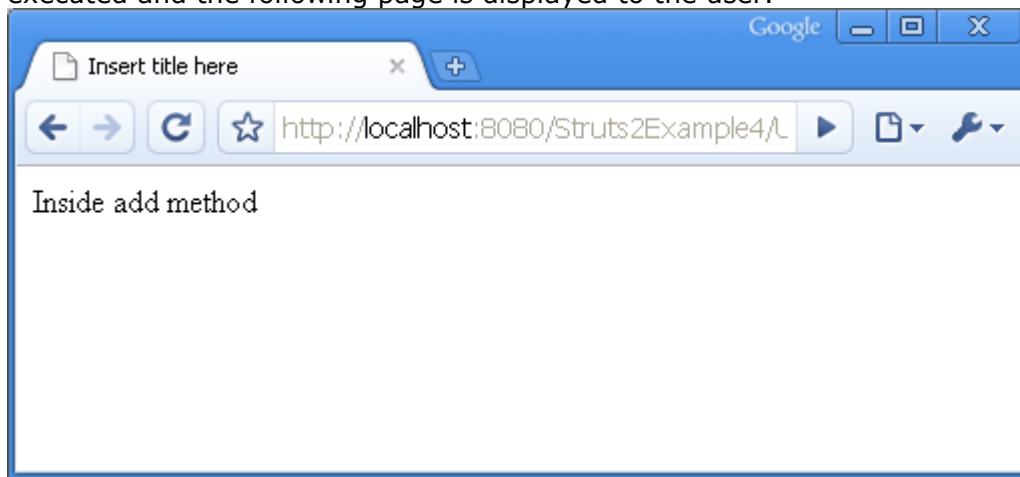
In the index.jsp page we create four different buttons to invoke the different methods in the UserAction class. The submit tag is used to create the buttons in the jsp page.

```
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.<%@taglib uri="/struts-tags" prefix="s" %>
05.<html>
06.<head>
07.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
08.<title>Insert title here</title>
09.</head>
10.<body>
11.<s:form action="User" >
12.<s:submit />
13.<s:submit action="addUser" value="Add" />
14.<s:submit action="updateUser" value="Update" />
15.<s:submit action="deleteUser" value="Delete" />
16.</s:form>
17.</body>
18.</html>
```

Now lets execute the example. On running the example the following page will be displayed to the user. When the user click a button the appropriate method in the UserAction class gets invoked.



When the user clicks the Add button the `addUser()` method in the `UserAction` class gets executed and the following page is displayed to the user.



Dynamic Method Invocation

This is a continuation of the previous example ([DispatchAction functionality in Struts 2](#)). In this example you will see how you can avoid configuring a separate action mapping for each method in the Action class by using the wildcard method. Look at the following action mapping.

```
01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.<struts>
06.<package name="default" extends="struts-default">
07.<action name="*User" method="{1}" class="vaannila.UserAction">
08.<result name="success">/success.jsp</result>
```

```
09.</action>

10.</package>

11.</struts>
```

Note the similarity between the action mapping and the following request URLs in this page.

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

05.<html>

06.<head>

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

08.<title>Insert title here</title>

09.</head>

10.<body>

11.<s:form action="User" >

12.<s:submit />

13.<s:submit action="addUser" value="Add" />

14.<s:submit action="updateUser" value="Update" />

15.<s:submit action="deleteUser" value="Delete" />

16.</s:form>

17.</body>

18.</html>
```

As you can see we have replaced all the method names with an asterisk symbol. The word that matches for the first asterisk will be substituted for the method attribute. So when the request URL is "addUser" the add() method in the UserAction class will be invoked.

Struts 2 Validation Example

In this example we will see how we can validate a login page using Struts 2. Let's first create the login page. We use Struts UI tags to create the login page. The <s:head /> tag should be placed in the head section of the HTML page. The s:head tag automatically generates links to the css and

javascript libraries that are necessary to render the form elements.

The `s:form` tag contains all the form elements. The `action` attribute contains the action name to which the form should be submitted. This action name should be same as the one specified in the XML declarative architecture. In this example we use `struts.xml` file to do the configuration.

The `textfield` tag is used to create a text box. The `label` attribute of the `textfield` tag contains the name to be displayed on the page and the `name` attribute contains the name of the property in the action class to be mapped. The `password` tag is same as the `textfield` tag except that the input value is masked. The `submit` tag is used to create a submit button, the value "Login" represents the label of the button.

Note that the code is simple without any HTML tables, this is because Struts 2 will automatically create the necessary tables for the page based on the theme selected. By default the XHTML theme is selected.

```
01.<%@taglib uri="/struts-tags" prefix="s" %>
02.<html>
03.<head>
04.<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
05.<title>Login Page</title>
06.<s:head />
07.</head>
08.<body>
09.<s:form action="Login">
10.<s:textfield name="userName" label="User Name" />
11.<s:password name="password" label="Password" />
12.<s:submit value="Login" />
13.</s:form>
14.</body>
15.</html>
```

When the user clicks the Login button the request will be forwarded to the Login action.

We do the action mapping using the `struts.xml` file. First we need to create a package for our action.

```
1.<struts>
```

```
2.<package name="default" extends="struts-default">
3.<action name="Login" class="vaannila.Login">
4.<result name="input">/login.jsp</result>
5.<result name="success">/success.jsp</result>
6.</action>
7.</package>
8.</struts>
```

Here our "default" package extends "struts-default" package. By extending the "struts-default" package the action will by default inherit the set of interceptors defined in the defaultstack. The "struts-default" package is defined in the struts-default.xml file.

All the common tasks done by the Actions are seperated and placed in different interceptors. You can define an interceptor stack for each action. Most commonly used interceptors are grouped in defaultstack of the struts-default package. The defaultstack will be sufficient in most cases. The inteceptors will be fired in the order in which they are declared in the stack both before and after the action is executed.

Here the "Login" action is mapped to the "Login" class in the "vaannila" package. The results are defined using the "<result>" element. If any validation errors occur the user will be forwarded to the login.jsp page. If the login is successfull then the user will be forwarded to the success.jsp page.

The defaultstack contains the following interceptors.

```
01.<interceptor-stack name="defaultStack">
02.<interceptor-ref name="exception"/>
03.<interceptor-ref name="alias"/>
04.<interceptor-ref name="servletConfig"/>
05.<interceptor-ref name="prepare"/>
06.<interceptor-ref name="i18n"/>
07.<interceptor-ref name="chain"/>
08.<interceptor-ref name="debugging"/>
09.<interceptor-ref name="profiling"/>
10.<interceptor-ref name="scopedModelDriven"/>
11.<interceptor-ref name="modelDriven"/>
12.<interceptor-ref name="fileUpload"/>
```

```

13.<interceptor-ref name="checkbox"/>
14.<interceptor-ref name="staticParams"/>
15.<interceptor-ref name="actionMappingParams"/>
16.<interceptor-ref name="params">
17.<param name="excludeParams">dojo\..*,^struts\..*</param>
18.</interceptor-ref>
19.<interceptor-ref name="conversionError"/>
20.<interceptor-ref name="validation">
21.<param name="excludeMethods">input,back,cancel,browse</param>
22.</interceptor-ref>
23.<interceptor-ref name="workflow">
24.<param name="excludeMethods">input,back,cancel,browse</param>
25.</interceptor-ref>
26.</interceptor-stack>

```

Our Login Action class extends ActionSupport. It is good to extend ActionSupport class as it provides default implementation for most common tasks.

```

01.public class Login extends ActionSupport {
02.
03.private String userName;
04.private String password;
05.
06.public Login() {
07.}
08.
09.public String execute() {
10.return SUCCESS;
11.}
12.
13.public void validate() {
14.if (getUserName().length() == 0) {

```

```

15.addFieldError("userName", "User Name is required");
16.} else if (!getUserName().equals("Eswar")) {
17.addFieldError("userName", "Invalid User");
18.}

19.if (getPassword().length() == 0) {
20.addFieldError("password", getText("password.required"));
21.}

22.}

23.

24.public String getUserName() {
25.return userName;
26.}

27.

28.public void setUserName(String userName) {
29.this.userName = userName;
30.}

31.

32.public String getPassword() {
33.return password;
34.}

35.

36.public void setPassword(String password) {
37.this.password = password;
38.}

39.}

```

The ActionSupport class implements Action interface which exposes the execute() method.

The following constants are declared in the Action interface which can be used as return values in the execute() method.

```

public static final String ERROR = "error"

public static final String INPUT = "input"

```

```
public static final String LOGIN = "login"

public static final String NONE = "none"

public static final String SUCCESS = "success"
```

ERROR is returned when the action execution fails.

INPUT is returned when the action requires more input from the user.

LOGIN is returned when the user is not logged into the system.

NONE is returned when the action execution is successful and there are no views to display.

SUCCESS is returned when the action executed successfully and the corresponding result is displayed to the user.

Now let's see the roles played by the different interceptors.

The params interceptor helps in transferring the request data onto the action object.

The workflow interceptor controls the flow of control.

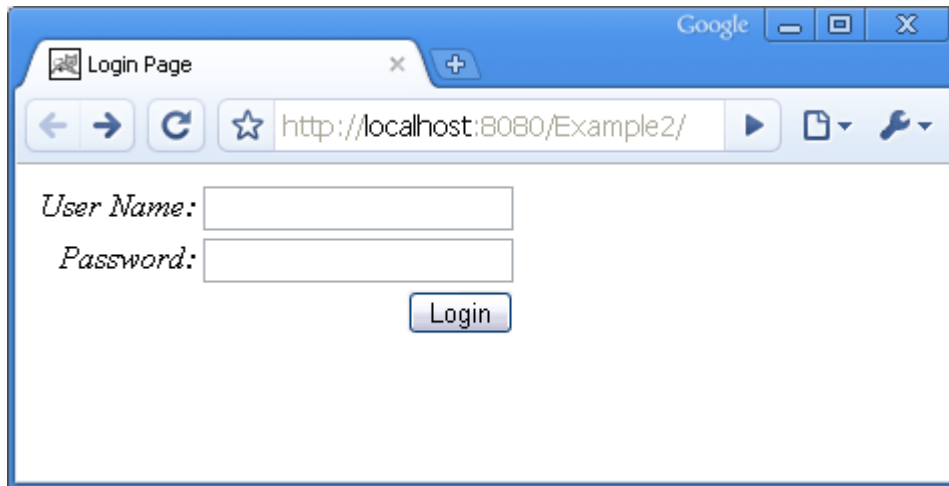
The workflow interceptor checks whether the action implements the `Validateable` interface, if it does, the workflow interceptor will invoke the `validate()` method of the Action class.

In the `validate()` method we validate the user name and the password. If the validation fails an error is added using the `addFiledError()` method.

The `validate()` method doesn't return any errors, instead it stores all the errors with the help of the `ValidationAware` interface.

Now the workflow interceptor will check any validation errors that have occurred. If any error has occurred the workflow interceptor will stop the request processing and transfer the control to the input page with the appropriate error messages.

On executing the sample example the following page will be displayed to the user.



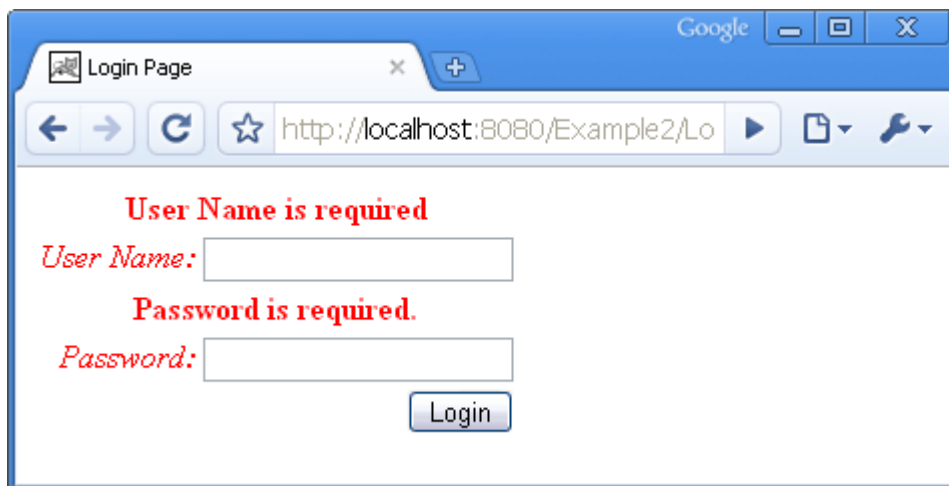
For each field the error messages can be added using the `addFieldError()` method. The error messages can either be added directly or it can be specified in a separate properties file.

The properties files should have the same name as the Action class. In our case the properties file name is "Login.properties" and the Action name is "Login.java".

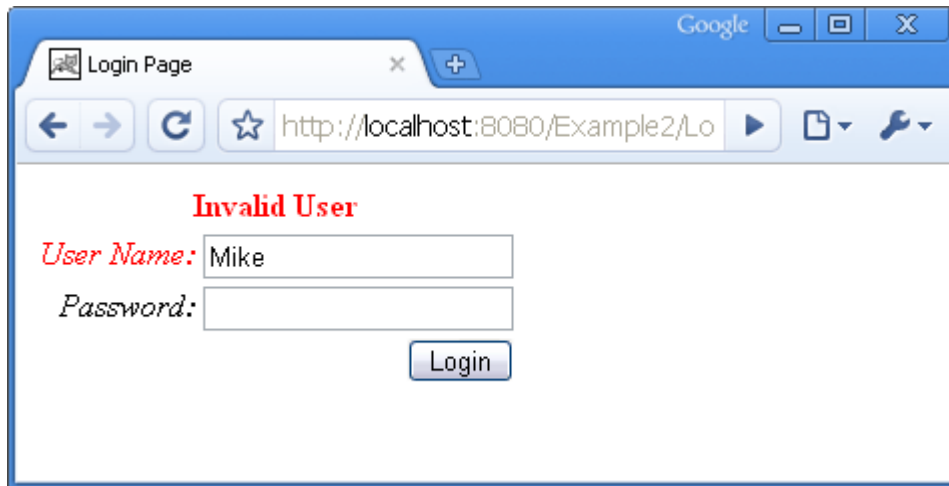
The Login.properties file contains the following entry.

```
1.password.required = Password is required.
```

The `getText()` method provided by the `TextProvider` interface can be used to retrieve the error messages.

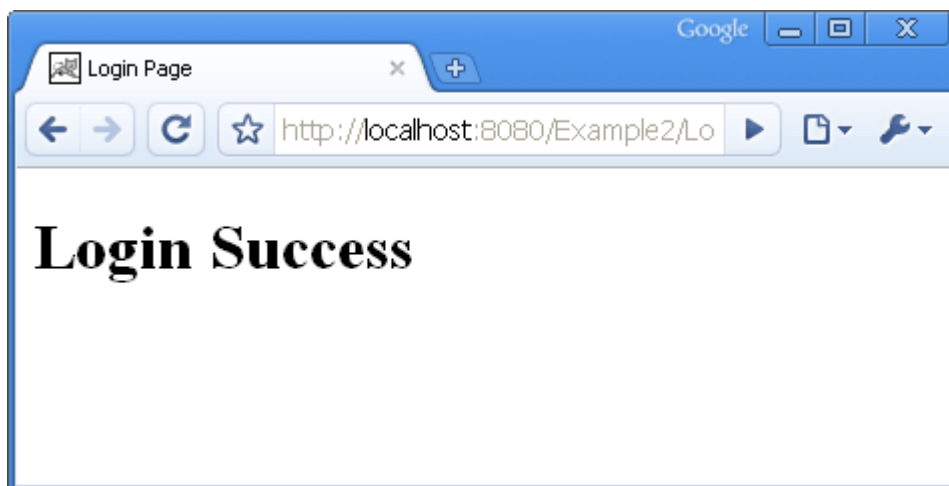


You can also do the business validations in the `validate()` method.



If there are no errors, then the `execute()` method will be invoked by the workflow interceptor.

In our `execute()` method we simply return "success". The user will be forwarded to the success page.



Struts 2 Validation Using XML File Example

In this example we will see how we can perform validations using XML validation file. The naming convention of the XML validation file should be `ActionClass-Validation.xml`. Here our Action Class name is "Login.java" and the XML validation file name is "Login-Validation.xml".

The Login-Validation.xml file contains the following code.

```
01.<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0.2//EN"
```

```
02."http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
```

```
03.
```

```
04.<validators>
```

```

05.<field name="userName">
06.<field-validator type="requiredstring">
07.<message>User Name is required.</message>
08.</field-validator>
09.</field>
10.<field name="password">
11.<field-validator type="requiredstring">
12.<message key="password.required" />
13.</field-validator>
14.</field>
15.</validators>

```

The field element contains the name of the form property that needs to be validated. The field-validator element inside the field element contains the type of validation that needs to be performed.

Here you can either specify the error message directly using the message element or you can use the properties file to define all the error messages and use the key attribute to specify the error key.

Note the properties file should also have the same name as the Action class.

The Login Action class contains the following code.

```

01.public class Login extends ActionSupport {
02.
03.private String userName;
04.private String password;
05.
06.public Login() {
07.}
08.
09.public String execute() {
10.return SUCCESS;
11.}
12.

```



```
13.public String getUsername() {
14.return userName;
15.}
16.
17.public void setUsername(String userName) {
18.this.userName = userName;
19.}
20.
21.public String getPassword() {
22.return password;
23.}
24.
25.public void setPassword(String password) {
26.this.password = password;
27.}
28.}
```

The login.jsp page contains the following code.

```
01.<%@page contentType="text/html" pageEncoding="UTF-8"%>
02.<%@taglib uri="/struts-tags" prefix="s" %>
03.<html>
04.<head>
05.<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
06.<title>Login Page</title>
07.<s:head />
08.</head>
09.<body>
10.<s:form action="LoginAction">
11.<s:textfield name="userName" label="User Name" />
12.<s:password name="password" label="Password" />
13.<s:submit value="Login" />
```

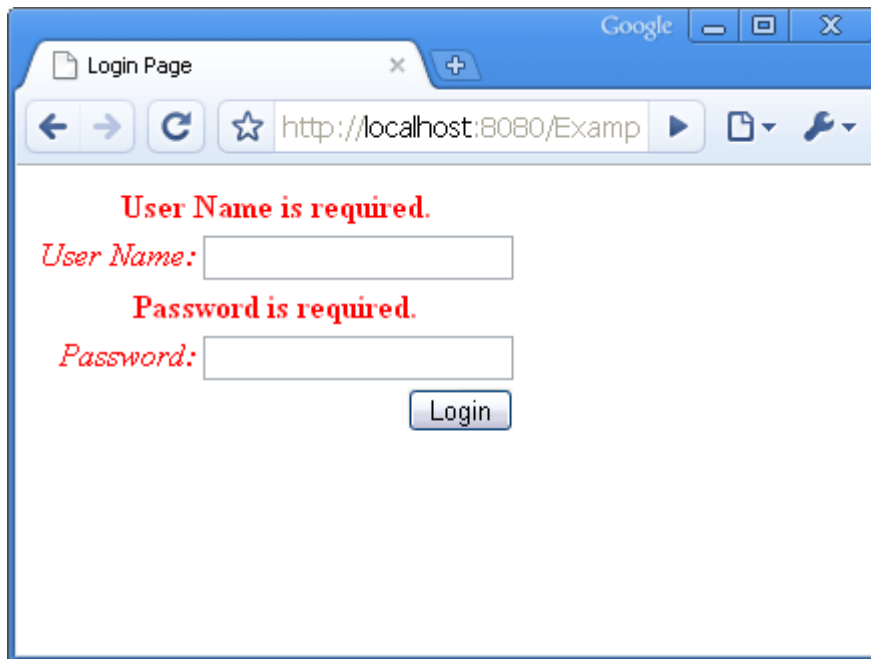
```
14.</s:form>
```

```
15.</body>
```

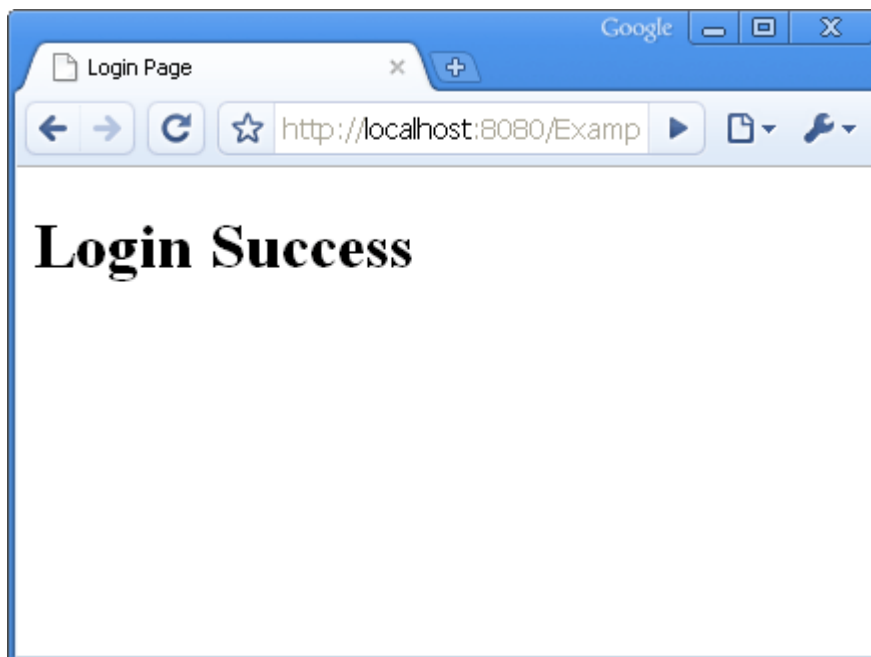
```
16.</html>
```

The `<s:head />` tag is used to include the required css and js file for the selected theme. By default the xhtml theme is used.

Execute the example and click the Login button without entering the user name and password the following page will be displayed.



Enter a user name and password and click the Login button the following success page will be displayed.



Domain Object as JavaBeans Property

In Struts 2 transferring the data to the domain object is automatically done by the `paramsinterceptor`.

You just need to create a domain object as a JavaBeans property and the corresponding getter and setter methods.

The framework will automatically initialize the domain object and transfers the form data. The `UserAction` class contains the following code.

```
01.public class UserAction extends ActionSupport{
02.
03.private User user;
04.
05.public UserAction() {
06.}
07.
08.public String execute() {
09.return SUCCESS;
10.}
11.
12.public User getUser() {
13.return user;
14.}
15.
16.public void setUser(User user) {
17.this.user = user;
18.}
19.}
```

To refer the user attributes like name, age etc. we need to first get the user object and then access its properties. For example to access the user's age in the Action you need to use the following syntax.

```
1.getUser().getAge();
```

The `User` class contains the following attributes and the corresponding getter and setter methods.

```
01.public class User {
02.
03.private String name;
04.private int age;
05.private String sex;
06.private String[] hobby;
07.private String country;
08.
09.
10.public String getName() {
11.return name;
12.}
13.
14.public void setName(String name) {
15.this.name = name;
16.}
17.
18.public int getAge() {
19.return age;
20.}
21.
22.public void setAge(int age) {
23.this.age = age;
24.}
25.
26.public String getSex() {
27.return sex;
28.}
29.
30.public void setSex(String sex) {
```

```

31.this.sex = sex;
32.}
33.
34.public String[] getHobby() {
35.return hobby;
36.}
37.
38.public void setHobby(String[] hobby) {
39.this.hobby = hobby;
40.}
41.
42.public String getCountry() {
43.return country;
44.}
45.
46.public void setCountry(String country) {
47.this.country = country;
48.}
49.}

```

In the jsp page the user attributes cannot be directly referenced. Since the attributes we refer in the jsp page belongs to the User object we need to go one level deeper to reference the attributes. To refer the user's age, the value of the name attribute should be

```
1.name="user.age"
```

The index.jsp page contains the following code.

```

01.<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
02."http://www.w3.org/TR/html4/loose.dtd">
03.<%@taglib uri="/struts-tags" prefix="s" %>
04.<html>
05.<head>
06.<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

```

```

07.<title>User Details</title>
08.</head>
09.<body>
10.<s:form action="UserAction" >
11.<s:textfield name="user.name" label="User Name" />
12.<s:textfield name="user.age" label="Age" />
13.<s:radio name="user.sex" label="Sex" list="{ 'M','F' }" />
14.<s:checkboxlist name="user.hobby" label="Hobby" list="{ 'Music','Art','Dance' }" />
15.<s:select name="user.country" label="Country" list="{ 'Select','India','USA','France','Spain' }" />
16.<s:submit />
17.</s:form>
18.</body>
19.</html>

```

The result.jsp page contains the following code.

```

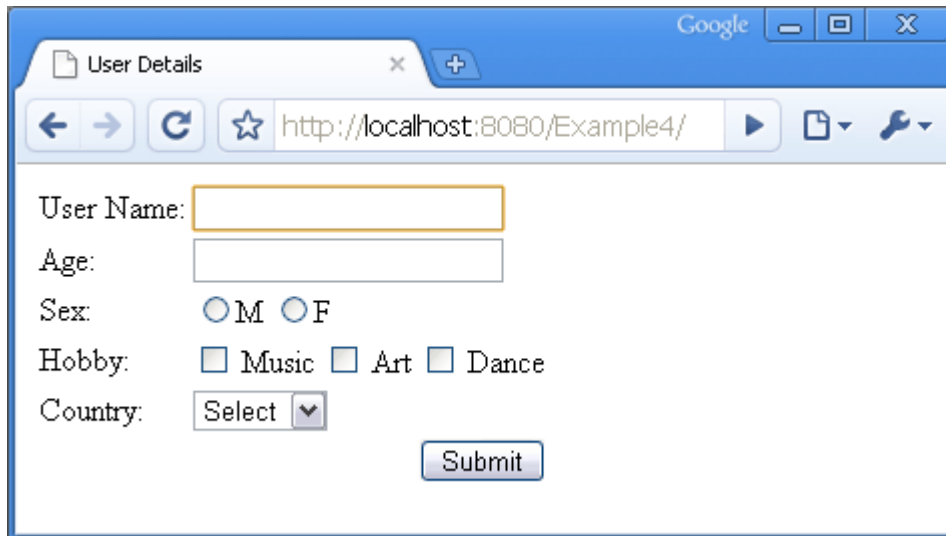
01.<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
02."http://www.w3.org/TR/html4/loose.dtd">
03.<%@taglib uri="/struts-tags" prefix="s" %>
04.<html>
05.<head>
06.<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
07.<title>User Details</title>
08.</head>
09.<body>
10.<h2>User Details</h2>
11.<hr>
12.User Name :<s:property value="user.name" /><br>
13.Age :<s:property value="user.age" /><br>
14.Hobbies :<s:property value="user.hobby" /><br>
15.Country :<s:property value="user.country" /><br>

```

16.</body>

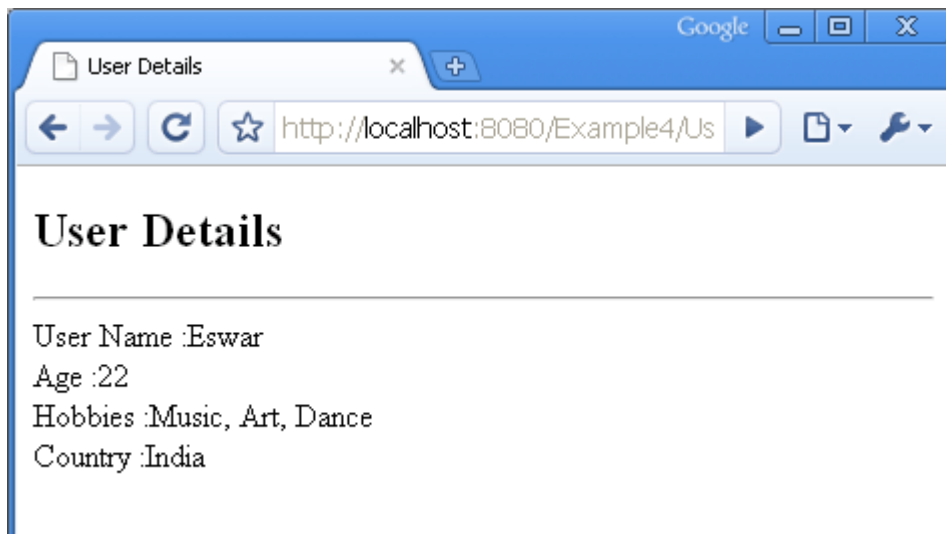
17.</html>

On executing the example the following page will be displayed to the user.



A screenshot of a web browser window with a single tab titled 'User Details'. The address bar shows 'http://localhost:8080/Example4/'. The form contains the following fields: 'User Name:' with a text input, 'Age:' with a text input, 'Sex:' with radio buttons for 'M' and 'F', 'Hobby:' with checkboxes for 'Music', 'Art', and 'Dance', and 'Country:' with a dropdown menu showing 'Select'. A 'Submit' button is located at the bottom right of the form.

On entering the user details and clicking the Submit button the following page will be displayed.



A screenshot of the same web browser window after the form has been submitted. The address bar now shows 'http://localhost:8080/Example4/Us'. The page displays the title 'User Details' followed by a horizontal line. Below the line, the submitted data is shown: 'User Name :Eswar', 'Age :22', 'Hobbies :Music, Art, Dance', and 'Country :India'.

Struts 2 ModelDriven Action

To create a ModelDriven Action our Action class should implement the ModelDriven interface and should include the modelDriven interceptor. The modelDriven interceptor is already included in the default stack.

The next step is to implement the getModel() method in such a way that it returns the application domain object, in our example we return the User object.

When using the ModelDriven method we need to initialize the User object ourselves.

The framework will automatically transfers the form data into the User object.

```
01.public class UserAction extends ActionSupport implements ModelDriven {
02.
03.private User user = new User();
04.
05.public UserAction() {
06.}
07.
08.public Object getModel() {
09.return user;
10.}
11.
12.public String execute() {
13.return SUCCESS;
14.}
15.
16.public User getUser() {
17.return user;
18.}
19.
20.public void setUser(User user) {
21.this.user = user;
22.}
23.}
```

You can directly access the user attributes like name, age etc in Action use the following syntax.

```
1.user.getXXX();
```

The User class contains the following attributes and the corresponding getter and setter methods.

```
01.public class User {
```



```
02.
03.private String name;
04.private int age;
05.private String sex;
06.private String[] hobby;
07.private String country;
08.
09.
10.public String getName() {
11.return name;
12.}
13.
14.public void setName(String name) {
15.this.name = name;
16.}
17.
18.public int getAge() {
19.return age;
20.}
21.
22.public void setAge(int age) {
23.this.age = age;
24.}
25.
26.public String getSex() {
27.return sex;
28.}
29.
30.public void setSex(String sex) {
31.this.sex = sex;
```

```

32.}

33.

34.public String[] getHobby() {

35.return hobby;

36.}

37.

38.public void setHobby(String[] hobby) {

39.this.hobby = hobby;

40.}

41.

42.public String getCountry() {

43.return country;

44.}

45.

46.public void setCountry(String country) {

47.this.country = country;

48.}

49.}

```

In the jsp page the user attributes can be accessed directly. To refer the user's age, the value of the name attribute should be

```
1.name = "age"
```

The index.jsp page contains the following code.

```

01.<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

02."http://www.w3.org/TR/html4/loose.dtd">

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

04.<html>

05.<head>

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

07.<title>User Details</title>

08.</head>

```

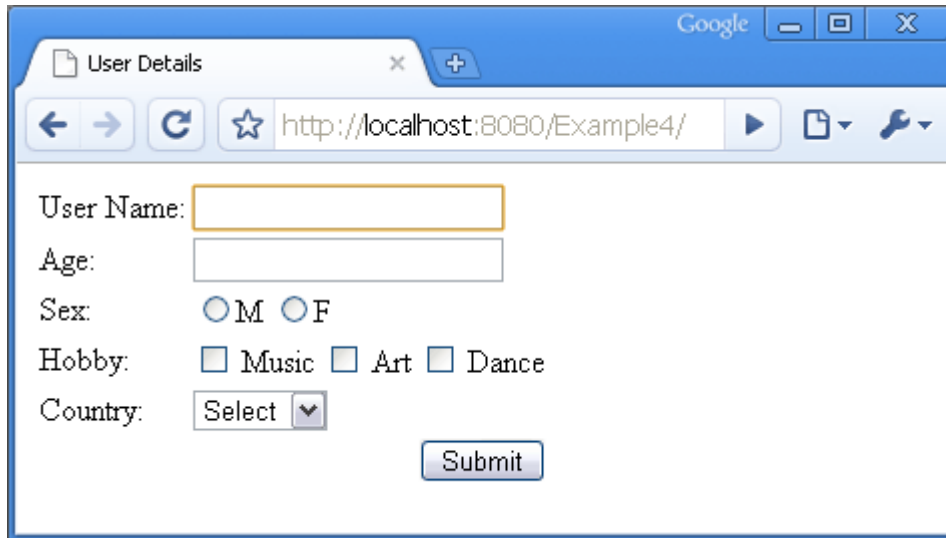
```
09.<body>
10.<s:form action="UserAction" >
11.<s:textfield name="name" label="User Name" />
12.<s:textfield name="age" label="Age" />
13.<s:radio name="sex" label="Sex" list="{ 'M', 'F' }" />
14.<s:checkboxlist name="hobby" label="Hobby"
15.list="{ 'Music', 'Art', 'Dance' }" />
16.<s:select name="country" label="Country"
17.list="{ 'Select', 'India', 'USA', 'France', 'Spain' }" />
18.<s:submit />
19.</s:form>
20.</body>
21.</html>
```

The result.jsp page contains the following code.

```
01.<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
02."http://www.w3.org/TR/html4/loose.dtd">
03.<%@taglib uri="/struts-tags" prefix="s" %>
04.<html>
05.<head>
06.<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
07.<title>User Details</title>
08.</head>
09.<body>
10.<h2>User Details</h2>
11.<hr>
12.User Name :<s:property value="name" /><br>
13.Age :<s:property value="age" /><br>
14.Hobbies :<s:property value="hobby" /><br>
15.Country :<s:property value="country" /><br>
16.</body>
```

17.</html>

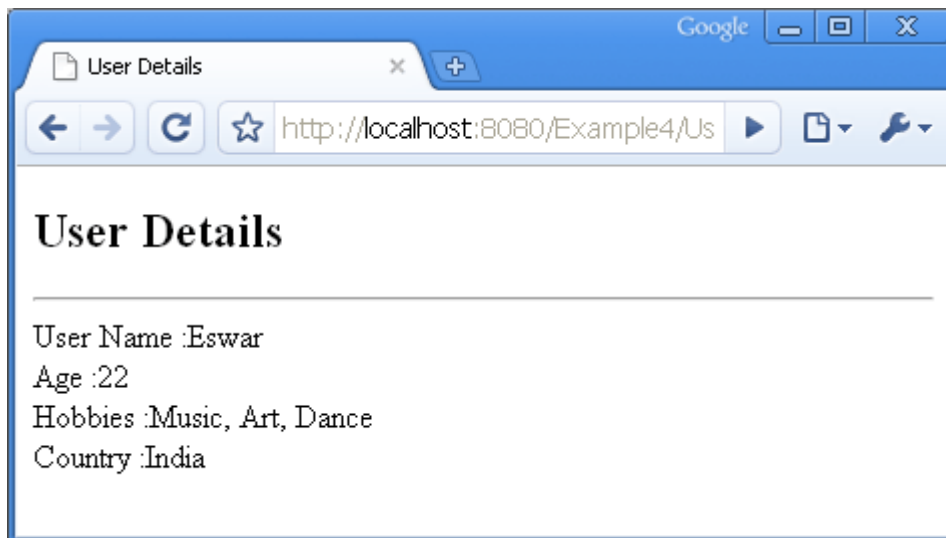
On executing the example the following page will be displayed to the user.



The screenshot shows a web browser window with a single tab titled 'User Details'. The address bar shows 'http://localhost:8080/Example4/'. The form contains the following elements:

- User Name:
- Age:
- Sex: ☐ M ☐ F
- Hobby: ☐ Music ☐ Art ☐ Dance
- Country: ▼
- Submit:

On entering the user details and clicking the Submit button the following page will be displayed.



The screenshot shows the same web browser window after the form has been submitted. The page title is 'User Details'. The displayed information is:

User Name :Eswar
Age :22
Hobbies :Music, Art, Dance
Country :India

Struts 2 File Upload Example

In this example you will learn how to do file upload with the help of the built-inFileUploadInterceptor. To do this first we need to get the file form the user. We use the Struts 2 tags to build our form. The encoding type of the form should be set tomultipart/form-data and the HTTP method should be set to post. The index.jsp page contains the following code.

```
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.<%@taglib uri="/struts-tags" prefix="s" %>
05.<html>
06.<head>
07.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
08.<title>Insert title here</title>
09.<s:head />
10.</head>
11.<body>
12.<s:form action="fileUpload" method="post" enctype="multipart/form-
data" >
13.<s:file name="userImage" label="User Image" />
14.<s:submit />
15.</s:form>
16.</body>
17.</html>

```

Here our file name is userImage. Next we will create our FileUploadAction class. We need to have a File attribute and the corresponding getter and setter methods in the action to receive the userImage file.

You will also get information regarding the file name and content type of the file if you implement the following setter methods. This step is optional if you implement the setter methods you will get more details regarding the file.

```

1.public void setUserImageContentType(String userImageContentType) {
2.this.userImageContentType = userImageContentType;
3.}
4.public void setUserImageFileName(String userImageFileName) {
5.this.userImageFileName = userImageFileName;
6.}

```

The FileUploadAction extends ActionSupport and contains the following code.

```

01.package vaannila;
02.
03.import java.io.File;

```

```
04.
05.import com.opensymphony.xwork2.ActionSupport;
06.
07.public class FileUploadAction extends ActionSupport{
08.
09.private File userImage;
10.
11.private String userImageContentType;
12.
13.private String userImageFileName;
14.
15.public String execute()
16.{
17.return SUCCESS;
18.}
19.
20.public File getUserImage() {
21.return userImage;
22.}
23.
24.public void setUserImage(File userImage) {
25.this.userImage = userImage;
26.}
27.
28.public String getUserImageContentType() {
29.return userImageContentType;
30.}
31.
32.public void setUserImageContentType(String userImageContentType) {
33.this.userImageContentType = userImageContentType;
```

```

34.}

35.

36.public String getUserImageFileName() {

37.return userImageFileName;

38.}

39.

40.public void setUserImageFileName(String userImageFileName) {

41.this.userImageFileName = userImageFileName;

42.}

43.

44.}

```

When the file is uploaded successful, the user will be forwarded to the success page, where the details regarding the file upload will be displayed. The success.jsp page contains the following code.

```

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>File Upload</title>

08.</head>

09.<body>

10.You have uploaded the following file.

11.<hr>

12.File Name : ${userImageFileName} <br>

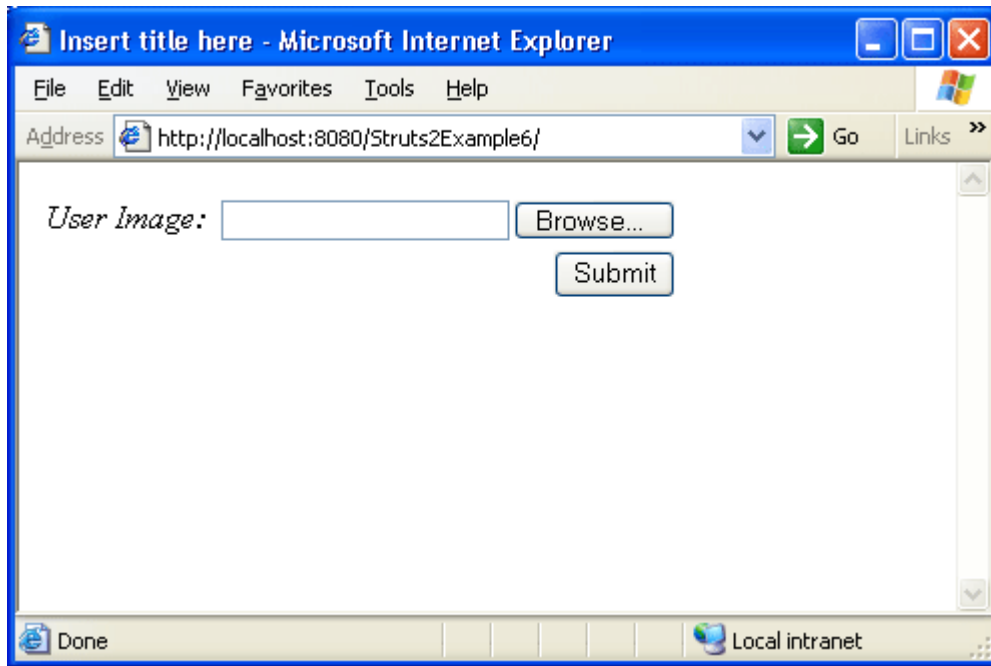
13.Content Type : ${userImageContentType}

14.</body>

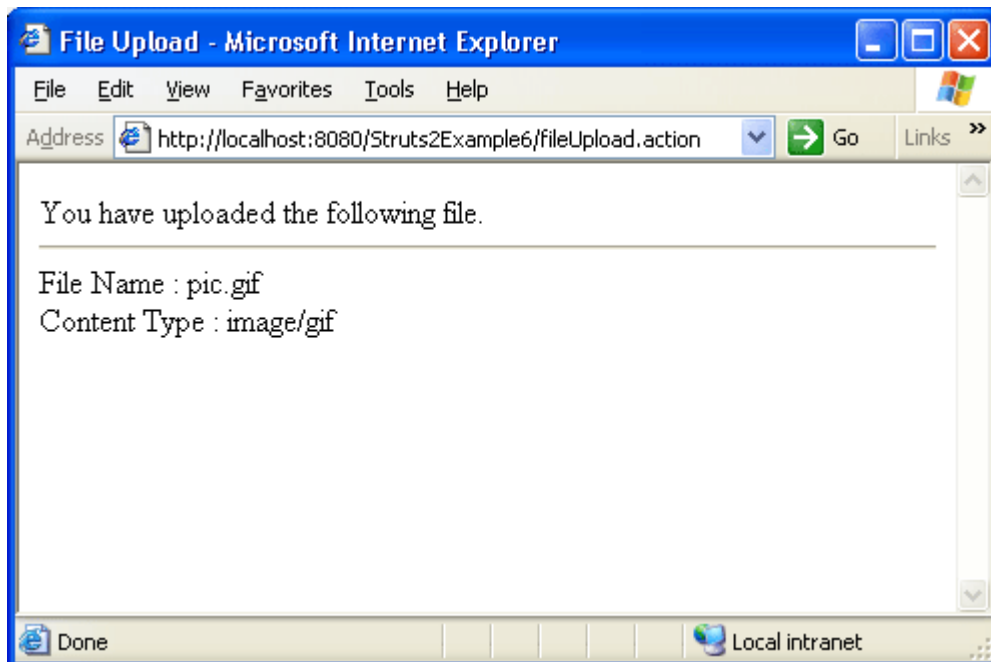
15.</html>

```

Now let's run the example.



When the user selects an image file and upload it. The following success page will be displayed to the user.



We only need an image file, what if the user uploads a pdf document? Let's see how to validate this in the next page.

We can validate either programmatically or declaratively. Let's see how to do programmatically first. Validate the file in the `validate()` method of the `FileUploadAction` class. You can get the file using the `getX()` method and the file type type using the `getXContentType()` method.

In this example we validate declaratively. To do this we need to create our own interceptor stack. The only change we need to do is to add few parameters to the fileUpload interceptor. So we copy the defaultStack from

the struts-default.xml and paste it in our struts.xml file and rename the stack to fileUploadStack. Our struts.xml file contains the following code.

```
01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.
06.<struts>
07.<package name="fileUploadPackage" extends="struts-default">
08.<interceptors>
09.<interceptor-stack name="fileUploadStack">
10.<interceptor-ref name="exception" />
11.<interceptor-ref name="alias" />
12.<interceptor-ref name="servletConfig" />
13.<interceptor-ref name="prepare" />
14.<interceptor-ref name="i18n" />
15.<interceptor-ref name="chain" />
16.<interceptor-ref name="debugging" />
17.<interceptor-ref name="profiling" />
18.<interceptor-ref name="scopedModelDriven" />
19.<interceptor-ref name="modelDriven" />
20.<interceptor-ref name="fileUpload">
21.<param name="maximumSize">10240</param>
22.<param name="allowedTypes"> image/jpeg,image/gif,image/png</param>
23.</interceptor-ref>
24.<interceptor-ref name="checkbox" />
25.<interceptor-ref name="staticParams" />
26.<interceptor-ref name="actionMappingParams" />
27.<interceptor-ref name="params">
28.<param name="excludeParams"> dojo\..*,^struts\..*</param>
```

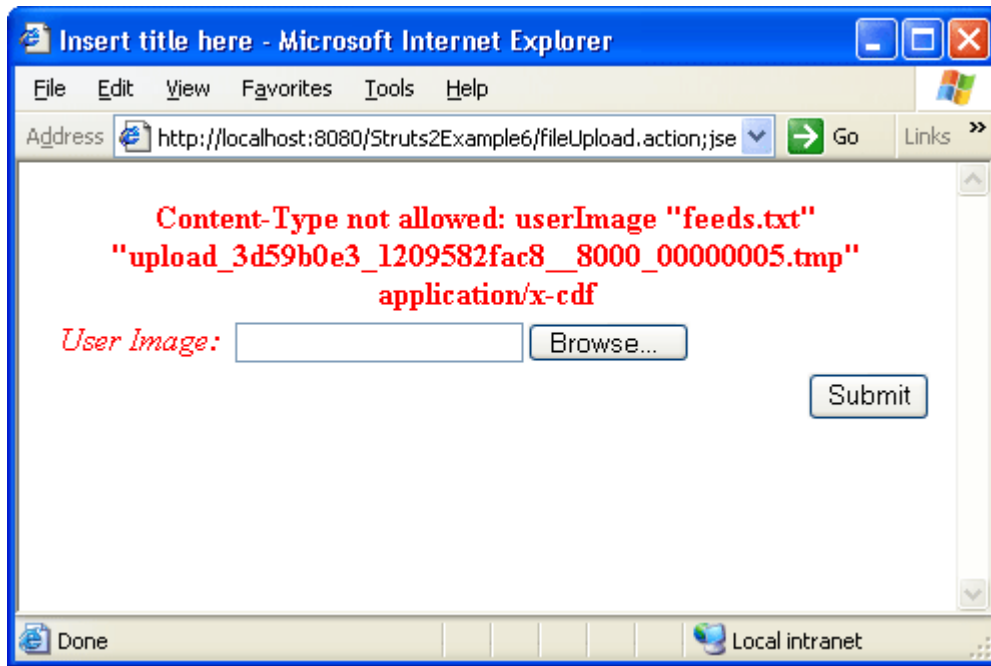
```

29.</interceptor-ref>
30.<interceptor-ref name="conversionError" />
31.<interceptor-ref name="validation">
32.<param name="excludeMethods"> input,back, cancel,browse</param>
33.</interceptor-ref>
34.<interceptor-ref name="workflow">
35.<param name="excludeMethods"> input,back, cancel,browse</param>
36.</interceptor-ref>
37.</interceptor-stack>
38.</interceptors>
39.
40.<action name="fileUpload" class="vaannila.FileUploadAction">
41.<interceptor-ref name="fileUploadStack" />
42.<result name="input">/index.jsp</result>
43.<result name="success">/success.jsp</result>
44.</action>
45.</package>
46.</struts>

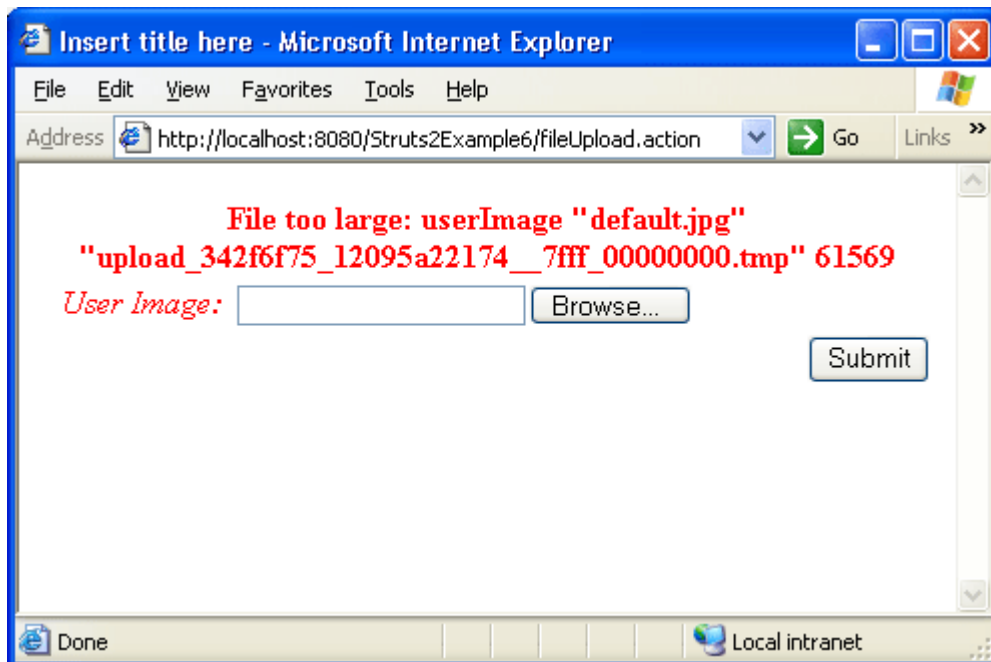
```

The `maximumSize` value is set in bytes. Here we set the `maximumSize` to 10kb. The `allowedTypes` indicate the file types that can be uploaded. Here we set it to only image files like `image/jpeg`, `image/gif`, `image/png`.

Now let's see how the validation works. We will upload a text file. The following error message is displayed to the user.



Now we will upload a file greater than 10kb, the following error message will be displayed.



Struts 2 Spring Integration

In this simple hello world example you will see how to integrate Spring and Struts 2 using the struts2-spring-plugin. By doing this you can utilize the Spring's powerful Dependency Injection feature. To learn more about Dependency Injection refer this [example](#).

First add the `org.springframework.web.context.ContextLoaderListener` to the `web.xml` file.

```
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.su
n.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.<display-name>Struts2Example14</display-name>

04.<filter>

05.<filter-name>struts2</filter-name>

06.<filter-class>org.apache.struts2.dispatcher.ng.filter.
StrutsPrepareAndExecuteFilter</filter-class>

07.</filter>

08.<listener>

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

10.</listener>

11.<filter-mapping>

12.<filter-name>struts2</filter-name>

13.<url-pattern>/*</url-pattern>

14.</filter-mapping>

15.<welcome-file-list>

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

17.</welcome-file-list>

18.</web-app>

```

By default the applicationContext.xml file will be used for doing the Spring bean configuration.

```

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

2.<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

3.<beans>

4.<bean id="helloWorldClass" class="com.vaannila.HelloWorld" >

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

6.</bean>

```

```
7.</beans>
```

As you can see we have registered the HelloWorld class and injected the "Hello World!" message to the message attribute using the setter injection method.

All the Struts 2 action configuration goes in the struts.xml file.

```
01.<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 2.0//EN" "http://struts.apache.org/dtds/struts-2.0.dtd">
```

```
02.
```

```
03.<struts>
```

```
04.<package name="default" extends="struts-default">
```

```
05.<action name="helloWorld" class="helloWorldClass">
```

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

```
07.</action>
```

```
08.</package>
```

```
09.</struts>
```

The only change here is instead of referring the com.vaannila.HelloWorld class directly, we relate it using the bean name given in the spring bean configuration file.

The HelloWorld class is shown below. In the execute() method we simply return "SUCCESS" and the message attribute is set using setter injection.

```
01.package com.vaannila;
```

```
02.
```

```
03.public class HelloWorld {
```

```
04.
```

```
05.private String message;
```

```
06.
```

```
07.public String getMessage() {
```

```
08.return message;
```

```
09.}
```

```
10.
```

```
11.public void setMessage(String message) {
```

```
12.this.message = message;
```

13.}

14.

15.public String execute() {

16.return "SUCCESS";

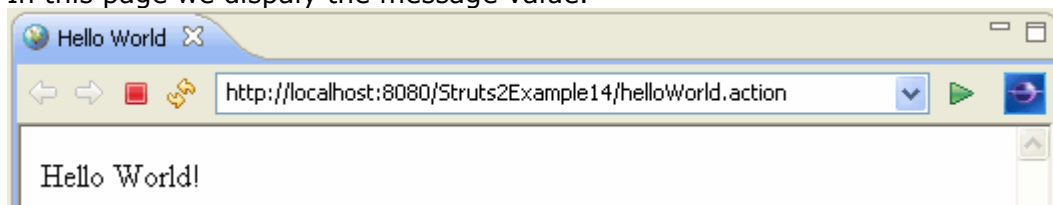
17.}

18.}

In the `index.jsp` page we forward the request to the `helloWorld` action.

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

After invoking the `execute()` method the user will be directed to the `success.jsp` page. In this page we display the message value.



You need to have the following jar files in the `WEB-INF/lib` directory.

01.commons-fileupload-1.2.1

02.commons-io-1.3.2

03.commons-logging-1.1

04.freemarker-2.3.13

05.junit-3.8.1

06.ognl-2.6.11

07.struts2-convention-plugin-2.1.6

08.struts2-core-2.1.6

09.xwork-2.1.2

10.

11.struts2-spring-plugin-2.1.6

12.

13.antlr-runtime-3.0

14.org.springframework.asm-3.0.0.M3

15.org.springframework.beans-3.0.0.M3

16.org.springframework.context-3.0.0.M3

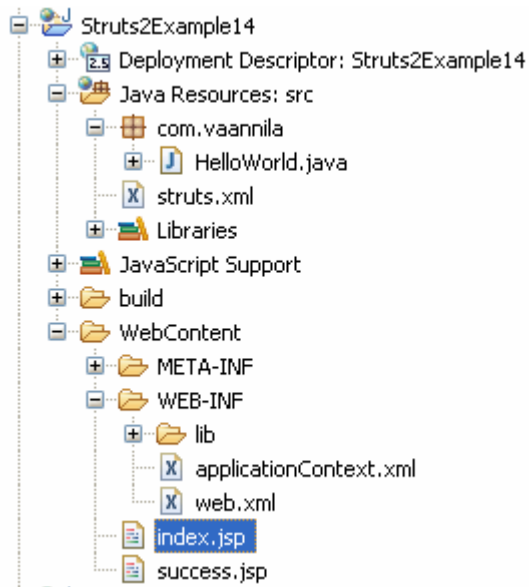
17.org.springframework.core-3.0.0.M3

18.org.springframework.expression-3.0.0.M3

19.org.springframework.web-3.0.0.M3

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

The directory structure of the example is shown below.



Struts 2 Tiles Example

The following example shows how to integrate Struts 2 and Tiles using the struts2 tiles plugin. In the deployment descriptor first setup the tiles definition file.

```
1.<context-param>

2.<param-name> org.apache.tiles.impl.BasicTilesContainer.DEFINITIONS_CONFIG
</param-name>

3.<param-value>/WEB-INF/tiles.xml</param-value>

4.</context-param>
```

Then setup the tiles listener.

```
1.<listener>

2.<listener-class>org.apache.struts2.tiles.StrutsTilesListener</listener-
class>

3.</listener>
```

The complete 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"xmlns:web="http://java.sun.com/
xml/ns/javaee/web-app_2_5.xsd"

04.xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"

05.id="WebApp_ID" version="2.5">

06.<display-name>Struts2Example15</display-name>

07.

08.<context-param>

09.<param-name>
org.apache.tiles.impl.BasicTilesContainer.DEFINITIONS_CONFIG </param-name>

10.<param-value>/WEB-INF/tiles.xml</param-value>

11.</context-param>

12.

13.<listener>

14.<listener-class>org.apache.struts2.tiles.StrutsTilesListener </listener-
class>

15.</listener>

16.

17.<filter>

18.<filter-name>struts2</filter-name>

19.<filter-class>

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

21.</filter-class>

22.</filter>

23.

24.<filter-mapping>

25.<filter-name>struts2</filter-name>

26.<url-pattern>/*</url-pattern>

27.</filter-mapping>

28.

29.<welcome-file-list>
```



```
30.<welcome-file>index.jsp</welcome-file>
31.</welcome-file-list>
32.</web-app>
```

The tiles.xml file contains the following tile definitions.

```
01.<?xml version="1.0" encoding="UTF-8" ?>
02.
03.<!DOCTYPE tiles-definitions PUBLIC
04."-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
05."http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
06.
07.<tiles-definitions>
08.
09.<definition name="baseLayout" template="/baseLayout.jsp">
10.<put-attribute name="title" value="Template"/>
11.<put-attribute name="header" value="/header.jsp"/>
12.<put-attribute name="menu" value="/menu.jsp"/>
13.<put-attribute name="body" value="/body.jsp"/>
14.<put-attribute name="footer" value="/footer.jsp"/>
15.</definition>
16.
17.<definition name="welcome" extends="baseLayout">
18.<put-attribute name="title" value="Welcome"/>
19.<put-attribute name="body" value="/welcome.jsp"/>
20.</definition>
21.
22.<definition name="friends" extends="baseLayout">
```

```

23.<put-attribute name="title" value="Friends"/>

24.<put-attribute name="body" value="/friends.jsp"/>

25.</definition>

26.

27.<definition name="office" extends="baseLayout">

28.<put-attribute name="title" value="Office"/>

29.<put-attribute name="body" value="/office.jsp"/>

30.</definition>

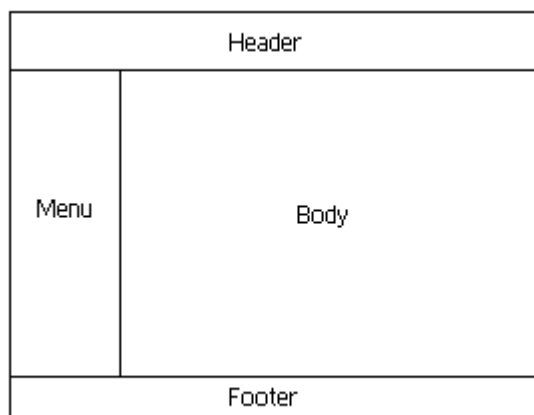
31.

32.</tiles-definitions>

```

Here we define a "baseLayout" that contains a title, header, menu, body and footer regions. The header, menu and footer region remains the same for all the layouts only the title and body content changes.

In the baseLayout.jsp page we create a classic tiles layout as shown below.



```

01.<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>

02.<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"

03."http://www.w3.org/TR/html4/loose.dtd">

04.

05.<html>

06.<head>

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

```

```
08.<title>

09.<tiles:insertAttribute name="title" ignore="true" />

10.</title>

11.</head>

12.<body>

13.<table border="1" cellpadding="2" cellspacing="2" align="center">

14.<tr>

15.<td height="30" colspan="2">

16.<tiles:insertAttribute name="header" />

17.</td>

18.</tr>

19.<tr>

20.<td height="250">

21.<tiles:insertAttribute name="menu" />

22.</td>

23.<td width="350">

24.<tiles:insertAttribute name="body" />

25.</td>

26.</tr>

27.<tr>

28.<td height="30" colspan="2">

29.<tiles:insertAttribute name="footer" />

30.</td>

31.</tr>

32.</table>

33.</body>

34.</html>
```

In the struts.xml file create a new result type for tiles as shown below.

```
01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.<struts>
06.<package name="default" extends="struts-default">
07.<result-types>
08.<result-
type name="tiles"class="org.apache.struts2.views.tiles.TilesResult" />
09.</result-types>
10.<action name="*Link" method="{1}"class="com.vaannila.action.LinkAction">
11.<result name="welcome" type="tiles">welcome</result>
12.<result name="friends" type="tiles">friends</result>
13.<result name="office" type="tiles">office</result>
14.</action>
15.</package>
16.</struts>
```

For each result instead of forwarding to the jsp page forward it to the tiles definition.
When you execute the example the following page gets displayed.



The menu.jsp page has the menu items, on clicking each menu item the title and body content alone changes.

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

```
2.
```

```
3.<a href="<s:url action="friendsLink"/>" >Friends</a><br>
```

```
4.<a href="<s:url action="officeLink"/>" >The Office</a><br>
```

When each menu item is clicked a different method in the LinkAction class is invoked.

```
01.package com.vaannila.action;
```

```
02.
```

```
03.import com.opensymphony.xwork2.ActionSupport;
```

```
04.
```

```
05.public class LinkAction extends ActionSupport {
```

```
06.
```

```
07.private static final long serialVersionUID = -2613425890762568273L;
```

```
08.
```

```
09.public String welcome()
```

```
10.{
11.return "welcome";
12.}

13.

14.public String friends()
15.{
16.return "friends";
17.}

18.

19.public String office()
20.{
21.return "office";
22.}

23.}
```

You need the following lib files to run the example.

```
01.commons-fileupload-1.2.1
02.commons-io-1.3.2
03.commons-logging-1.1
04.freemarker-2.3.13
05.junit-3.8.1
06.ognl-2.6.11
07.struts2-convention-plugin-2.1.6
08.struts2-core-2.1.6
09.xwork-2.1.2

10.

11.struts2-tiles-plugin-2.1.6.jar

12.

13.tiles-api-2.1.2
14.tiles-compat-2.1.2
15.tiles-core-2.1.2
```

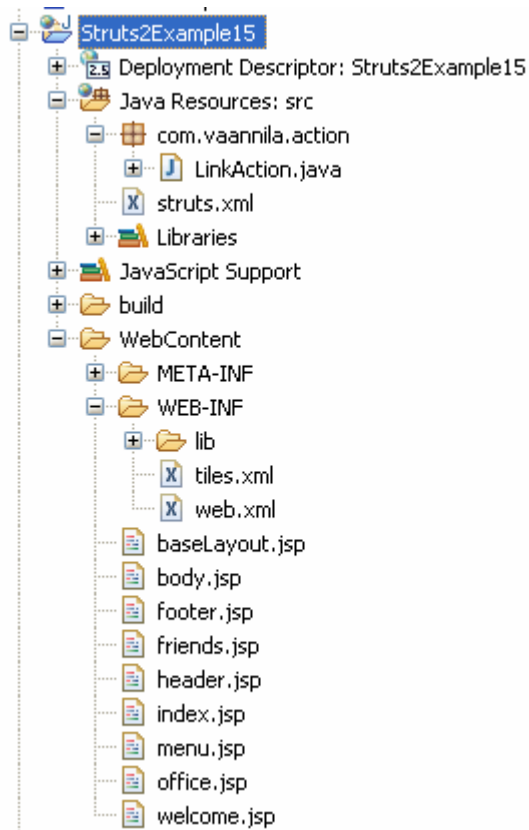
16.tiles-jsp-2.1.2

17.tiles-servlet-2.1.2

18.commons-beanutils-1.8.0

19.commons-digester-1.8.1

The directory structure of the example is shown below.



Struts 2 Hibernate Integration

In this example you will see how to integrate Struts 2 and Hibernate using the "Full Hibernate Plugin 1.4 GA" ([Full Hibernate Plugin 1.4 GA](#)).

You will see how to add a user using the user registration form shown below and to list all the existing users.

Registration Page

http://localhost:8080/Struts2Example17/listUser

User Name:

Password:

Gender: ☐ Male ☐ Female

Select a country:

About You:

☐ Would you like to join our mailing list?

Name	Gender	Country	About You	Mailing List
Eswar	Male	India	Software Engineer	true
Joy	Male	India	Ruby programmer	false

To use the Full Hibernate Plugin 1.4 GA you need to add the following lib files to the libdirectory.

01.antlr-2.7.6.jar

02.commons-collections-3.1.jar

03.commons-fileupload-1.2.1.jar

04.commons-io-1.3.2.jar

05.commons-lang-2.3.jar

06.commons-logging-1.1.jar

07.dom4j-1.6.1.jar

08.ejb3-persistence.jar

09.freemarker-2.3.13.jar

10.hibernate3.jar

11.hibernate-annotations.jar

12.hibernate-commons-annotations.jar

13.hibernate-validator.jar

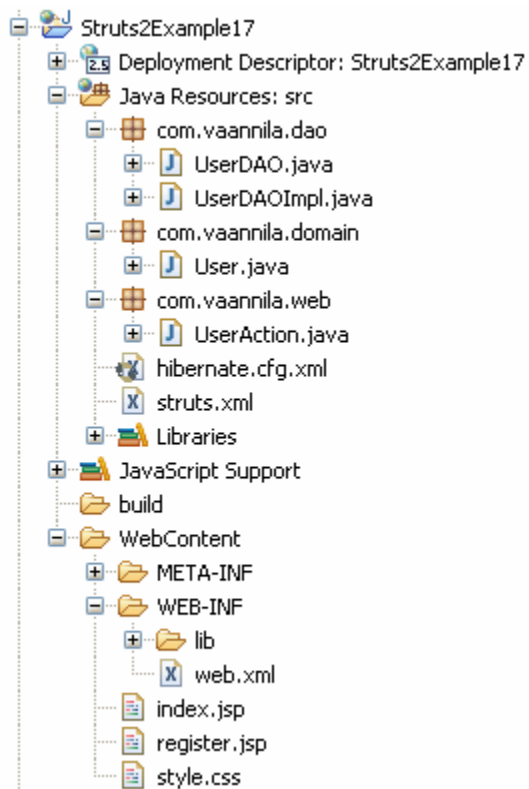
14.hsqldb.jar

15.javassist-3.9.0.GA.jar

16.jta-1.1.jar

17.junit-3.8.1.jar
18.log4j-1.2.15.jar
19.ognl-2.6.11.jar
20.slf4j-api-1.5.8.jar
21.slf4j-log4j12-1.5.8.jar
22.struts2-convention-plugin-2.1.6.jar
23.struts2-core-2.1.6.jar
24.struts2-fullhibernatecore-plugin-1.4-GA.jar
25.xwork-2.1.2.jar

The directory structure of the example is shown below.



The session object and transaction object will be injected using the @SessionTarget and @TransactionTarget annotation respectively.

```
01.package com.vaannila.dao;  
  
02.  
  
03.import java.util.List;  
  
04.  
  
05.import org.hibernate.Session;
```

```
06.import org.hibernate.Transaction;

07.

08.import com.googlecode.s2hibernate.struts2.plugin.annotations.SessionTarget;

09.import com.googlecode.s2hibernate.struts2.plugin.annotations.TransactionTarget;

10.import com.vaannila.domain.User;

11.

12.public class UserDAOImpl implements UserDAO {

13.

14.@SessionTarget

15.Session session;

16.

17.@TransactionTarget

18.Transaction transaction;

19.

20.@SuppressWarnings("unchecked")

21.@Override

22.public List<User> listUser() {

23.List<User> courses = null;

24.try {

25.courses = session.createQuery("from User").list();

26.} catch (Exception e) {

27.e.printStackTrace();

28.}

29.return courses;

30.}

31.
```

```

32.@Override

33.public void saveUser(User user) {

34.try {

35.session.save(user);

36.} catch (Exception e) {

37.transaction.rollback();

38.e.printStackTrace();

39.}

40.}

41.

42.}

```

For the session and transaction injection to happen through the plug-in the `org.hibernate.Session` and `org.hibernate.Transaction` objects **should be declared as class variables** and not at method level. You can keep these variables in a generic DAO class and extend all the other DAO's from it. When using this plug-in there is **no need to explicitly commit the transaction and close the session**, both will be done automatically.

The `"transaction.rollback()"` should be used only in the methods that updates the database.

In the hibernate configuration file, we configure it to work with the hsqldb database.

```

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

02.<!DOCTYPE hibernate-configuration PUBLIC

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

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

05.<hibernate-configuration>

06.<session-factory>

07.<property name="hibernate.connection.driver_class">
org.hsqldb.jdbcDriver </property>

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

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

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

```

```
11.<property name="connection.pool_size">1</property>

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

13.<property name="show_sql">true</property>

14.<property name="hbm2ddl.auto">create</property>

15.<mapping class="com.vaannila.domain.User" />

16.</session-factory>

17.</hibernate-configuration>
```

The domain object User class is shown below.

```
01.package com.vaannila.domain;

02.

03.import javax.persistence.Column;

04.import javax.persistence.Entity;

05.import javax.persistence.GeneratedValue;

06.import javax.persistence.Id;

07.import javax.persistence.Table;

08.

09.@Entity
10.@Table (name="USER")
11.public class User {

12.

13.private Long id;

14.private String name;

15.private String password;

16.private String gender;

17.private String country;
```

```
18.private String aboutYou;

19.private Boolean mailingList;

20.

21.@Id
22.@GeneratedValue
23.@Column(name="USER_ID")
24.public Long getId() {

25.return id;

26.}

27.public void setId(Long id) {

28.this.id = id;

29.}

30.

31.@Column(name="USER_NAME")
32.public String getName() {

33.return name;

34.}

35.public void setName(String name) {

36.this.name = name;

37.}

38.

39.@Column(name="USER_PASSWORD")
40.public String getPassword() {

41.return password;

42.}

43.public void setPassword(String password) {

44.this.password = password;
```

```
45.}

46.

47.@Column(name="USER_GENDER")

48.public String getGender() {

49.return gender;

50.}

51.public void setGender(String gender) {

52.this.gender = gender;

53.}

54.

55.@Column(name="USER_COUNTRY")

56.public String getCountry() {

57.return country;

58.}

59.public void setCountry(String country) {

60.this.country = country;

61.}

62.

63.@Column(name="USER_ABOUT_YOU")

64.public String getAboutYou() {

65.return aboutYou;

66.}

67.public void setAboutYou(String aboutYou) {

68.this.aboutYou = aboutYou;

69.}

70.

71.@Column(name="USER_MAILING_LIST")
```

```
72.public Boolean getMailingList() {  
  
73.return mailingList;  
  
74.}  
  
75.public void setMailingList(Boolean mailingList) {  
  
76.this.mailingList = mailingList;  
  
77.}  
  
78.  
  
79.}
```

In the UserAction class we have two methods add() and list() to add and list all users respectively.

```
01.package com.vaannila.web;  
  
02.  
  
03.import java.util.ArrayList;  
  
04.import java.util.List;  
  
05.  
  
06.import com.opensymphony.xwork2.ActionSupport;  
  
07.import com.opensymphony.xwork2.ModelDriven;  
  
08.import com.vaannila.dao.UserDAO;  
  
09.import com.vaannila.dao.UserDAOImpl;  
  
10.import com.vaannila.domain.User;  
  
11.  
  
12.public class UserAction extends ActionSupport implements ModelDriven<User>  
{  
  
13.  
  
14.private static final long serialVersionUID = -6659925652584240539L;  
  
15.  
  
16.private User user = new User();
```

```
17.private List<User> userList = new ArrayList<User>();

18.private UserDAO userDAO = new UserDAOImpl();

19.

20.@Override

21.public User getModel() {

22.return user;

23.}

24.

25.public String add()

26.{

27.userDAO.saveUser(user);

28.return SUCCESS;

29.}

30.

31.public String list()

32.{

33.userList = userDAO.listUser();

34.return SUCCESS;

35.}

36.

37.public User getUser() {

38.return user;

39.}

40.

41.public void setUser(User user) {

42.this.user = user;
```



```

43.}

44.

45.public List<User> getUserList() {

46.return userList;

47.}

48.

49.public void setUserList(List<User> userList) {

50.this.userList = userList;

51.}

52.

53.}

```

For the session and transaction injection to happen through the plug-in, the `UserDAO` **should be a class level declaration** and should not be declared at method level.

To use this plug-in you need to extend the package from `hibernate-default` package. To configure this you can either use XML or annotations, I am using XML.

```

01.<!DOCTYPE struts PUBLIC

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

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

04.

05.<struts>

06.<package name="default" extends="hibernate-default">

07.<action name="addUser" method="add"class="com.vaannila.web.UserAction">

08.<result name="success" type="redirect">listUser</result>

09.</action>

10.<action name="listUser" method="list"class="com.vaannila.web.UserAction">

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

12.</action>

13.</package>

```

14.</struts>

The register.jsp page is shown below.

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

05.<html>

06.<head>

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

08.<title>Registration Page</title>

09.<s:head />

10.<style type="text/css">

11.@import url(style.css);

12.</style>

13.</head>

14.<body>

15.<s:form action="addUser">

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

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

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

19.<s:select name="country" list="{ 'India', 'USA', 'UK' }" headerKey=""

20.headerValue="Country" label="Select a country" />

21.<s:textarea name="aboutYou" label="About You" />

22.<s:checkbox name="mailingList"

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

24.<s:submit />

25.</s:form>

```
26.

27.<s:if test="userList.size() > 0">

28.<div class="content">

29.<table class="userTable" cellpadding="5px">

30.<tr class="even">

31.<th>Name</th>

32.<th>Gender</th>

33.<th>Country</th>

34.<th>About You</th>

35.<th>Mailing List</th>

36.</tr>

37.<s:iterator value="userList" status="userStatus">

38.<tr

39.class="<s:if test="#userStatus.odd == true
">odd</s:if><s:else>even</s:else>">

40.<td><s:property value="name" /></td>

41.<td><s:property value="gender" /></td>

42.<td><s:property value="country" /></td>

43.<td><s:property value="aboutYou" /></td>

44.<td><s:property value="mailingList" /></td>

45.</tr>

46.</s:iterator>

47.</table>

48.</div>

49.</s:if>

50.</body>

51.</html>
```

Struts 2 Hibernate Validation Tutorial

The Hibernate Validator framework follows the DRY (Don't Repeat Yourself) principle. Using Hibernate Validator you need to specify the constraints using annotations in the domain object. Once you specify the constraints you can use it in any layer of your application without duplicating it.

Hibernate Validator comes with basic built-in constraints like `@Length(min=, max=)`, `@Max(value=)`, `@Min(value=)`, `@NotNull`, `@NotEmpty` and so on. You can also build your own constraints easily.

In this example you will see how to integrate Struts 2 with Hibernate Validator using the Full Hibernate Plugin 1.4 GA.

You need to have all the lib files that we used in the previous example ([Struts 2 Hibernate Integration](#)).

The domain object `User`, with the validation constraints is shown below.

```
001.package com.vaannila.domain;
002.
003.import java.io.Serializable;
004.
005.import javax.persistence.Column;
006.import javax.persistence.Entity;
007.import javax.persistence.GeneratedValue;
008.import javax.persistence.Id;
009.import javax.persistence.Table;
010.
011.import org.hibernate.validator.Length;
012.import org.hibernate.validator.NotEmpty;
013.
014.@Entity
015.@Table(name = "USER")
016.public class User implements Serializable {
017.
018.private static final long serialVersionUID = 6295524232169619097L;
019.
020.private Long id;
021.
022.private String name;
023.
024.private String password;
025.
026.private String gender;
027.
028.private String country;
029.
030.private String aboutYou;
031.
032.private Boolean mailingList;
033.
034.@Id
035.@GeneratedValue
```

```
036.@Column(name = "USER_ID")
037.public Long getId() {
038.return id;
039.}
040.
041.public void setId(Long id) {
042.this.id = id;
043.}
044.
045.@NotEmpty
046.@Length(max=50)
047.@Column(name = "USER_NAME", nullable = false, length = 50)
048.public String getName() {
049.return name;
050.}
051.
052.public void setName(String name) {
053.this.name = name;
054.}
055.
056.@Length(min=6, max=10)
057.@Column(name = "USER PASSWORD", nullable = false, length = 10)
058.public String getPassword() {
059.return password;
060.}
061.
062.public void setPassword(String password) {
063.this.password = password;
064.}
065.
066.@NotEmpty(message="Please select a gender")
067.@Column(name = "USER GENDER")
068.public String getGender() {
069.return gender;
070.}
071.
072.public void setGender(String gender) {
073.this.gender = gender;
074.}
075.
076.@NotEmpty(message="Please select a country")
077.@Column(name = "USER_COUNTRY")
078.public String getCountry() {
079.return country;
080.}
081.
082.public void setCountry(String country) {
083.this.country = country;
084.}
085.
086.@NotEmpty
087.@Length(max=100)
```

```

088.@Column(name = "USER ABOUT YOU", length = 100)
089.public String getAboutYou() {
090.return aboutYou;
091.}
092.
093.public void setAboutYou(String aboutYou) {
094.this.aboutYou = aboutYou;
095.}
096.
097.@Column(name = "USER MAILING LIST")
098.public Boolean getMailingList() {
099.return mailingList;
100.}
101.
102.public void setMailingList(Boolean mailingList) {
103.this.mailingList = mailingList;
104.}
105.
106.}

```

As you can see in addition to the JPA annotations you have the Hibernate Validator constraints.

The **@NotEmpty** constraint checks if the String is not null or not empty.

The **@Length(min=6, max=10)** constraint checks whether the length is within the min max range.

The validation messages are auto generated by the plug-in. You can also override the default message using the message attribute of the constraint. For gender and country properties we specify a customized message.

```

01.@NotEmpty(message="Please select a gender")
02.@Column(name = "USER GENDER")
03.public String getGender() {
04.return gender;
05.}
06.
07.@NotEmpty(message="Please select a country")
08.@Column(name = "USER_COUNTRY")
09.public String getCountry() {
10.return country;
11.}

```

In the `UserAction` class you need to specify the `@Valid` annotation for the domain object that needs to be validated.

```

01.package com.vaannila.web;
02.
03.import java.util.ArrayList;
04.import java.util.List;
05.
06.import org.hibernate.validator.Valid;
07.
08.import com.opensymphony.xwork2.ActionSupport;
09.import com.vaannila.dao.UserDAO;
10.import com.vaannila.dao.UserDAOImpl;
11.import com.vaannila.domain.User;

```

```

12.
13.public class UserAction extends ActionSupport {
14.
15.private static final long serialVersionUID = -6659925652584240539L;
16.
17.@Valid
18.private User user;
19.private List<User> userList = new ArrayList<User>();
20.private UserDAO userDAO = new UserDAOImpl();
21.
22.public String add()
23.{
24.userDAO.saveUser(user);
25.return SUCCESS;
26.}
27.
28.public String list()
29.{
30.userList = userDAO.listUser();
31.return SUCCESS;
32.}
33.
34.public User getUser() {
35.return user;
36.}
37.
38.public void setUser(User user) {
39.this.user = user;
40.}
41.
42.public List<User> getUserList() {
43.return userList;
44.}
45.
46.public void setUserList(List<User> userList) {
47.this.userList = userList;
48.}
49.
50.}

```

Since we use the `Hibernate` Plugin you need to extend the package from `hibernate-default` package. The `hibernate-default` package has the following three interceptor stacks.

- **basicStackHibernate:** Struts2 basickStack + Hibernate session and transaction capability.
- **defaultStackHibernate:** Struts2 defaultStack (no validations) + Hibernate validation, session and transaction capability.
- **defaultStackHibernateStrutsValidation:** Struts2 defaultStack (with validation) + basicStackHibernate.

The struts configuration file is shown below.

```

01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.

```

```

05.<struts>
06.<package name="default" extends="hibernate-default">
07.<action name="addUser" method="add"class="com.vaannila.web.UserAction">
08.<result name="input">/register.jsp</result>
09.<result name="success" type="redirect">listUser</result>
10.</action>
11.<action name="listUser" method="list"class="com.vaannila.web.UserAction">
12.<interceptor-ref name="basicStackHibernate" />
13.<result name="success">/register.jsp</result>
14.</action>
15.</package>
16.</struts>

```

By default the **defaultStackHibernate** interceptor stack will be used. In the `addUser` action we use this inteceptor stack since we need validation capability and during the `listUser` action we use **basicStackHibernate** because we don't need validation capability this time.

In the `register.jsp` page instead of using the `name` attribute to specify the property value we use the `key` attribute. This is need for the default validation messages to be generated.

```

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.<%@taglib uri="/struts-tags" prefix="s"%>
05.<html>
06.<head>
07.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
08.<title>Registration Page</title>
09.<s:head />
10.<style type="text/css">
11.@import url(style.css);
12.</style>
13.</head>
14.<body>
15.<s:actionerror/>
16.<s:form action="addUser">
17.<s:hidden name="user.id" />
18.<s:textfield key="user.name" />
19.<s:password key="user.password" />
20.<s:select key="user.gender" list="{ 'Male', 'Female' }" headerKey=""
21.headerValue="Select" label="Gender" />
22.<s:select key="user.country" list="{ 'India', 'USA', 'UK' }" headerKey=""
23.headerValue="Select" label="Country" />
24.<s:textarea key="user.aboutYou" />
25.<s:checkbox key="user.mailingList"
26.label="Would you like to join our mailing list?" />
27.<s:submit />
28.</s:form>
29.
30.

```



```

31.<s:if test="userList.size() > 0">
32.<div class="content">
33.<table class="userTable" cellpadding="5px">
34.<tr class="even">
35.<th>Name</th>
36.<th>Gender</th>
37.<th>Country</th>
38.<th>About You</th>
39.<th>Mailing List</th>
40.</tr>
41.<s:iterator value="userList" status="userStatus">
42.<tr
43.class="<s:if test="#userStatus.odd == true
">odd</s:if><s:else>even</s:else>">
44.<td><s:property value="name" /></td>
45.<td><s:property value="gender" /></td>
46.<td><s:property value="country" /></td>
47.<td><s:property value="aboutYou" /></td>
48.<td><s:property value="mailingList" /></td>
49.</tr>
50.</s:iterator>
51.</table>
52.</div>
53.</s:if>
54.</body>
55.</html>

```

The key values should be specified in the `UserAction.properties` file and this file should be saved next to the `UserAction.java` file.

```

1.user.name=User Name
2.user.password=Password
3.user.aboutYou=About You
4.user.mailingList=Mailing List

```

The default validation messages will be generated using the field labels as prefix.

When you execute the example and submit the form without entering any values you will see the following validation messages.

The screenshot shows a web browser window titled "Registration Page". The address bar displays "http://localhost:8080/Struts2Example18/addUser.action". The page contains several form fields with red validation error messages:

- User Name - may not be null or empty**: The "User Name:" label is followed by an empty text input field.
- Password - length must be between 6 and 10**: The "Password:" label is followed by an empty text input field.
- Please select a gender**: The "Gender:" label is followed by a dropdown menu with "Select" and a downward arrow.
- Please select a country**: The "Country:" label is followed by a dropdown menu with "Select" and a downward arrow.
- About You - may not be null or empty**: The "About You:" label is followed by a text area with up and down arrows.

Below the text area, there is a checkbox labeled "Would you like to join our mailing list?". At the bottom right, there is a "Submit" button.

The validation message for the fields User Name, Password and About You has the field label before the default validation error message. This is because we specified the key values in the `UserAction.properties` file. If you only want the validation message to be displayed then don't specify an entry for that field in the properties file. Here the gender and the country fields have only the customized error messages and not the field labels. Everything else remains the same as the previous example ([Struts 2 Hibernate Integration](#)).

Struts 2 CRUD Example

In this example you will see how to perform Create, Read, Update and Delete (CRUD) operations. I will be explaining only the points that is not covered in the previous examples.

Let's get started, the screen shot of the example is shown below.

Registration Page

http://localhost:8080/Struts2Example19/listUser

User Name:

Gender: ☐ Male ☐ Female

Select a country:

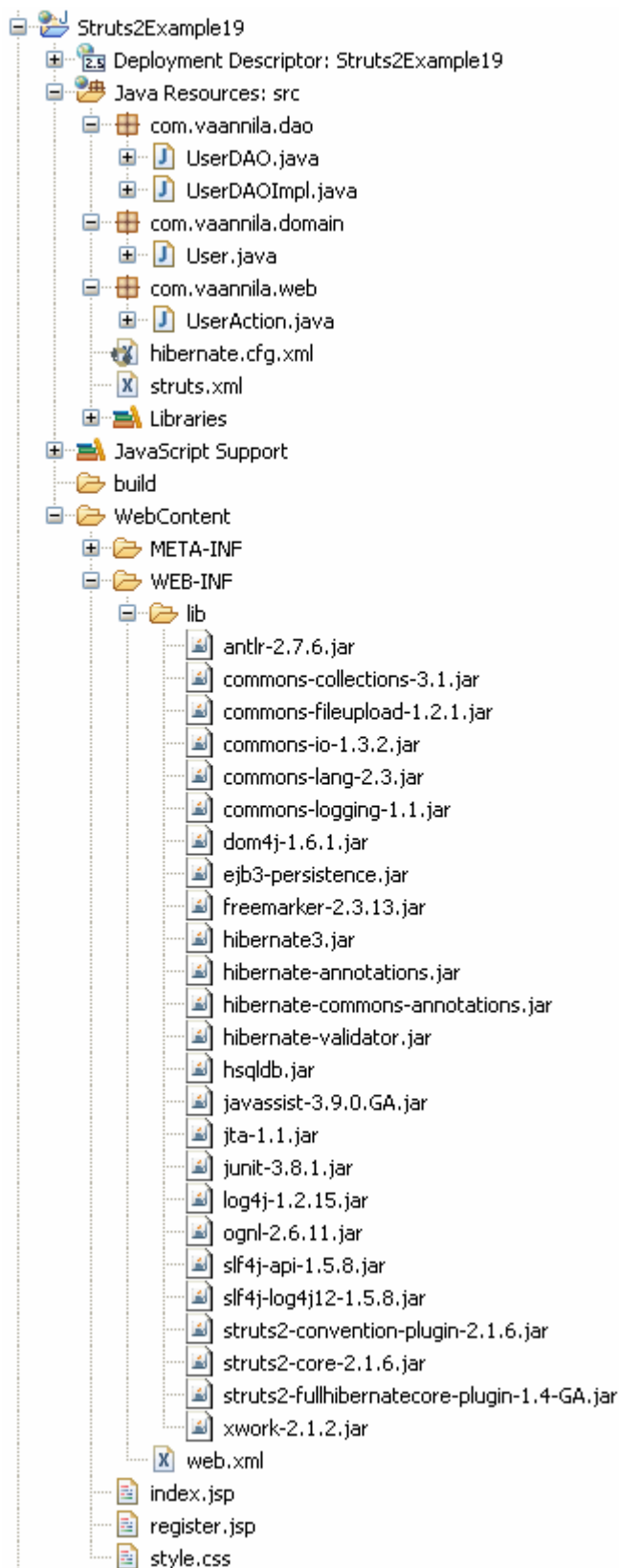
About You:

☐ Would you like to join our mailing list?

Name	Gender	Country	About You	Mailing List	Edit	Delete
Eswar	Male	India	Java programmer	true	Edit	Delete
Joy	Male	India	Software Engineer	false	Edit	Delete

You will have options to edit and delete in addition to the one we saw before. ([Struts 2 Hibernate Integration](#))

The directory structure of the example.



Let's see the flow from the back-end.

The `UserDAOImpl` has four methods to perform the various CRUD operations.

```
01.package com.vaannila.dao;
```

```
02.

03.import java.util.List;

04.

05.import org.hibernate.Session;

06.import org.hibernate.Transaction;

07.

08.import com.googlecode.s2hibernate.struts2.plugin.annotations.SessionTarget;

09.import com.googlecode.s2hibernate.struts2.plugin.annotations.TransactionTarget;

10.import com.vaannila.domain.User;

11.

12.public class UserDAOImpl implements UserDAO {

13.

14.    @SessionTarget

15.    Session session;

16.

17.    @TransactionTarget

18.    Transaction transaction;

19.

20.    /**

21.     * Used to save or update a user.

22.     */

23.    @Override

24.    public void saveOrUpdateUser(User user) {

25.        try {

26.            session.saveOrUpdate(user);
```

```
27.} catch (Exception e) {
28.transaction.rollback();
29.e.printStackTrace();
30.}
31.}
32.
33./**
34.* Used to delete a user.
35.* /
36.@Override
37.public void deleteUser(Long userId) {
38.try {
39.User user = (User) session.get(User.class, userId);
40.session.delete(user);
41.} catch (Exception e) {
42.transaction.rollback();
43.e.printStackTrace();
44.}
45.}
46.
47./**
48.* Used to list all the users.
49.* /
50.@SuppressWarnings("unchecked")
51.@Override
52.public List<User> listUser() {
53.List<User> courses = null;
54.try {
```

```

55.courses = session.createQuery("from User").list();

56.} catch (Exception e) {

57.e.printStackTrace();

58.}

59.return courses;

60.}

61.

62./**

63.* Used to list a single user by Id.

64.*/

65.@Override

66.public User listUserById(Long userId) {

67.User user = null;

68.try {

69.user = (User) session.get(User.class, userId);

70.} catch (Exception e) {

71.e.printStackTrace();

72.}

73.return user;

74.}

75.

76.}

```

The org.hibernate.Session and org.hibernate.Transaction objects are injected using the Full Hibernate Plug-in 1.4 GA.

The User domain object with the JPA annotations to create the USER table.

```

01.package com.vaannila.domain;

02.

03.import javax.persistence.Column;

```

```
04.import javax.persistence.Entity;

05.import javax.persistence.GeneratedValue;

06.import javax.persistence.Id;

07.import javax.persistence.Table;

08.

09.@Entity

10.@Table(name="USER")

11.public class User {

12.

13.private Long id;

14.private String name;

15.private String gender;

16.private String country;

17.private String aboutYou;

18.private Boolean mailingList;

19.

20.@Id

21.@GeneratedValue

22.@Column(name="USER_ID")

23.public Long getId() {

24.return id;

25.}

26.public void setId(Long id) {

27.this.id = id;

28.}

29.
```



```
30.@Column(name="USER_NAME")

31.public String getName() {

32.return name;

33.}

34.public void setName(String name) {

35.this.name = name;

36.}

37.

38.@Column(name="USER_GENDER")

39.public String getGender() {

40.return gender;

41.}

42.public void setGender(String gender) {

43.this.gender = gender;

44.}

45.

46.@Column(name="USER_COUNTRY")

47.public String getCountry() {

48.return country;

49.}

50.public void setCountry(String country) {

51.this.country = country;

52.}

53.

54.@Column(name="USER_ABOUT_YOU")

55.public String getAboutYou() {

56.return aboutYou;
```

```

57.}

58.public void setAboutYou(String aboutYou) {
59.this.aboutYou = aboutYou;
60.}

61.

62.@Column(name="USER_MAILING_LIST")

63.public Boolean getMailingList() {

64.return mailingList;

65.}

66.public void setMailingList(Boolean mailingList) {

67.this.mailingList = mailingList;

68.}

69.

70.}

```

Our UserAction implements ModelDriven interface, so the domain object User will be exposed as a model object.
Use ActionContext.getContext().get(ServletActionContext.HTTP_REQUEST) method to access the HttpServletRequest object in action.

```

01.package com.vaannila.web;

02.

03.import java.util.ArrayList;

04.import java.util.List;

05.

06.import javax.servlet.http.HttpServletRequest;

07.

08.import org.apache.struts2.ServletActionContext;

09.

10.import com.opensymphony.xwork2.ActionContext;

```

```
11.import com.opensymphony.xwork2.ActionSupport;

12.import com.opensymphony.xwork2.ModelDriven;

13.import com.vaannila.dao.UserDAO;

14.import com.vaannila.dao.UserDAOImpl;

15.import com.vaannila.domain.User;

16.

17.public class UserAction extends ActionSupport implements ModelDriven<User>
{

18.

19.private static final long serialVersionUID = -6659925652584240539L;

20.

21.private User user = new User();

22.private List<User> userList = new ArrayList<User>();

23.private UserDAO userDAO = new UserDAOImpl();

24.

25.@Override

26.public User getModel() {

27.return user;

28.}

29.

30./**

31.* To save or update user.

32.* @return String

33.* /

34.public String saveOrUpdate()

35.{
```

```
36.userDAO.saveOrUpdateUser(user);

37.return SUCCESS;

38.}

39.

40.**

41.* To list all users.
42.* @return String
43.* /

44.public String list()

45.{

46.userList = userDAO.listUser();

47.return SUCCESS;

48.}

49.

50.**

51.* To delete a user.
52.* @return String
53.* /

54.public String delete()

55.{

56.HttpServletRequest request = (HttpServletRequest)
ServletContext.getContext().get( ServletActionContext.HTTP_REQUEST);

57.userDAO.deleteUser(Long.parseLong( request.getParameter("id")));

58.return SUCCESS;

59.}

60.

61.**

62.* To list a single user by Id.
63.* @return String
```

```
64.*/

65.public String edit()

66.{

67.HttpServletRequest request = (HttpServletRequest)
ServletContext.getContext().get( ServletActionContext.HTTP_REQUEST);

68.user = userDao.listUserById(Long.parseLong(
request.getParameter("id")));

69.return SUCCESS;

70.}

71.

72.public User getUser() {

73.return user;

74.}

75.

76.public void setUser(User user) {

77.this.user = user;

78.}

79.

80.public List<User> getUserList() {

81.return userList;

82.}

83.

84.public void setUserList(List<User> userList) {

85.this.userList = userList;

86.}

87.

88.}
```

In the struts configuration file we have four different actions corresponding to the different CRUD operations. During the save, update and delete operations, we need to update the list of users displayed below so we redirect the result to the `listUser` action.

```
01.<!DOCTYPE struts PUBLIC
02."-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
03."http://struts.apache.org/dtds/struts-2.0.dtd">
04.
05.<struts>
06.<package name="default" extends="hibernate-default">
07.<action name="saveOrUpdateUser" method="saveOrUpdate" class="com.vaannila.
web.UserAction">
08.<result name="success" type="redirect">listUser</result>
09.</action>
10.<action name="listUser" method="list" class="com.vaannila.web.UserAction">
11.<result name="success">/register.jsp</result>
12.</action>
13.<action name="editUser" method="edit" class="com.vaannila.web.UserAction">
14.<result name="success">/register.jsp</result>
15.</action>
16.<action name="deleteUser" method="delete" class="com.vaannila.web.UserActi
on">
17.<result name="success" type="redirect">listUser</result>
18.</action>
19.</package>
20.</struts>
```

The hibernate.cfg.xml file contains the following configuration.

```
01.<?xml version="1.0" encoding="UTF-8"?>
02.<!DOCTYPE hibernate-configuration PUBLIC
03."-//Hibernate/Hibernate Configuration DTD 3.0//EN"
04."http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```

05.<hibernate-configuration>

06.<session-factory>

07.<property name="hibernate.connection.driver_class">
org.hsqldb.jdbcDriver </property>

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

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

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

11.<property name="connection.pool_size">1</property>

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

13.<property name="show_sql">true</property>

14.<property name="hbm2ddl.auto">create</property>

15.<mapping class="com.vaannila.domain.User" />

16.</session-factory>

17.</hibernate-configuration>

```

The deployment descriptor contains the following configuration.

```

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.su
n.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.<display-name>Struts2Example19</display-name>

04.<filter>

05.<filter-name>struts2</filter-name>

06.<filter-class>

07.org.apache.struts2.dispatcher.ng.filter. StrutsPrepareAndExecuteFilter
</filter-class>

08.</filter>

```

```
09.<filter-mapping>
10.<filter-name>struts2</filter-name>
11.<url-pattern>/*</url-pattern>
12.</filter-mapping>
13.
14.<welcome-file-list>
15.<welcome-file>index.jsp</welcome-file>
16.</welcome-file-list>
17.</web-app>
```

In the register.jsp page we use the Struts 2 tags to display the form elements.

```
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.<%@taglib uri="/struts-tags" prefix="s"%>
05.<html>
06.<head>
07.<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
08.<title>Registration Page</title>
09.<s:head />
10.<style type="text/css">
11.@import url(style.css);
12.</style>
13.</head>
14.<body>
15.<s:form action="saveOrUpdateUser">
16.<s:push value="user">
17.<s:hidden name="id" />
18.<s:textfield name="name" label="User Name" />
```



```
19.<s:radio name="gender" label="Gender"

20.list="{ 'Male', 'Female' }" />

21.<s:select name="country" list="{ 'India', 'USA', 'UK' }"

22.headerKey="" headerValue="Select"

23.label="Select a country" />

24.<s:textarea name="aboutYou" label="About You" />

25.<s:checkbox name="mailingList"

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

27.<s:submit />

28.</s:push>

29.</s:form>

30.

31.<s:if test="userList.size() > 0">

32.<div class="content">

33.<table class="userTable" cellpadding="5px">

34.<tr class="even">

35.<th>Name</th>

36.<th>Gender</th>

37.<th>Country</th>

38.<th>About You</th>

39.<th>Mailing List</th>

40.<th>Edit</th>

41.<th>Delete</th>

42.</tr>

43.<s:iterator value="userList" status="userStatus">

44.<tr
```

```

45.class="<s:if test="#userStatus.odd == true ">odd</s:if>
<s:else>even</s:else>">

46.<td><s:property value="name" /></td>

47.<td><s:property value="gender" /></td>

48.<td><s:property value="country" /></td>

49.<td><s:property value="aboutYou" /></td>

50.<td><s:property value="mailingList" /></td>

51.<td>

52.<s:url id="editURL" action="editUser">

53.<s:param name="id" value="%{id}"></s:param>

54.</s:url>

55.<s:a href="%{editURL}">Edit</s:a>

56.</td>

57.<td>

58.<s:url id="deleteURL" action="deleteUser">

59.<s:param name="id" value="%{id}"></s:param>

60.</s:url>

61.<s:a href="%{deleteURL}">Delete</s:a>

62.</td>

63.</tr>

64.</s:iterator>

65.</table>

66.</div>

67.</s:if>

68.</body>

69.</html>

```

The **push** tag is used to move the object to the top of the `ValueStack`. During add operation we will refer to the model object `User` exposed by the `ModelDriven` interface, in this case the push tag is not necessary. But during update operation we refer to the

JavaBean property `user` that is returned by the `listUserById()` method, now the `push` tag will be useful, it pushes the `User` object to the top of the `ValueStack` so we need not use the second-level OGNL expression language like `user.name` to access the domain object properties.

```
1.public String edit()  
  
2.{  
  
3.HttpServletRequest request = (HttpServletRequest)  
  ActionContext.getContext().get( ServletActionContext.HTTP_REQUEST);  
  
4.user = userDao.listUserById(Long.parseLong(request.getParameter("id")));  
  
5.return SUCCESS;  
  
6.}
```

The **url** tag is used to create a new URL. Along with the URL we append the `id` value, this can be done using the `param` tab. In the OGNL expression language we use `"%{"` as the escape sequence to refer a value on the `ActionContext`.