The struts framework was initially created by **Craig McClanahan** and donated to Apache Foundation in May, 2000 and Struts 1.0 was released in June 2001.

The current stable release of Struts is Struts 2.3.16.1 in March 2, 2014.

The Struts 2 provides supports to POJO based actions, Validation Support, AJAX Support, Integration support to various frameworks such as Hibernate, Spring, Tiles etc, support to various result types such as Freemarker, Velocity, JSP etc.

Struts 2 Features Tutorial

1. [Struts 2 Features Tutorial](https://www.javatpoint.com/struts-2-features-tutorial)

Struts 2 provides many features that were not in struts 1. The **important features** of struts 2 framework are as follows:

1. Configurable MVC components
2. POJO based actions
3. AJAX support
4. Integration support
5. Various Result Types
6. Various Tag support
7. Theme and Template support

1) Configurable MVC components

In struts 2 framework, we provide all the components (view components and action) information in struts.xml file. If we need to change any information, we can simply change it in the xml file.

2) POJO based actions

In struts 2, action class is POJO (Plain Old Java Object) i.e. a simple java class. Here, you are not forced to implement any interface or inherit any class.

3) AJAX support

Struts 2 provides support to ajax technology. It is used to make asynchronous request i.e. it doesn't block the user. It sends only required field data to the server side not all. So it makes the performance fast.

4) Integration Support

We can simply integrate the struts 2 application with hibernate, spring, tiles etc. frameworks.

5) Various Result Types

We can use JSP, freemarker, velocity etc. technologies as the result in struts 2.

6) Various Tag support

Struts 2 provides various types of tags such as UI tags, Data tags, control tags etc to ease the development of struts 2 application.

7) Theme and Template support

Struts 2 provides three types of theme support: xhtml, simple and css\_xhtml. The xhtml is default theme of struts 2. Themes and templates can be used for common look and feel.

Model 1 and Model 2 (MVC) Architecture

1. [Model 1 and Model 2 (MVC) Architecture](https://www.javatpoint.com/model-1-and-model-2-mvc-architecture)
2. [Model 1 Architecture](https://www.javatpoint.com/model-1-and-model-2-mvc-architecture#m1)
3. [Model 2 (MVC) Architecture](https://www.javatpoint.com/model-1-and-model-2-mvc-architecture#m2)

Before developing the web applications, we need to have idea about design models. There are two types of programming models (design models)

1. Model 1 Architecture
2. Model 2 (MVC) Architecture

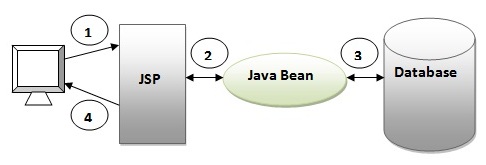
Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

**Servlet** was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

**Problem in Servlet technology** Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

**JSP** overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which show the flow of the model1 architecture.

1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

Advantage of Model 1 Architecture

* Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

* **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
* **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
* **Hard to extend** It is better for small applications but not for large applications.

Model 2 (MVC) Architecture

Model 2 is based on the MVC (Model View Controller) design pattern. The MVC design pattern consists of three modules model, view and controller.

**Model** The model represents the state (data) and business logic of the application.

**View** The view module is responsible to display data i.e. it represents the presentation.

**Controller** The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.



Advantage of Model 2 (MVC) Architecture

* **Navigation control is centralized** Now only controller contains the logic to determine the next page.
* **Easy to maintain**
* **Easy to extend**
* **Easy to test**
* **Better separation of concerns**

Disadvantage of Model 2 (MVC) Architecture

* We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

Solution of Model 2 Architecture: Configurable MVC Components

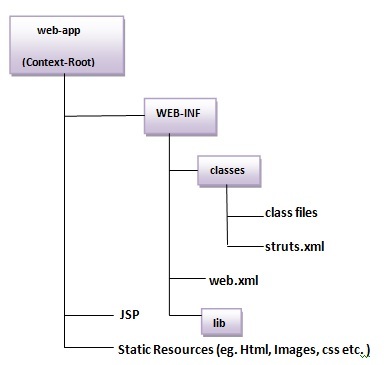
It uses the declarative approach for defining view components, request mapping etc. It resolves the problem of Model 2 architecture. The Struts framework provides the configurable MVC support. In struts 2, we define all the action classes and view components in struts.xml file.

In this example, we are creating the struts 2 example without IDE. We can simply create the struts 2 application by following these simple steps:

1. Create the directory structure
2. Create input page (index.jsp)
3. Provide the entry of Controller in (web.xml) file
4. Create the action class (Product.java)
5. Map the request with the action in (struts.xml) file and define the view components
6. Create view components (welcome.jsp)
7. load the jar files
8. start server and deploy the project

1) Create the directory structure

The directory structure of struts 2 is same as servlet/JSP. Here, struts.xml file must be located in the classes folder.



2) Create input page (index.jsp)

This jsp page creates a form using struts UI tags. To use the struts UI tags, you need to specify uri /struts-tags. Here, we have used s:form to create a form, s:textfield to create a text field, s:submit to create a submit button.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <s:form action="product">
3. <s:textfield name="id" label="Product Id"></s:textfield>
4. <s:textfield name="name" label="Product Name"></s:textfield>
5. <s:textfield name="price" label="Product Price"></s:textfield>
6. <s:submit value="save"></s:submit>
7. </s:form>

3) Provide the entry of Controller in (web.xml) file

In struts 2, **StrutsPrepareAndExecuteFilter** class works as the controller. As we know well, struts 2 uses filter for the controller. It is implicitly provided by the struts framework.

**web.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <web-app>
3. <filter>
4. <filter-name>struts2</filter-name>
5. <filter-**class**>
6. org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
7. </filter-**class**>
8. </filter>
9. <filter-mapping>
10. <filter-name>struts2</filter-name>
11. <url-pattern>/\*</url-pattern>
12. </filter-mapping>
13. </web-app>

4) Create the action class (Product.java)

This is simple bean class. In struts 2, action is POJO (Plain Old Java Object). It has one extra method **execute** i.e. invoked by struts framework by default.

**Product.java**

1. **package** com.javatpoint;
3. **public** **class** Product {
4. **private** **int** id;
5. **private** String name;
6. **private** **float** price;
7. **public** **int** getId() {
8. **return** id;
9. }
10. **public** **void** setId(**int** id) {
11. **this**.id = id;
12. }
13. **public** String getName() {
14. **return** name;
15. }
16. **public** **void** setName(String name) {
17. **this**.name = name;
18. }
19. **public** **float** getPrice() {
20. **return** price;
21. }
22. **public** **void** setPrice(**float** price) {
23. **this**.price = price;
24. }
26. **public** String execute(){
27. **return** "success";
28. }
29. }

5) Map the request in (struts.xml) file and define the view components

It is the important file from where struts framework gets information about the action and decides which result to be invoked. Here, we have used many elements such as struts, package, action and result.

**struts** element is the root elements of this file. It represents an application.

**package** element is the sub element of struts. It represents a module of the application. It generally extends the **struts-default** package where many interceptors and result types are defined.

**action** element is the sub element of package. It represents an action to be invoked for the incoming request. It has name, class and method attributes. If you don't specify name attribute by default execute() method will be invoked for the specified action class.

**result** element is the sub element of action. It represents an view (result) that will be invoked. Struts framework checks the string returned by the action class, if it returns success, result page for the action is invoked whose name is success or has no name. It has **name** and **type** attributes. Both are optional. If you don't specify the result name, by default success is assumed as the result name. If you don't specify the type attribute, by default **dispatcher** is considered as the default result type. We will learn about result types later.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
5. <**package** name="default" **extends**="struts-default">
7. <action name="product" **class**="com.javatpoint.Product">
8. <result name="success">welcome.jsp</result>
9. </action>
11. </**package**>
12. </struts>

6) Create view components (welcome.jsp)

It is the view component the displays information of the action. Here, we are using struts tags to get the information.

The s:property tag returns the value for the given name, stored in the action object.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Product Id:<s:property value="id"/><br/>
4. Product Name:<s:property value="name"/><br/>
5. Product Price:<s:property value="price"/><br/>

7) Load the jar files

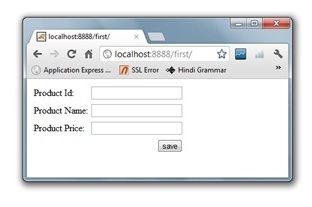
To run this application, you need to have the struts 2 jar files. Here, we are providing all the necessary jar files for struts 2. Download it and put these jar files in the lib folder of your project.

[download the struts2 jar files](https://static.javatpoint.com/src/st/struts2jars.zip)

8) start server and deploy the project

Finally, start the server and deploy the project and access it.

[download this example](https://static.javatpoint.com/src/st/firststruts.zip)


Struts 2 Interceptors Tutorial

1. [Struts 2 Interceptors Tutorial](https://www.javatpoint.com/struts-2-interceptors-tutorial)
2. [Advantage of interceptors](https://www.javatpoint.com/struts-2-interceptors-tutorial)
3. [Struts 2 default interceptors](https://www.javatpoint.com/struts-2-interceptors-tutorial#default)

Interceptor is an object that is invoked at the preprocessing and postprocessing of a request. In Struts 2, interceptor is used to perform operations such as validation, exception handling, internationalization, displaying intermediate result etc.

Advantage of interceptors

**Pluggable** If we need to remove any concern such as validation, exception handling, logging etc. from the application, we don't need to redeploy the application. We only need to remove the entry from the struts.xml file.

Struts 2 default interceptors

There are many interceptors provided by struts 2 framework. We have option to create our own interceptors. The struts 2 default interceptors are as follows:

**1) alias** It converts similar parameters that have different names between requests.

**2) autowiring**

**3) chain** If it is used with chain result type, it makes the properties of previous action available in the current action.

**4) checkbox** It is used to handle the check boxes in the form. By this, we can detect the unchecked checkboxes.

**5) cookie** It adds a cookie to the current action.

**6) conversionError** It adds conversion errors to the action's field errors.

**7) createSession** It creates and HttpSession object if it doesn't exists.

**8) clearSession** It unbinds the HttpSession object.

**9) debugging** It provides support of debugging.

**10) externalRef**

**11) execAndWait** It sends an intermediate waiting page for the result.

**12) exception** It maps exception to a result.

**13) fileUpload** It provides support to file upload in struts 2.

**14) i18n** It provides support to internationalization and localization.

**15) jsonValidation** It provides support to asynchronous validation.

**16) logger** It outputs the action name.

**17) store** It stores and retrieves action messages, action errors or field errors for action that implements ValidationAware interface.

**18) modelDriven** It makes other model object as the default object of valuestack.

**19) scopedModelDriven** It is similar to ModelDriven but works for action that implements ScopedModelDriven.

**20) params** It populates the action properties with the request parameters.

**21) actionMappingParams**

**22) prepare** It performs preparation logic if action implements Preparable interface.

**23) profiling** It supports action profiling.

**24) roles** It supports role-based action.

**25) scope** It is used to store the action state in the session or application scope.

**26) servletConfig** It provides access to maps representing HttpServletRequest and HttpServletResponse.

**27) sessionAutowiring**

**28) staticParams** It maps static properties to action properties.

**29) timer** It outputs the time needed to execute an action.

**30) token** It prevents duplication submission of request.

**31) tokenSession** It prevents duplication submission of request.

**32) validation** It provides support to input validation.

**33) workflow** It calls the validate method of action class if action class implements Validateable interface.

**34) annotationWorkflow**

**35) multiselect**

Struts 2 ValueStack Tutorial

1. [Struts 2 ValueStack Tutorial](https://www.javatpoint.com/struts-2-ValueStack-tutorial)
2. [ValueStack Interface](https://www.javatpoint.com/struts-2-ValueStack-tutorial)
3. [Methods of ValueStack interface](https://www.javatpoint.com/struts-2-ValueStack-tutorial#methods)

A valueStack is simply a stack that contains application specific objects such as action objects and other model object.

At the execution time, action is placed on the top of the stack.

We can put objects in the valuestack, query it and delete it.

ValueStack Interface

The struts 2 framework provides an interface to deal with valuestack. It provides many useful methods.

Methods of ValueStack interface

There are many methods in ValueStack interface. The commonly used methods are as follows:

**public String findString(String expr)** finds the string by evaluating the given expression.

**public Object findValue(String expr)** finds the value by evaluating the specified expression.

**public Object findValue(String expr, Class c)** finds the value by evaluating the specified expression.

**public Object peek()** It returns the object located on the top of the stack.

**public Object pop()** It returns the object located on the top of the stack and removes it.

**public void push(Object o)** It puts the object on the top of the stack.

**public void set(String key, Object value)** It sets the object on the stack with the given key. It can be get by calling the findValue(key) method.

**public int size()** It returns the number of objects from the stack.

Struts 2 ActionContext Tutorial

1. [Struts 2 ActionContext Tutorial](https://www.javatpoint.com/struts-2-ActionContext-tutorial)

The ActionContext is a container of objects in which action is executed. The values stored in the ActionContext are unique per thread (i.e. ThreadLocal). So we don't need to make our action thread safe.

We can get the reference of ActionContext by calling the getContext() method of ActionContext class. It is a static factory method. For example:

1. ActionContext context = ActionContext.getContext();

The **ActionInvocation** represents the execution state of an action. It holds the action and interceptors objects.

ActionInvocation Interface

The struts framework provides **ActionInvocation interface** to deal with ActionInvocation. It provides many methods, some of them can be used to get the instance of ValueStack, ActionProxy, ActionContext, Result etc.

Methods of ActionInvocation Interface

The commonly used methods of ActionInvocation interface are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public ActionContext getInvocationContext() | returns the ActionContext object associated with the ActionInvocation. |
| 2) | public ActionProxy getProxy() | returns the ActionProxy instance holding this ActionInvocation. |
| 3) | public ValueStack getStack() | returns the instance of ValueStack. |
| 4) | public Action getAction() | returns the instance of Action associated with this ActionInvocation. |
| 5) | public void invoke() | invokes the next resource in processing this ActionInvocation. |
| 6) | public Result getResult() | returns the instance of Result. |

Struts 2 OGNL Tutorial

1. [Struts 2 OGNL Tutorial](https://www.javatpoint.com/struts-2-OGNL-tutorial)

The **Object Graph Navigation Language** (OGNL) is an expression language. It simplifies the accessibility of data stored in the ActionContext.

The struts framework sets the **ValueStack** as the root object of OGNL. Notice that action object is pushed into the ValueStack. We can direct access the action property.

1. <s:property value="username"/>

Here, username is the property key.

The struts framework places other objects in ActionContext also e.g. map representing the **request**, **session**, **application** scopes.

To get these values i.e. not the action property, we need to use # notation. For example to get the data from session scope, we need to use #session as given in the following example:

1. <s:property name="#session.username"/>

(or)

1. <s:property name="#session['username']"/>

Struts 2 Architecture and Flow

1. [Struts 2 Architecture](https://www.javatpoint.com/struts-2-architecture-and-flow)
2. [Struts 2 Basic Architecture](https://www.javatpoint.com/struts-2-architecture-and-flow)
3. [Struts 2 Standard Architecture](https://www.javatpoint.com/struts-2-architecture-and-flow)

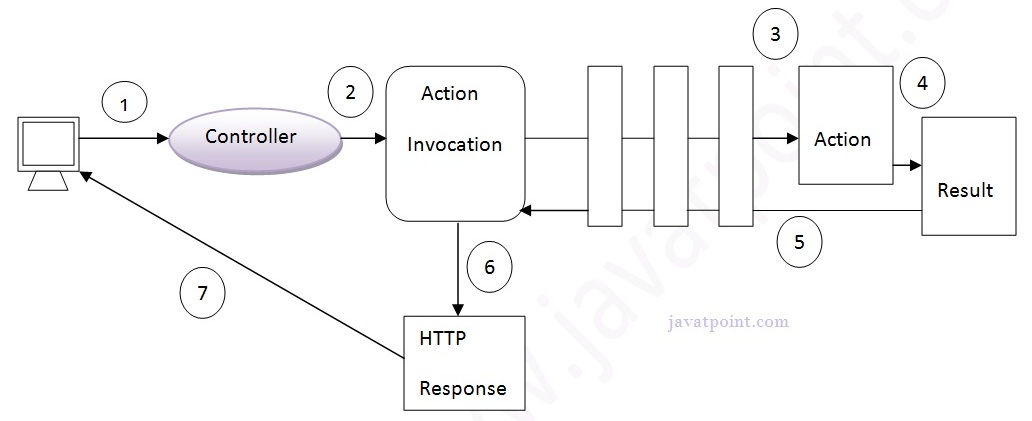
The **architecture and flow of struts 2 application**, is combined with many components such as Controller, ActionProxy, ActionMapper, Configuration Manager, ActionInvocation, Inerceptor, Action, Result etc.

Here, we are going to understand the struts flow by 2 ways:

1. struts 2 basic flow
2. struts 2 standard architecture and flow provided by apache struts

Struts 2 basic flow

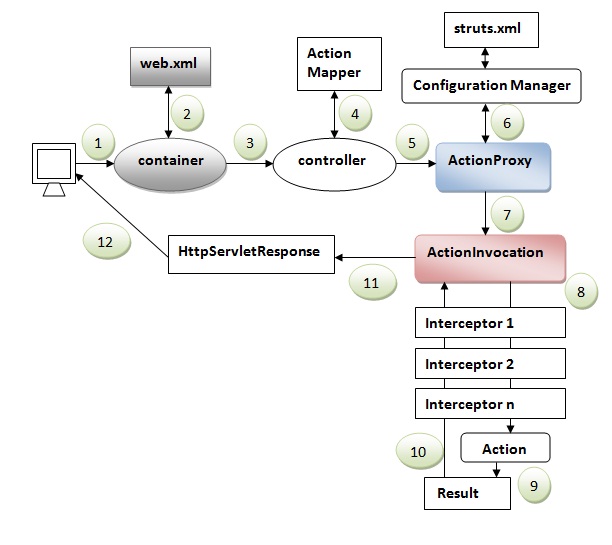
Let's try to understand the basic flow of struts 2 application by this simple figure:



1. User sends a request for the action
2. Controller invokes the ActionInvocation
3. ActionInvocation invokes each interceptors and action
4. A result is generated
5. The result is sent back to the ActionInvocation
6. A HttpServletResponse is generated
7. Response is sent to the user

Struts 2 standard flow (Struts 2 architecture)

Let's try to understand the standard architecture of struts 2 application by this simple figure:



1. User sends a request for the action
2. Container maps the request in the web.xml file and gets the class name of controller.
3. Container invokes the controller (StrutsPrepareAndExecuteFilter or FilterDispatcher). Since struts2.1, it is StrutsPrepareAndExecuteFilter. Before 2.1 it was FilterDispatcher.
4. Controller gets the information for the action from the ActionMapper
5. Controller invokes the ActionProxy
6. ActionProxy gets the information of action and interceptor stack from the configuration manager which gets the information from the struts.xml file.
7. ActionProxy forwards the request to the ActionInvocation
8. ActionInvocation invokes each interceptors and action
9. A result is generated
10. The result is sent back to the ActionInvocation
11. A HttpServletResponse is generated
12. Response is sent to the user

[**next>>**](https://www.javatpoint.com/struts-2-configuration-file)[**<<prev**](https://www.javatpoint.com/struts-2-architecture-and-flow)

Struts 2 Action

1. [Struts 2 Action](https://www.javatpoint.com/struts-2-action)
2. [Action Interface](https://www.javatpoint.com/struts-2-action)
3. [ActionSupport class](https://www.javatpoint.com/struts-2-action)

In struts 2, action class is **POJO** (Plain Old Java Object).

POJO means you are not forced to implement any interface or extend any class.

Generally, **execute** method should be specified that represents the business logic. The simple action class may look like:

**Welcome.java**

1. **package** com.javatpoint;
2. **public** **class** Welcome {
3. **public** String execute(){
4. **return** "success";
5. }
6. }

Action Interface

A convenient approach is to implement the **com.opensymphony.xwork2.Action** interface that defines 5 constants and one execute method.

5 Constants of Action Interface

Action interface provides 5 constants that can be returned form the action class. They are:

1. **SUCCESS** indicates that action execution is successful and a success result should be shown to the user.
2. **ERROR** indicates that action execution is failed and a error result should be shown to the user.
3. **LOGIN** indicates that user is not logged-in and a login result should be shown to the user.
4. **INPUT** indicates that validation is failed and a input result should be shown to the user again.
5. **NONE** indicates that action execution is successful but no result should be shown to the user.

Let's see what values are assigned to these constants:

1. **public** **static** **final** String SUCCESS = "success";
2. **public** **static** **final** String ERROR = "error";
3. **public** **static** **final** String LOGIN  = "login";
4. **public** **static** **final** String INPUT = "input";
5. **public** **static** **final** String NONE = "none";

Method of Action Interface

Action interface contains only one method execute that should be implemented overridden by the action class even if you are not forced.

1. **public** String execute();

Example of Struts Action that implements Action interface

If we implement the Action interface, we can directly use the constants instead of values.

**Welcome.java**

1. **package** com.javatpoint;
2. **import** com.opensymphony.xwork2.Action;
3. **public** **class** Welcome **implements** Action{
4. **public** String execute(){
5. **return** SUCCESS;
6. }
7. }

ActionSupport class

It is a convenient class that implements many interfaces such as Action, Validateable, ValidationAware, TextProvider, LocaleProvider and Serializable . So it is mostly used instead of Action.

Example of Struts Action that extends ActionSupport class

Let's see the example of Action class that extends the ActionSupport class.

**Welcome.java**

1. **package** com.javatpoint;
2. **import** com.opensymphony.xwork2.ActionSupport;
3. **public** **class** Welcome **extends** ActionSupport{
4. **public** String execute(){
5. **return** SUCCESS;
6. }
7. }

The struts application contains two main configuration files **struts.xml** file and **struts.properties** file.

The struts.properties file is used to override the default values of **default.xml** file provided by struts framework. So it is not mandatory. Mostly, you will not use struts.properties file. We will learn about it later.

Here, we are going to learn all about struts.xml file. First of all let us see the simple example of struts.xml file

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
5. <**package** name="default" **extends**="struts-default">
7. <action name="product" **class**="com.javatpoint.Product">
8. <result name="success">welcome.jsp</result>
9. </action>
11. </**package**>
12. </struts>

1) package element

We can easily divide our struts application into sub modules. The package element specifies a module. You can have one or more packages in the struts.xml file.

Attributes of package element

* **name** name is must for defining any package.
* **namespace** It is an optional attribute of package. If namespace is not present, / is assumed as the default namespace. In such case, to invoke the action class, you need this URI:
  1. /actionName.action

If you specify any namespace, you need this URI:

* 1. /namespacename/actionName.action
* **extends** The package element mostly extends the **struts-default** package where interceptors and result types are defined. If you extend struts-default, all the actions of this package can use the interceptors and result-types defined in the **struts-default.xml** file.

2) action element

The action is the subelement of package and represents an action.

Attributes of action element

* **name** name is must for defining any action.
* **class** class is the optional attribute of action. If you omit the class attribute, **ActionSupport** will be considered as the default action. A simple action may be as:
  1. <action name="product">
* **method** It is an optional attribute. If you don't specify method attribute, **execute** method will be considered as the method of action class. So this code:
  1. <action name="product" **class**="com.javatpoint.Product">

will be same as:

* 1. <action name="product" **class**="com.javatpoint.Product" method="execute">

If you want to invoke a particular method of the action, you need to use method attribute.

3) result element

It is the sub element of action that specifies where to forward the request for this action.

Attributes of result element

* **name** is the optional attribute. If you omit the name attribute, success is assumed as the default result name.
* **type** is the optional attribute. If you omit the type attribute, dispatcher is assumed as the default result type.

Other elements

There are many other elements also such as **global-exception-mappings**, **global-results**, **include** etc. It will be discussed later.

Struts 2 Tutorial: Multiple Configuration File Example

1. [Multiple Configuration File Example](https://www.javatpoint.com/struts-2-tutorial-multiple-configuration-file-example)
2. [Example to create multiple configuration files](https://www.javatpoint.com/struts-2-tutorial-multiple-configuration-file-example)

For a large application, it is better to use multiple configuration file that one so that it may be easy to manage the application.

We can create many configuration files but we need to define it in the struts.xml file. The **include**sub-element of **struts** is used to define the supported configuration file.

Example to create multiple configuration files

Let's see the simple example to define multiple configuration files.

1) Define entry of multiple configuration files in struts.xml

In this struts.xml file, we have defined 2 configuration files struts-first.xml file and struts-second.xml file.

The struts-first.xml file is located inside the first directory and struts-second.xml file is located inside the second directory.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
6. <**package** name="default1" namespace="/" **extends**="struts-default">
7. </**package**>
9. <include file="first/struts-first.xml"></include>
10. <include file="second/struts-second.xml"></include>
12. </struts>

2) Create configuration files

Let's create two configuration files that defines action with results.

**struts-first.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
6. <**package** name="first" namespace="/first" **extends**="struts-default">
8. <action name="hello"  **class**="com.javatpoint.Welcome">
9. <result>welcome.jsp</result>
10. </action>
11. </**package**>
13. </struts>

**struts-second.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
5. <**package** name="second" namespace="/second" **extends**="struts-default">
6. <action name="hello"  **class**="com.javatpoint.Welcome">
7. <result>welcome.jsp</result>
8. </action>
9. </**package**>
10. </struts>

Other Required Resources

To run this application, we need following resources:

1. index.jsp
2. Welcome.java
3. 2 View Components

index.jsp

This jsp page creates two links.

**index.jsp**

1. <a href="first/hello">first namespace</a>|
2. <a href="second/hello">second namespace</a>

Action class

It is the simple Action class containing only execute method.

**Welcome.java**

1. **package** com.javatpoint;
3. **public** **class** Welcome {
4. **public** String execute(){
5. **return** "success";
6. }
7. }

2 view components

The name of 2 view components are same i.e. welcome.jsp but both are located differently.

**welcome.jsp**

It is located inside the first directory under root.

1. <h1>Welcome to first namespace</h1>
2. </pre></div>
4. <strong>welcome.jsp</strong>
5. <p>It is located inside the second directory under root.</p>
6. <div **class**="codeblock"><pre name="code" **class**="java" >
7. <h1>Welcome to second namespace</h1>

Struts 2 Multiple Namespace Example

1. [Struts 2 Multiple Namespace Example](https://www.javatpoint.com/struts-2-tutorial-multiple-namespace-example)
2. [Define multiple namespaces in struts.xml](https://www.javatpoint.com/struts-2-tutorial-multiple-namespace-example)

We can define multiple namespaces in struts.xml file by the namespace attribute of package element. As we know, default namespace is / (root).

Let's see the simple example to define multiple namespaces in struts.xml file.

Define multiple namespaces in struts.xml

This struts.xml file contains three packages with different names and namespaces.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
6. <**package** name="default1" namespace="/" **extends**="struts-default">
7. <action name="hello"  **class**="com.javatpoint.Welcome">
8. <result>welcome.jsp</result>
9. </action>
10. </**package**>
12. <**package** name="default2" namespace="/first" **extends**="struts-default">
13. <action name="hello"  **class**="com.javatpoint.Welcome">
14. <result>welcome.jsp</result>
15. </action>
16. </**package**>
18. <**package** name="default3" namespace="/second" **extends**="struts-default">
19. <action name="hello"  **class**="com.javatpoint.Welcome">
20. <result>welcome.jsp</result>
21. </action>
22. </**package**>
24. </struts>

Other required resources

We need some other required files to understand the full example of multiple namespaces.

* index.jsp
* action class (Welcome.java)
* 3 view components

1) Create index.jsp

This jsp page provides three links.

**index.jsp**

1. <a href="hello">root namespace</a>|
2. <a href="first/hello">first namespace</a>|
3. <a href="second/hello">second namespace</a>

2) Create the action class

It is the simple action class containing execute method only.

**Welcome.java**

1. **package** com.javatpoint;
3. **public** **class** Welcome {
4. **public** String execute(){
5. **return** "success";
6. }
7. }

3) 3 view components

Here name of all the 3 view components are same but there location are different.

**welcome.jsp**

It must be located in the root directory.

1. <h1>Welcome to root namespace</h1>

**welcome.jsp**

It must be located in the first directory under the root directory.

1. <h1>Welcome to first namespace</h1>

**welcome.jsp**

It must be located in the second directory under the root directory.

1. <h1>Welcome to second namespace</h1>

Struts 2 Interceptors Tutorial

1. [Struts 2 Interceptors Tutorial](https://www.javatpoint.com/struts-2-interceptors-tutorial)
2. [Advantage of interceptors](https://www.javatpoint.com/struts-2-interceptors-tutorial)
3. [Struts 2 default interceptors](https://www.javatpoint.com/struts-2-interceptors-tutorial#default)

Interceptor is an object that is invoked at the preprocessing and postprocessing of a request. In Struts 2, interceptor is used to perform operations such as validation, exception handling, internationalization, displaying intermediate result etc.

Advantage of interceptors

**Pluggable** If we need to remove any concern such as validation, exception handling, logging etc. from the application, we don't need to redeploy the application. We only need to remove the entry from the struts.xml file.

Struts 2 default interceptors

There are many interceptors provided by struts 2 framework. We have option to create our own interceptors. The struts 2 default interceptors are as follows:

**1) alias** It converts similar parameters that have different names between requests.

**2) autowiring**

**3) chain** If it is used with chain result type, it makes the properties of previous action available in the current action.

**4) checkbox** It is used to handle the check boxes in the form. By this, we can detect the unchecked checkboxes.

**5) cookie** It adds a cookie to the current action.

**6) conversionError** It adds conversion errors to the action's field errors.

**7) createSession** It creates and HttpSession object if it doesn't exists.

**8) clearSession** It unbinds the HttpSession object.

**9) debugging** It provides support of debugging.

**10) externalRef**

**11) execAndWait** It sends an intermediate waiting page for the result.

**12) exception** It maps exception to a result.

**13) fileUpload** It provides support to file upload in struts 2.

**14) i18n** It provides support to internationalization and localization.

**15) jsonValidation** It provides support to asynchronous validation.

**16) logger** It outputs the action name.

**17) store** It stores and retrieves action messages, action errors or field errors for action that implements ValidationAware interface.

**18) modelDriven** It makes other model object as the default object of valuestack.

**19) scopedModelDriven** It is similar to ModelDriven but works for action that implements ScopedModelDriven.

**20) params** It populates the action properties with the request parameters.

**21) actionMappingParams**

**22) prepare** It performs preparation logic if action implements Preparable interface.

**23) profiling** It supports action profiling.

**24) roles** It supports role-based action.

**25) scope** It is used to store the action state in the session or application scope.

**26) servletConfig** It provides access to maps representing HttpServletRequest and HttpServletResponse.

**27) sessionAutowiring**

**28) staticParams** It maps static properties to action properties.

**29) timer** It outputs the time needed to execute an action.

**30) token** It prevents duplication submission of request.

**31) tokenSession** It prevents duplication submission of request.

**32) validation** It provides support to input validation.

**33) workflow** It calls the validate method of action class if action class implements Validateable interface.

**34) annotationWorkflow**

**35) multiselect**

Struts 2 Custom Interceptor Example Tutorial

1. [Struts 2 Custom Interceptor Tutorial](https://www.javatpoint.com/struts-2-custom-interceptor-example-tutorial)
2. [Example to create custom interceptor in struts 2](https://www.javatpoint.com/ex)

In struts 2, we can create the custom interceptor by implementing the Interceptor interface in a class and overriding its three life cycle method.

For creating the custom interceptor, **Interceptor** interface must be implemented. It has three methods:

1. **public void init()** It is invoked only once and used to initialize the interceptor.
2. **public String intercept(ActionInvocation ai)** It is invoked at each request, it is used to define the request processing logic. If it returns string, result page will be invoked, if it returns invoke() method of ActionInvocation interface, next interceptor or action will be invoked.
3. **public void destroy()** It is invoked only once and used to destroy the interceptor.

Interceptor can change the flow of the application by returning the string.

Example to create custom interceptor in struts 2

In this example, we are going to create custom interceptor that converts request processing data into uppercase letter.

You need to follow **2 steps** to create custom interceptor

1. **Create an interceptor (must implement Interceptor interface)**
2. **Define the entry of interceptor in the struts.xml file**

1) Create an interceptor (must implement Interceptor interface)

By this interceptor, we are converting the name property of action class into uppercase letter.

The **getStack()** method of ActionInvocation returns the reference of ValueStack.

We are getting the value set in the name property by **findString** method of ValueStack.

The **set** method of ValueStack sets the name property by the specified value. In such case, we are converting the value of name property into uppercase letter and storing it into the valuestack.

The **invoke** method of ActionInvocation returns the information of next resource.

**MyInterceptor.java**

1. **package** com;
2. **import** com.opensymphony.xwork2.ActionInvocation;
3. **import** com.opensymphony.xwork2.interceptor.Interceptor;
4. **import** com.opensymphony.xwork2.util.ValueStack;
6. **public** **class** MyInterceptor **implements** Interceptor{
8. **public** **void** init() {}
9. **public** String intercept(ActionInvocation ai) **throws** Exception {
10. ValueStack stack=ai.getStack();
11. String s=stack.findString("name");
13. stack.set("name",s.toUpperCase());
14. **return** ai.invoke();
15. }
16. **public** **void** destroy() {}
17. }

2) Define the entry of interceptor in the struts.xml file

To define the interceptor, we need to declare an interceptor first. The **interceptors** element of package is used to specify interceptors. The interceptor element of interceptors is used to define the custom interceptor. Here, we are defining the custom interceptor by upper.

The **interceptor-ref** subelement of action specifies the interceptor that will be applied for this action. Here, we are specifying the defaultstack interceptors and upper interceptor.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
5. <**package** name="abc" **extends**="struts-default">
7. <interceptors>
8. <interceptor name="upper" **class**="com.MyInterceptor"></interceptor>
9. </interceptors>
11. <action name="login" **class**="com.Login">
12. <interceptor-ref name="defaultStack"></interceptor-ref>
13. <interceptor-ref name="upper"></interceptor-ref>
14. <result>welcome.jsp</result>
15. </action>
17. </**package**>
18. </struts>

Other Required Resources

The other required resources are

* index.jsp
* Login.java
* welcome.jsp

1) Create form to get input (index.jsp)

**index.jsp**

This jsp page creates a form using struts UI tags. It receives name from the user.

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <s:form action="login">
3. <s:textfield name="name" label="Name"></s:textfield>
4. <s:submit value="login"></s:submit>
5. </s:form>

2) Create the action class

It is the simple action class containing name property with its setter and getter methods.

**Login.java**

1. **package** com;
2. **public** **class** Login {
3. **private** String name;
5. **public** String getName() {
6. **return** name;
7. }
8. **public** **void** setName(String name) {
9. **this**.name = name;
10. }
11. **public** String execute(){
12. **return** "success";
13. }
14. }

3) Create view component

This jsp page displays the name input by the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. Welcome, <s:property value="name"/>

Struts 2 params interceptor example

1. [Params interceptor](https://www.javatpoint.com/struts-2-params-interceptor-example)
2. [Parameters of params interceptor](https://www.javatpoint.com/struts-2-params-interceptor-example)
3. [Example of params interceptor](https://www.javatpoint.com/struts-2-params-interceptor-example)

The **params interceptor** also known as parameters interceptor is used to set all parameters on the valuestack.

It is found in the default stack bydefault. So you don't need to specify it explicitely.

Internal working of params interceptor

It gets all parameters by calling the **getParameters() method of ActionContext** and sets it on the valuestack by calling the **setValue() method of ValueStack**.

Parameters of params interceptor

There are 4 parameters defined for params interceptor.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **ordered** | It is true bydefault but can be used to top-down the property setter behaviour. |
| **paramNameMaxLength** | specifies the maximum length for the parameter. The default length is 100 characters bydefault. The parameters length more that 100 will be ignored. |
| **excludeParams** | specifies the unallowed parameter names. You can specify multiple names separated with comma. |
| **acceptParamNames** | specifies the accepted parameter names. |

Example of params interceptor

Let's see the simple example of params interceptor.

1. **<action** name="login" class="com.javatpoint.Login"**>**
2. **<interceptor-ref** name="params"**/>**
3. **<result** name="success"**>**login-success.jsp**</result>**
4. **</action>**

Note: If you specify any interceptor for the action explicitely, default interceptors will not be available for the action class.

Struts 2 defaultStack interceptors

The params interceptor is found in the default stack. You don't need to specify interceptors found in the default-stack. The defaultStack interceptors are as follows:

1. **<interceptor-stack** name="defaultStack"**>**
2. **<interceptor-ref** name="exception"**/>**
3. **<interceptor-ref** name="alias"**/>**
4. **<interceptor-ref** name="servletConfig"**/>**
5. **<interceptor-ref** name="prepare"**/>**
6. **<interceptor-ref** name="i18n"**/>**
7. **<interceptor-ref** name="chain"**/>**
8. **<interceptor-ref** name="debugging"**/>**
9. **<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>**

|  |
| --- |
| Struts 2 modelDriven interceptor example   1. [modelDriven interceptor](https://www.javatpoint.com/struts-2-modeldriven-interceptor-example) 2. [Parameters of modelDriven interceptor](https://www.javatpoint.com/struts-2-modeldriven-interceptor-example) 3. [Example of modelDriven interceptor](https://www.javatpoint.com/struts-2-modeldriven-interceptor-example)   The **modelDriven interceptor** makes other model object as the default object of valuestack.  Bydefault, action object is the default object of valuestack.  To use the modelDriven interceptor, you need to implement **ModelDriven interface** in your action class and override its method getModel().  It is found in the default stack bydefault. So you don't need to specify it explicitely.  Parameters of modelDriven interceptor  There is no parameter defined for modelDriven interceptor.  Example of modelDriven interceptor   1. **<action** name="login" class="com.javatpoint.LoginAction"**>** 2. **<interceptor-ref** name="params"**/>** 3. **<interceptor-ref** name="modelDriven"**/>** 4. **<result** name="success"**>**login-success.jsp**</result>** 5. **</action>**   Full example of modelDriven interceptor  Let's see the full example of modelDriven interceptor.  *File: index.jsp*   1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>** 3. **<s:form** action="login"**>** 4. **<s:textfield** name="name" label="Name"**></s:textfield>** 5. **<s:password** name="password" label="Password"**></s:password>** 6. **<s:submit** value="login"**></s:submit>** 7. **</s:form>**   *File: struts.xml*   1. **<?xml** version="1.0" encoding="UTF-8" **?>** 2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd"**>** 3. **<struts>** 5. **<package** name="abc" extends="struts-default" **>** 7. **<action** name="login" class="com.javatpoint.Login"**>** 8. **<result** name="success" **>**/login-success.jsp**</result>** 9. **<result** name="error"**>**/login-error.jsp**</result>** 10. **</action>** 12. **</package>** 13. **</struts>**   *File: Login.java*   1. **package** com.javatpoint; 2. **import** com.opensymphony.xwork2.ModelDriven; 4. **public** **class** Login **implements** ModelDriven<User>{ 5. **private** User user; 7. **public** User getUser() { 8. **return** user; 9. } 11. **public** **void** setUser(User user) { 12. **this**.user = user; 13. } 14. **public** User getModel(){ 15. user=**new** User(); 16. **return** user; 17. } 18. **public** String execute(){ 19. **if**(user.getPassword().equals("admin")){ 20. **return** "success"; 21. } 22. **else**{ 23. **return** "error"; 24. } 25. } 27. }   *File: User.java*   1. **package** com.javatpoint; 3. **public** **class** User { 4. **private** String name,password; 5. //getters and setters 6. }   *File: login-success.jsp*   1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>** 3. Welcome, **<s:property** value="name"**/>**   *File: login-error.jsp*   1. Sorry, username or password error! 2. **<jsp:include** page="index.jsp"**></jsp:include>**   [download the full example (developed in Eclipse ide without jar)](https://static.javatpoint.com/src/st/eclipse/modelDriven.zip)  [download the full example (developed in Myeclipse ide)](https://static.javatpoint.com/src/st/modelDriven.zip)  Output  struts 2 modeldriven interceptor example output 1 struts 2 modeldriven interceptor example output 2  If you don't implement the ModelDriven interface, you need to use user.name and user.password field names in index.jsp file otherwise given value will not be set.  Next Topic |

Struts 2 Exception Handling - exception interceptor

1. [Struts 2 Exception Handling](https://www.javatpoint.com/struts-2-exception-handling-exception-interceptor)
2. [Example of Exception Handling in struts 2](https://www.javatpoint.com/struts-2-exception-handling-exception-interceptor)

In our web application, there might occur exception at any point.

To overcome this problem, struts 2 provides a mechanism of **global exception handling** where we can display a global result to the user.

Struts 2 automatically log the uncaught exceptions and redirects the user to the error handler page.

Understanding the internal working of exception interceptor

If there occurs exception, it is wrapped in **ExceptionHolder** and pushed in the valuestack so that we can easily access exception object from the result.

It is recommended to make this interceptor as the first interceptor, so that it can handle all the exception whether it is thrown by other interceptors.

Parameters of exception interceptor

There are 3 parameters defined for exception interceptor. All are optional.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **logEnabled** | specifies log should be enabled or not. You can pass true or false. |
| **logLevel** | specifies the log level. It may be trace, debug, info, warn, error, fatal. Default log level is **debug**. |
| **logCategory** | specifies the log category eg. com.mycompany.app. The default is com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor.. |

Example of exception handling in struts 2

For exception handling, we specify the global-result and global-exception-mappings in the struts.xml file.

**struts.xml**

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd"**>**
4. **<struts>**
5. **<package** name="aa" extends="struts-default"**>**
7. **<global-results>**
8. **<result** name="myresult"**>**globalresult.jsp**</result>**
9. **</global-results>**
11. **<global-exception-mappings>**
12. **<exception-mapping** result="myresult" exception="java.lang.Exception"**></exception-mapping>**
13. **</global-exception-mappings>**
15. **<action** name="login" class="com.Login"**>**
16. **<result>**welcome.jsp**</result>**
17. **<result** name="error"**>**error.jsp**</result>**
18. **</action>**
20. **</package>**
21. **</struts>**

The **global-results** sub-element of package specifies the global-result for this package.

The **result** sub-element of global-result specifies the result page that will be rendered to the user as a view.

The **global-exception-mappings** sub-element of package specifies the exception mapping for all the actions of this package.

The **exception-mapping** sub-element of global-exception-mapping maps the given result for the given exception type. In this example, we are using the Exception which the parent of many exception classes such as IOException, ArithmeticException etc. It means if any exception occurs, specified result will be invoked.

Notice that global-results must be specified before global-exception-mappings as we are using global result in global-exception-mappings.

Displaying exception

We can display the exception on the browser by printing the exception or exceptionStack.

The exception object prints the exception name whereas exceptionStack prints the exception details.

**globalresult.jsp**

1. **<p>**Exception Name: **<s:property** value="exception" **/>** **</p>**
2. **<p>**Exception Details: **<s:property** value="exceptionStack" **/></p>**

Full example of exception handling

The other required resources to complete this example are as follows:

1. **Input page (index.jsp)**
2. **Action class (Login.java)**
3. **View components (globalresult.jsp, welcome.jsp and error.jsp)**

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name and password from the user.

**index.jsp**

1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>**
3. **<s:form** action="login"**>**
4. **<s:textfield** name="name" label="Name"**></s:textfield>**
5. **<s:password** name="password" label="Password"**></s:password>**
6. **<s:submit** value="login"**></s:submit>**
7. **</s:form>**

2) Create the action class

This action class contains two fields name and password and one method execute. Here, we are throwing exception self if password matches struts.

This is dummy example, if you comment the mentioned line in the execute method, exception will not occur.

**Login.java**

1. **package** com;
2. **public** **class** Login {
3. **private** String name,password;
4. //getters and setters
6. **public** String execute(){
7. **if**(password.equals("struts")){
8. **int** a=12/0;//If you comment this, exception will not occur
9. **return** "success";
10. }**else**
11. **return** "error";
12. }
13. }

3) Create view components

There are three view components globalresult.jsp that displays the exception message, welcome.jsp that displays the welcome message with the username and error.jsp that displays the error message.

**globalresult.jsp**

1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>**
3. Sorry an exception occured!
4. **<p>**Exception Name: **<s:property** value="exception" **/>** **</p>**
5. **<p>**Exception Details: **<s:property** value="exceptionStack" **/></p>**
6. **<form>**
7. **<input** type="button" value="back" onclick="history.back()"**>**
8. **</form>**

**welcome.jsp**

1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>**
2. Welcome, **<s:property** value="name"**/>**

**error.jsp**

1. Sorry username or password error!
2. **<jsp:include** page="index.jsp"**></jsp:include>**

Struts2 File Upload Example

1. [FileUpload interceptor](https://www.javatpoint.com/struts-2-file-upload-example)
2. [Parameters of fileUpload interceptor](https://www.javatpoint.com/struts-2-file-upload-example)
3. [Example of fileUpload interceptor](https://www.javatpoint.com/struts-2-file-upload-example)

The **fileUpload** interceptor automatically works for all the requests that includes files.

We can use this interceptor to control the working of file upload in struts2 such as defining allowed types, maximum file size etc.

Parameters of fileupload interceptor

There are 2 parameters defined for fileupload interceptor.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **maximumSize** | specifies maximum size of the file to be uploaded. |
| **allowedTypes** | specifies allowed types. It may be image/png, image/jpg etc. |

Automatically added parameters

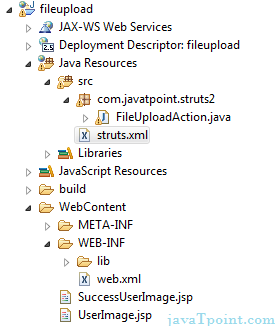
It automatically adds 2 parameters in the request:

1. **String fileName** represents the filename of the file.
2. **String contentType** specifies the content type of the file.

The fileName and contentType name depends on the request parameter for the file. If filename is file, you need to use fileFileName and fileContentType. If filename is userImage, you need to use userImageFileName and userImageContentType in the Action class.

Image upload example using struts 2

Let's see the **directory structure** of file upload application.



1) Create UserImage.jsp

This jsp page creates a form using struts UI tags. It receives file from the user.

**index.jsp**

1. **<**%@ page contentType="text/html; charset=UTF-8"%**>**
2. **<**%@ taglib prefix="s" uri="/struts-tags"%**>**
3. **<html>**
4. **<head>**
5. **<title>**Upload User Image**</title>**
6. **</head>**
7. **<body>**
8. **<h2>**
9. Struts2 File Upload & Save Example without Database
10. **</h2>**
11. **<s:actionerror** **/>**
12. **<s:form** action="userImage" method="post" enctype="multipart/form-data"**>**
13. **<s:file** name="userImage" label="Image" **/>**
14. **<s:submit** value="Upload" align="center" **/>**
15. **</s:form>**
16. **</body>**
17. **</html>**

2) Create SuccessUserImage.jsp

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**SuccessUserImage.jsp**

1. **<**%@ page contentType="text/html; charset=UTF-8"%**><**%@ taglib prefix="s"
2. uri="/struts-tags"%**>**
3. **<html>**
4. **<head>**
5. **<title>**Success: Upload User Image**</title>**
6. **</head>**
7. **<body>**
8. **<h2>**
9. Struts2 File Upload Example
10. **</h2>**
11. User Image: **<s:property** value="userImage" **/><br/>**
12. Content Type:**<s:property** value="userImageContentType" **/><br/>**
13. File Name:  **<s:property** value="userImageFileName" **/><br/>**
14. Uploaded Image: **<img** src="userimages/**<s:property** value="userImageFileName"/>"
15. width="100" height="100" **/>**
16. **</body>**
17. **</html>**

3) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
2. **import** java.io.File;
3. **import** javax.servlet.http.HttpServletRequest;
4. **import** org.apache.commons.io.FileUtils;
5. **import** com.opensymphony.xwork2.ActionSupport;
7. **public** **class** FileUploadAction **extends** ActionSupport{
8. **private** File userImage;
9. **private** String userImageContentType;
10. **private** String userImageFileName;
12. **public** String execute() {
13. **try** {
14. String filePath = ServletActionContext.getServletContext().getRealPath("/").concat("userimages");
16. System.out.println("Image Location:" + filePath);//see the server console for actual location
17. File fileToCreate = **new** File(filePath,userImageFileName);
18. FileUtils.copyFile(userImage, fileToCreate);//copying source file to new file
20. **return** SUCCESS;
21. }
22. **public** File getUserImage() {
23. **return** userImage;
24. }
25. **public** **void** setUserImage(File userImage) {
26. **this**.userImage = userImage;
27. }
28. **public** String getUserImageContentType() {
29. **return** userImageContentType;
30. }
32. **public** **void** setUserImageContentType(String userImageContentType) {
33. **this**.userImageContentType = userImageContentType;
34. }
35. **public** String getUserImageFileName() {
36. **return** userImageFileName;
37. }
38. **public** **void** setUserImageFileName(String userImageFileName) {
39. **this**.userImageFileName = userImageFileName;
40. }
41. }

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

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

7. **<struts>**
8. **<package** name="fileUploadPackage" extends="struts-default"**>**
9. **<action** name="userImage" class="com.javatpoint.FileUploadAction"**>**
10. **<interceptor-ref** name="fileUpload"**>**
11. **<param** name="maximumSize"**>**2097152**</param>**
13. **<param** name="allowedTypes"**>**
14. image/png,image/gif,image/jpeg,image/pjpeg
15. **</param>**
16. **</interceptor-ref>**
17. **<interceptor-ref** name="defaultStack"**></interceptor-ref>**
18. **<result** name="success"**>**SuccessUserImage.jsp**</result>**
19. **<result** name="input"**>**UserImage.jsp**</result>**
20. **</action>**
21. **</package>**
22. **</struts>**

Struts 2 Validation Tutorial

1. [Struts 2 Validation Tutorial](https://www.javatpoint.com/struts-2-validation-tutorial)
2. [Three mechanism to perform validation](https://www.javatpoint.com/struts-2-validation-tutorial)

To avoid the wrong values, we need to perform validation on forms where user submits some values. For example, if user writes his/her email id as abc, we need to give error message to the user that the given email id is not correct. So that we can have only valuable informations.

There are three ways to perform validation in struts 2.

**1)**[**By Custom Validation**](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor) Here, we must implement the Validateable interface (or extend ActionSupport class) and provide the implementation of validate method.

**2)**[**By Input Validation (built-in validators)**](https://www.javatpoint.com/struts-2-validation-by-bundled-validators) Struts 2 provides a lot of predefined that can be used in struts 2 application to perform validation.

Struts 2 provides following bundled validators.

* [requiredstring validator](https://www.javatpoint.com/struts-2-requiredstring-validation-example)
* [stringlength validator](https://www.javatpoint.com/struts-2-string-length-validation-example)
* [email validator](https://www.javatpoint.com/struts-2-email-validation-example)
* [date validator](https://www.javatpoint.com/struts-2-date-validation-example)
* [int validator](https://www.javatpoint.com/struts-2-int-validation-example)
* [double validator](https://www.javatpoint.com/struts-2-double-validation-example)
* [url validator](https://www.javatpoint.com/struts-2-url-validation-example)
* [regex validator](https://www.javatpoint.com/struts-2-regex-validation-example)

**3)**[**By Ajax Validation (built-in validators with ajax)**](https://www.javatpoint.com/struts-2-ajax-validation-jsonValidation-interceptor) If we don't want to refresh the page, we can use jsonValidation interceptor to perform validation with ajax.

Struts 2 Custom Validation - Workflow Interceptor

1. [Struts 2 Custom Validation](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor)
2. [Workflow Interceptor](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor)
3. [Validateable Interface](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor)
4. [ValidationAware Interface](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor)
5. [Example of Custom validation](https://www.javatpoint.com/struts-2-custom-validation-workflow-interceptor)

We can define our own validation logic (custom validation) in struts 2 by implementing the **Validateable** interface in the action class.

The **workflow interceptor** is used to get information about the error messages defined in the action class.

Workflow Interceptor

The **workflow interceptor** checks if there is any validation errors or not. It doesn't perform any validation.

It is applied when action class implements the Validateable interface. The **input** is the default parameter for this interceptor that determines the result to be invoked for the action or field error.

It is found in the defaultStack so we don't need to define it explicitly.

Parameters of workflow interceptor

There is only 1 parameter defined for workflow interceptor.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **inputResultName** | specifies the result name to be returned if field error or action error is found. It is set to **input** bydefault. |

Validateabale interface

The **Validateable** interface must be implemented to perform validation logic in the action class. It contains only one method **validate()** that must be overridden in the action class to define the validation logic. Signature of the validate method is:

1. **public** **void** validate();

ValidationAware interface

The **ValidationAware** interface can accept the **field level** or **action class level** error messages. The field level messages are kept in Map and Action class level messages are kept in collection. It should be implemented by the action class to add any error message.

Methods of ValidatationAware interface

The methods of ValidationAware interface are as follows:

|  |  |
| --- | --- |
| **Method** | **Description** |
| **void addFieldError(String fieldName,String errorMessage)** | adds the error message for the specified field. |
| **void addActionError(String errorMessage)** | adds an Action-level error message for this action. |
| **void addActionMessage(String message)** | adds an Action-level message for this action. |
| **void setFieldErrors(Map<String,List<String>> map)** | sets a collection of error messages for fields. |
| **void setActionErrors(Collection<String> errorMessages)** | sets a collection of error messages for this action. |
| **void setActionMessages(Collection<String> messages)** | sets a collection of messages for this action. |
| **boolean hasErrors()** | checks if there are any field or action errors. |
| **boolean hasFieldErrors()** | checks if there are any field errors. |
| **boolean hasActionErrors()** | checks if there are any Action-level error messages. |
| **boolean hasActionMessages()** | checks if there are any Action-level messages. |
| **Map<String,List<String>> getFieldErrors()** | returns all the field level error messages. |
| **Collection<String> getActionErrors()** | returns all the Action-level error messages. |
| **Collection<String> getActionMessages()** | returns all the Action-level messages. |

Note: ActionSupport class implements Validateable and ValidationAware interfaces, so we can inherit the ActionSupport class to define the validation logic and error messages.

Steps to perform custom validation

The steps are as follows:

1. **create the form to get input from the user**
2. **Define the validation logic in action class by extending the ActionSupport class and overriding the validate method**
3. **Define result for the error message by the name input in struts.xml file**

Example to perform custom validation

In this example, we are creating 4 pages :

1. **index.jsp** for input from the user.
2. **RegisterAction.java** for defining the validation logic.
3. **struts.xml** for defining the result and action.
4. **welcome.jsp** for the view component.

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <s:form action="register">
3. <s:textfield name="name" label="Name"></s:textfield>
4. <s:password name="password" label="Password"></s:password>
5. <s:submit value="register"></s:submit>
6. </s:form>

2) Create the action class

This action class inherits the ActionSupport class and overrides the validate method to define the validation logic.

**RegisterAction.java**

1. **package** com.javatpoint;
2. **import** com.opensymphony.xwork2.ActionSupport;
4. **public** **class** RegisterAction **extends** ActionSupport{
5. **private** String name,password;
6. **public** **void** validate() {
7. **if**(name.length()<1)
8. addFieldError("name","Name can't be blank");
9. **if**(password.length()<6)
10. addFieldError("password","Password must be greater than 5");
11. }
13. //getters and setters
15. **public** String execute(){
16. //perform business logic here
17. **return** "success";
18. }
19. }

3) Define a input result in struts.xml

This xml file defines an extra result by the name input, that will be invoked if any error message is found in the action class.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts
3. Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
6. <**package** name="default" **extends**="struts-default">
7. <action name="register" **class**="com.javatpoint.RegisterAction">
8. <result>welcome.jsp</result>
9. <result name="input">index.jsp</result>
10. </action>
11. </**package**>
12. </struts>

4) Create view component

It is the simple jsp file displaying the information of the user.

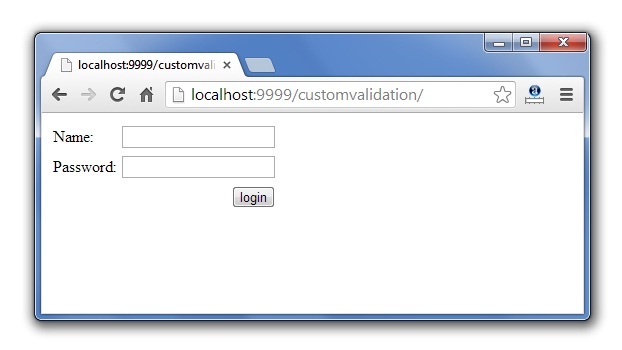
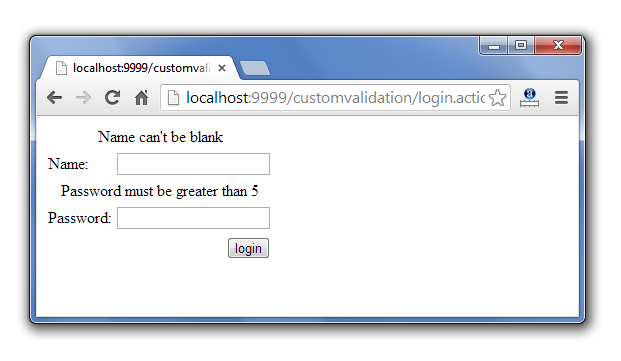
**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. Name:<s:property value="name"/><br/>
3. Password:<s:property value="password"/><br/>

[download this example (developed in Eclipse ide without jar)](https://static.javatpoint.com/src/st/eclipse/customvalidation.zip)

[download this example (developed in Myeclipse ide)](https://static.javatpoint.com/src/st/customvalidation.zip)

Output

Defining action level error message

The action level error message works for the whole form. We can define the action level error message by **addActionError()** method of ValidationAware interface in validate() method. For example:

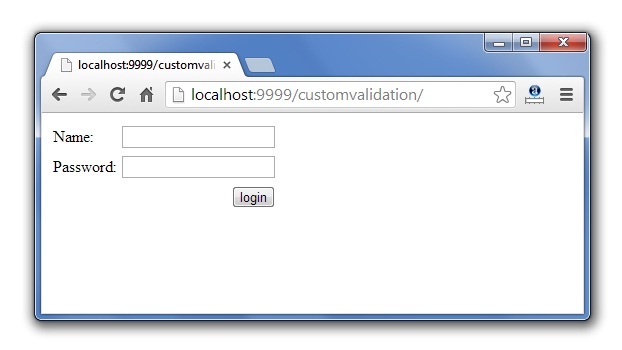
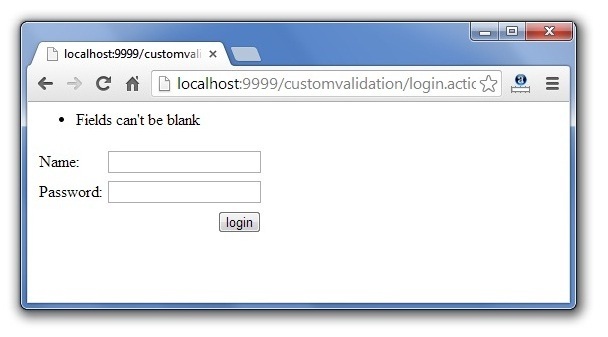
1. **package** com.javatpoint;
2. **import** com.opensymphony.xwork2.ActionSupport;
3. **public** **class** RegisterAction **extends** ActionSupport{
4. **private** String name,password,email;
5. **public** **void** validate() {
6. **if**(name.trim().length()<1 || password.trim().length()<1){
7. addActionError("Fields can't be blank");
8. }
9. }
11. //getters and setters
13. **public** String execute(){
14. **return** "success";
15. }
16. }

Now you need to use **actionerror** tag in index.jsp file to display the action level error message.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <s:actionerror/>
3. <s:form action="register">
4. <s:textfield name="name" label="Name"></s:textfield>
5. <s:password name="password" label="Password"></s:password>
6. <s:textfield name="email" label="Email Id"></s:textfield>
7. <s:submit value="register"></s:submit>
8. </s:form>

Output

[download this example (developed in Eclipse ide without jar)](https://static.javatpoint.com/src/st/eclipse/customvalidation.zip)

[download this example (developed in Myeclipse ide)](https://static.javatpoint.com/src/st/customvalidation.zip)

Struts 2 requiredstring validation example

1. [requiredstring validator](https://www.javatpoint.com/struts-2-requiredstring-validation-example)
2. [Parameters of requiredstring validator](https://www.javatpoint.com/struts-2-requiredstring-validation-example)
3. [Example of requiredstring validator](https://www.javatpoint.com/struts-2-requiredstring-validation-example)

The **requiredstring validator** specifies that string can't be null or blank.

It trims the string bydefault then checks if its length is greater that 0.

Parameters of requiredstring validator

There are two parameters defined for requiredstring validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **trim** | trims the field values. It is bydefault true means it is enabled bydefault. |

Example of requiredstring validator

1. **<validators>**
2. <!-- Plain-Validator Syntax -->
3. **<validator** type="requiredstring"**>**
4. **<param** name="fieldName"**>**username**</param>**
5. **<param** name="trim"**>**true**</param>**
6. **<message>**username is required**</message>**
7. **</validator>**
9. **</validators>**
10. **<validators>**
11. <!-- Field-Validator Syntax -->
12. **<field** name="username"**>**
13. **<field-validator** type="requiredstring"**>**
14. **<param** name="trim"**>**true**</param>**
15. **<message>**username is required**</message>**
16. **</field-validator>**
17. **</field>**
19. **</validators>**

Full example of requiredstring validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>**
2. **<html>**
3. **<head>**
4. **<STYLE** type="text/css"**>**
5. .errorMessage{color:red;}
6. **</STYLE>**
7. **</head>**
8. **<body>**
10. **<s:form** action="register"**>**
11. **<s:textfield** name="username" label="Username"**></s:textfield>**
12. **<s:password** name="userpass" label="Password"**></s:password>**
13. **<s:submit** value="register"**></s:submit>**
14. **</s:form>**
16. **</body>**
17. **</html>**

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** String username,userpass;
8. **public** String getUsername() {
9. **return** username;
10. }
12. **public** **void** setUsername(String username) {
13. **this**.username = username;
14. }
16. **public** String getUserpass() {
17. **return** userpass;
18. }
20. **public** **void** setUserpass(String userpass) {
21. **this**.userpass = userpass;
22. }
24. **public** String execute(){
25. **return** "success";
26. }
28. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd"**>**
7. **<validators>**
9. **<field** name="username"**>**
10. **<field-validator** type="requiredstring"**>**
11. **<message>**Name can't be blank**</message>**
12. **</field-validator>**
13. **</field>**

16. **</validators>**

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd"**>**
3. **<struts>**
5. **<package** name="default" extends="struts-default"**>**
6. **<action** name="register" class="com.javatpoint.Register"**>**
7. **<result** name="input"**>**index.jsp**</result>**
8. **<result>**welcome.jsp**</result>**
9. **</action>**
11. **</package>**
12. **</struts>**

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. **<**%@ taglib uri="/struts-tags" prefix="s" %**>**
3. Welcome,**<s:property** value="username"**/>**

Struts2 String Length Validation Example

1. [stringlength validator](https://www.javatpoint.com/struts-2-string-length-validation-example)
2. [Parameters of stringlength validator](https://www.javatpoint.com/struts-2-string-length-validation-example)
3. [Example of stringlength validator](https://www.javatpoint.com/struts-2-string-length-validation-example)

The **stringlength validator** specifies that string must be of given length. It can be used in username, password etc.

It trims the string bydefault then checks if its length is of the given length.

Parameters of stringlength validator

There are 4 parameters defined for stringlength validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **minLength** | specifies the minimum length. It is ignored bydefault. |
| **maxLength** | specifies the maximum length. It is ignored bydefault. |
| **trim** | trims the field values. It is bydefault true means it is enabled bydefault. |

Example of stringlength validator

1. **<validators>**
2. <!-- Plain Validator Syntax -->
3. **<validator** type="stringlength"**>**
4. **<param** name="fieldName"**>**password**</param>**
5. **<param** name="minLength"**>**6**</param>**
6. **<param** name="maxLength"**>**10**</param>**
7. **<param** name="trim"**>**true**</param>**
8. **<message>**Password must be between 6 to 10 characters long**</message>**
9. **</validator>**
11. **</validators>**
12. **<validators>**
13. <!-- Field-Validator Syntax -->
14. **<field** name="password"**>**
15. **<field-validator** type="stringlength"**>**
16. **<param** name="minLength"**>**6**</param>**
17. **<param** name="maxLength"**>**10**</param>**
18. **<param** name="trim"**>**true**</param>**
19. **<message>**Password must be between 6 to 10 characters long**</message>**
20. **</field-validator>**
21. **</field>**
23. **</validators>**

Full example of stringlength validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <html>
3. <head>
4. <STYLE type="text/css">
5. .errorMessage{color:red;}
6. </STYLE>
7. </head>
8. <body>
10. <s:form action="register">
11. <s:textfield name="username" label="Username"></s:textfield>
12. <s:password name="userpass" label="Password"></s:password>
13. <s:submit value="register"></s:submit>
14. </s:form>
16. </body>
17. </html>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** String username,userpass;
8. **public** String getUsername() {
9. **return** username;
10. }
12. **public** **void** setUsername(String username) {
13. **this**.username = username;
14. }
16. **public** String getUserpass() {
17. **return** userpass;
18. }
20. **public** **void** setUserpass(String userpass) {
21. **this**.userpass = userpass;
22. }
24. **public** String execute(){
25. **return** "success";
26. }
28. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
7. <validators>
9. <field name="username">
10. <field-validator type="requiredstring">
11. <message>Name can't be blank</message>
12. </field-validator>
13. </field>
15. <field name="userpass">
16. <field-validator type="requiredstring">
17. <message>Password can't be blank</message>
18. </field-validator>
19. <field-validator type="stringlength">
20. <param name="minLength">6</param>
21. <param name="maxLength">10</param>
22. <message>Password must be greater than or equal to 6 and less than or equal to 10</message>
23. </field-validator>
25. </field>
27. </validators>

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
3. <struts>
5. <**package** name="default" **extends**="struts-default">
6. <action name="register" **class**="com.javatpoint.Register">
7. <result name="input">index.jsp</result>
8. <result>welcome.jsp</result>
9. </action>
11. </**package**>
13. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Welcome,<s:property value="username"/>

Struts2 Email Validation Example

1. [email validator](https://www.javatpoint.com/struts-2-email-validation-example)
2. [Parameters of email validator](https://www.javatpoint.com/struts-2-email-validation-example)
3. [Example of email validator](https://www.javatpoint.com/struts-2-email-validation-example)

The **email validator** checks that given field has valid email address. It works only if field is not blank.

Parameters of email validator

There is only one parameter defined for email validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |

Example of email validator

1. **<validators>**
2. <!-- Plain Validator Syntax -->
3. **<validator** type="email"**>**
4. **<param** name="fieldName"**>**email**</param>**
5. **<message>**Please enter a valid email address**</message>**
6. **</validator>**
8. **</validators>**
9. **<validators>**
10. <!-- Field-Validator Syntax -->
11. **<field** name="email"**>**
12. **<field-validator** type="email"**>**
13. **<message>**Please enter a valid email address.**</message>**
14. **</field-validator>**
15. **</field>**
17. **</validators>**

Full example of email validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <html>
3. <head>
4. <STYLE type="text/css">
5. .errorMessage{color:red;}
6. </STYLE>
7. </head>
8. <body>
10. <s:form action="register">
11. <s:textfield name="email" label="Enter Email Id"></s:textfield>
12. <s:submit value="register"></s:submit>
13. </s:form>
15. </body>
16. </html>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** String email;

9. **public** String getEmail() {
10. **return** email;
11. }

14. **public** **void** setEmail(String email) {
15. **this**.email = email;
16. }

19. **public** String execute(){
20. **return** "success";
21. }
23. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
7. <validators>
9. <field name="email">
10. <field-validator type="requiredstring">
11. <message>Email Id can't be blank</message>
12. </field-validator>
13. <field-validator type="email">
14. <message>Please enter correct email id</message>
15. </field-validator>
17. </field>


21. </validators>

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
3. <struts>
5. <**package** name="default" **extends**="struts-default">
6. <action name="register" **class**="com.javatpoint.Register">
7. <result name="input">index.jsp</result>
8. <result>welcome.jsp</result>
9. </action>
11. </**package**>
13. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Email Id is:,<s:property value="email"/>

Struts2 Date Validation Example

1. [date validator](https://www.javatpoint.com/struts-2-date-validation-example)
2. [Parameters of date validator](https://www.javatpoint.com/struts-2-date-validation-example)
3. [Example of date validator](https://www.javatpoint.com/struts-2-date-validation-example)

The **date validator** checks that given date is witin the specified range.

Parameters of date validator

There is 3 parameters defined for date validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **min** | specifies the minimum range. It is ignored bydefault. |
| **max** | specifies the maximum range. It is ignored bydefault. |

Example of date validator

1. **<validators>**
2. <!-- Plain Validator syntax -->
3. **<validator** type="date"**>**
4. **<param** name="fieldName"**>**dob**</param>**
5. **<param** name="min"**>**01/01/1980**</param>**
6. **<param** name="max"**>**01/01/2010**</param>**
7. **<message>**Date of Birth must be within ${min} and ${max}**</message>**
8. **</validator>**
9. **</validators>**
10. **<validators>**
11. <!-- Field Validator Syntax -->
12. **<field** name="dob"**>**
13. **<field-validator** type="date"**>**
14. **<param** name="min"**>**01/01/1980**</param>**
15. **<param** name="max"**>**01/01/2010**</param>**
16. **<message>**Date of Birth must be within ${min} and ${max}**</message>**
17. **</field>**
18. **</field>**
20. **</validators>**

Full example of date validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. <s:form action="register">
4. <s:textfield name="dob" label="Date of Birth"></s:textfield>
5. <s:submit value="register"></s:submit>
6. </s:form>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** java.util.Date;
5. **import** com.opensymphony.xwork2.ActionSupport;
7. **public** **class** Register **extends** ActionSupport{
8. **private** Date dob;

11. **public** Date getDob() {
12. **return** dob;
13. }

16. **public** **void** setDob(Date dob) {
17. **this**.dob = dob;
18. }

21. **public** String execute(){
22. **return** "success";
23. }
25. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
7. <validators>
9. <field name="dob">
10. <field-validator type="date">
11. <param name="min">01/01/1950</param>
12. <param name="max">01/01/2010</param>
14. <message>Date of Birth must be between ${min} to ${max}</message>
15. </field-validator>
16. </field>

19. </validators>

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
3. <struts>
5. <**package** name="default" **extends**="struts-default">
6. <action name="register" **class**="com.javatpoint.Register">
7. <result name="input">index.jsp</result>
8. <result>welcome.jsp</result>
9. </action>
10. </**package**>
12. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Date of Birth:<s:property value="dob"/>

Struts 2 int validation example

1. [int validator](https://www.javatpoint.com/struts-2-int-validation-example)
2. [Parameters of int validator](https://www.javatpoint.com/struts-2-int-validation-example)
3. [Example of int validator](https://www.javatpoint.com/struts-2-int-validation-example)

The **int validator** checks that given number is witin the specified range. It can be used in productId, employeeId etc.

Parameters of int validator

There is 3 parameters defined for int validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **min** | specifies the minimum value. It is ignored bydefault. |
| **max** | specifies the maximum value. It is ignored bydefault. |

Example of int validator

1. **<validators>**
2. <!-- Plain Validator Syntax -->
3. **<validator** type="int"**>**
4. **<param** name="fieldName"**>**age**</param>**
5. **<param** name="min"**>**16**</param>**
6. **<param** name="max"**>**50**</param>**
7. **<message>**Age must be between ${min} and ${max}**</message>**
8. **</validator>**
10. **</validators>**
11. **<validators>**
12. <!-- Field Validator Syntax -->
13. **<field** name="age"**>**
14. **<field-validator** type="int"**>**
15. **<param** name="min"**>**16**</param>**
16. **<param** name="max"**>**50**</param>**
17. **<message>**Age must be between ${min} and ${max}**</message>**
18. **</field-validator>**
19. **</field>**
21. **</validators>**

Full example of int validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <html>
3. <head>
4. <STYLE type="text/css">
5. .errorMessage{color:red;}
6. </STYLE>
7. </head>
8. <body>
10. <s:form action="register">
11. <s:textfield name="id" label="Product Id"></s:textfield>
12. <s:textfield name="price" label="Product Price"></s:textfield>
13. <s:submit value="register"></s:submit>
14. </s:form>
16. </body>
17. </html>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** **int** id;
7. **private** **double** price;
9. **public** **int** getId() {
10. **return** id;
11. }
13. **public** **void** setId(**int** id) {
14. **this**.id = id;
15. }
17. **public** **double** getPrice() {
18. **return** price;
19. }
21. **public** **void** setPrice(**double** price) {
22. **this**.price = price;
23. }
25. **public** String execute(){
26. **return** "success";
27. }
29. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
6. <validators>
8. <field name="id">
9. <field-validator type="int">
10. <param name="min">1</param>
11. <param name="max">999</param>
13. <message>Id must be between ${min} to ${max}</message>
14. </field-validator>
15. </field>
17. </validators>

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
3. <struts>
4. <**package** name="default" **extends**="struts-default">
5. <action name="register" **class**="com.javatpoint.Register">
6. <result name="input">index.jsp</result>
7. <result>welcome.jsp</result>
8. </action>
10. </**package**>
11. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Product Id:<s:property value="id"/><br/>
4. Product price:<s:property value="price"/>

Struts 2 double validation example

1. [double validator](https://www.javatpoint.com/struts-2-double-validation-example)
2. [Parameters of double validator](https://www.javatpoint.com/struts-2-double-validation-example)
3. [Example of double validator](https://www.javatpoint.com/struts-2-double-validation-example)

The **double validator** checks that given floating-point number is within the specified range. It can be used in product price etc.

Parameters of double validator

There is 5 parameters defined for double validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **minInclusive** | specifies the minimum inclusive value. It is ignored bydefault. |
| **maxInclusive** | specifies the maximum inclusive value. It is ignored bydefault. |
| **minExclusive** | specifies the minimum exclusive value. It is ignored bydefault. |
| **maxExclusive** | specifies the maximum exclusive value. It is ignored bydefault. |

Example of double validator

1. **<validators>**
2. <!-- Plain Validator Syntax -->
3. **<validator** type="double"**>**
4. **<param** name="fieldName"**>**price**</param>**
5. **<param** name="minInclusive"**>**100.0**</param>**
6. **<param** name="maxInclusive"**>**10000.0**</param>**
7. **<message>**Price must be between ${minInclusive} and ${maxInclusive}**</message>**
8. **</validator>**
10. **</validators>**
11. **<validators>**
12. <!-- Field Validator Syntax -->
13. **<field** name="price"**>**
14. **<field-validator** type="double"**>**
15. **<param** name="minInclusive"**>**100.0**</param>**
16. **<param** name="maxInclusive"**>**10000.0**</param>**
17. **<message>**Price must be between ${minInclusive} and ${maxInclusive}**</message>**
18. **</field-validator>**
19. **</field>**
21. **</validators>**

Full example of double validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <html>
3. <head>
4. <STYLE type="text/css">
5. .errorMessage{color:red;}
6. </STYLE>
7. </head>
8. <body>
10. <s:form action="register">
11. <s:textfield name="id" label="Product Id"></s:textfield>
12. <s:textfield name="price" label="Product Price"></s:textfield>
13. <s:submit value="register"></s:submit>
14. </s:form>
16. </body>
17. </html>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** **int** id;
7. **private** **double** price;
9. **public** **int** getId() {
10. **return** id;
11. }
13. **public** **void** setId(**int** id) {
14. **this**.id = id;
15. }
17. **public** **double** getPrice() {
18. **return** price;
19. }
21. **public** **void** setPrice(**double** price) {
22. **this**.price = price;
23. }
25. **public** String execute(){
26. **return** "success";
27. }
29. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd"**>**
7. **<validators>**
9. **<field** name="price"**>**
10. **<field-validator** type="double"**>**
11. **<param** name="minInclusive"**>**100.0**</param>**
12. **<param** name="maxExclusive"**>**9999.9**</param>**
14. **<message>**Price must be between ${minInclusive} to ${maxExclusive}**</message>**
15. **</field-validator>**
16. **</field>**
17. **</validators>**

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd"**>**
3. **<struts>**
4. **<package** name="default" extends="struts-default"**>**
5. **<action** name="register" class="com.javatpoint.Register"**>**
6. **<result** name="input"**>**index.jsp**</result>**
7. **<result>**welcome.jsp**</result>**
8. **</action>**
10. **</package>**
11. **</struts>**

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Product Id:<s:property value="id"/><br/>
4. Product price:<s:property value="price"/>

Struts2 Regex Validation Example

1. [regex validator](https://www.javatpoint.com/struts-2-regex-validation-example)
2. [Parameters of regex validator](https://www.javatpoint.com/struts-2-regex-validation-example)
3. [Example of regex validator](https://www.javatpoint.com/struts-2-regex-validation-example)

The **regex validator** validates the given string with the specified regular expression. It can be used in password, security key etc.

Parameters of regex validator

There is 4 parameters defined for regex validator.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **fieldName** | specifies the field name that is to be validated. It is required in Plain-Validator only. |
| **expression** | specifies the regular expression. |
| **caseSensitive** | specifies if the expression should be matched in case sensitive way. It is true bydefault. |
| **trim** | specifies if the value should be trimmed before matching. It is true bydefault. |

Example of regex validator

1. **<validators>**
2. <!-- Plain Validator Syntax -->
3. **<validator** type="regex"**>**
4. **<param** name="fieldName"**>**data**</param>**
5. **<param** name="expression"**>**[A-Z,a-z,0-9]{5}**</param>**
6. **<message>**data must be alpha numeric of 5 digits**</message>**
7. **</validator>**
9. **</validators>**
10. **<validators>**
11. <!-- Field Validator Syntax -->
12. **<field** name="data"**>**
13. **<field-validator** type="regex"**>**
14. **<param** name="expression"**>**[A-Z,a-z,0-9]{5}**</param>**
15. **<message>**data must be alpha numeric of 5 digits**</message>**
16. **</field-validator>**
17. **</field>**
19. **</validators>**

Full example of regex validator

1) Create index.jsp for input

This jsp page creates a form using struts UI tags. It receives name, password and email id from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. <html>
3. <head>
4. <STYLE type="text/css">
5. .errorMessage{color:red;}
6. </STYLE>
7. </head>
8. <body>
9. <marquee>validation...........</marquee>
10. <s:form action="register">
11. <s:textfield name="data" label="Data"></s:textfield>
12. <s:submit value="register"></s:submit>
13. </s:form>
15. </body>
16. </html>

2) Create the action class

This action class inherits the ActionSupport class and overrides the execute method.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **import** com.opensymphony.xwork2.ActionSupport;
5. **public** **class** Register **extends** ActionSupport{
6. **private** String data;
8. **public** String getData() {
9. **return** data;
10. }
12. **public** **void** setData(String data) {
13. **this**.data = data;
14. }
16. **public** String execute(){
17. **return** "success";
18. }
20. }

3) Create the validation file

Here, we are using bundled validators to perform the validation.

**Register-validation.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
3. <!DOCTYPE validators PUBLIC
4. "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
5. "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
7. <validators>
9. <!--<field name="data">
10. <field-validator type="regex">
11. <param name="expression">[A-Z,a-z,0-9]{5}</param>
12. <message>data must be alpha numeric of 5 digits</message>
13. </field-validator>
14. </field>
16. -->
17. <field name="data">
18. <field-validator type="regex">
19. <param name="expression">[A,a][A-Z,a-z,0-9]{5}</param>
20. <message>data must be alpha numeric of 6 digits and starts with a or A</message>
21. </field-validator>
22. </field>
24. </validators>

4) Create struts.xml

This xml file defines an extra result by the name input, and an interceptor jsonValidatorWorkflowStack.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
3. <struts>
5. <**package** name="default" **extends**="struts-default">
6. <action name="register" **class**="com.javatpoint.Register">
7. <result name="input">index.jsp</result>
8. <result>welcome.jsp</result>
9. </action>
11. </**package**>
13. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. Data is,<s:property value="data"/>

Struts 2 Registration Form Example

1. [Struts 2 Registration Form Example](https://www.javatpoint.com/struts-2-registration-form-example)
2. [Steps to create registration form](https://www.javatpoint.com/struts-2-registration-form-example)

In this example, we are going to create a registration form using struts UI tags and store these information into the oracle database. You may use other database also such as mysql, DB2 etc. according to your requirement.

Let's see the table first that we need to create in the oracle database.

1. CREATE TABLE  "STRUTSUSER"
2. (    "NAME" VARCHAR2(4000),
3. "PASSWORD" VARCHAR2(4000),
4. "EMAIL" VARCHAR2(4000),
5. "GENDER" VARCHAR2(4000),
6. "COUNTRY" VARCHAR2(4000)
7. )
8. /

It will be better for you to create an id for each user. To simply the example, we have not alloted any id with primary key enabled. But you can do it.

The steps to create the registration application in struts2 are as follows:

1. Create input page (index.jsp)
2. Create the action class (RegisterAction.java)
3. Create the class to store data (RegisterDao.java)
4. Map the request in (struts.xml) file and define the view components
5. Create view components

1) Create input page (index.jsp)

It is the simple jsp page that uses struts 2 UI tags to create a form to get input from the user.

**index.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. <s:form action="register">
4. <s:textfield name="name" label="UserName"></s:textfield>
5. <s:password name="password" label="Password"></s:password>
6. <s:textfield name="email" label="Email"></s:textfield>
7. <s:radio list="{'male','female'}" name="gender"></s:radio>
8. <s:select cssStyle="width:155px;"list="{'india','pakistan','other',}"
9. name="country" label="Country"></s:select>
11. <s:submit value="register"></s:submit>
13. </s:form>

2) Create the action class (RegisterAction.java)

This Action class has five fields and one execute method. As we know, struts framework creates instance of the action class per request, we are passing this object in the save method of RegisterDao class.

**RegisterAction.java**

1. **package** com.javatpoint;
3. **public** **class** RegisterAction {
4. **private** String name,password,email,gender,country;
6. //setters and getters
7. **public** String execute(){
8. **int** i=RegisterDao.save(**this**);
9. **if**(i>0){
10. **return** "success";
11. }
12. **return** "error";
13. }
14. }

3) Create the class to store data (RegisterDao.java)

This class gets information from the object of RegisterAction class and stores these information in the strutsuser table.

**RegisterDao.java**

1. **package** com.javatpoint;
2. **import** java.sql.\*;
3. **public** **class** RegisterDao {
5. **public** **static** **int** save(RegisterAction r){
6. **int** status=0;
7. **try**{
8. Class.forName("oracle.jdbc.driver.OracleDriver");
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
12. PreparedStatement ps=con.prepareStatement("insert into strutsuser values(?,?,?,?,?)");
13. ps.setString(1,r.getName());
14. ps.setString(2,r.getPassword());
15. ps.setString(3,r.getEmail());
16. ps.setString(4,r.getGender());
17. ps.setString(5,r.getCountry());
19. status=ps.executeUpdate();
21. }**catch**(Exception e){e.printStackTrace();}
22. **return** status;
23. }
24. }

4) Map the request in (struts.xml) file and define the view components

This xml file contains information about the package, action class and view components.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD
3. Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">
4. <struts>
6. <**package** name="default" **extends**="struts-default">
8. <action name="register" **class**="com.javatpoint.RegisterAction">
9. <result name="success">register-success.jsp</result>
10. <result name="error">register-error.jsp</result>
11. </action>
13. </**package**>
14. </struts>

5) Create view components

Here, we are creating two view components register-success.jsp and register-error.jsp.

**register-success.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. Welcome, <s:property value="name"></s:property>

**register-error.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
2. Sorry, some error occured!
3. <s:include value="index.jsp"></s:include>

|  |
| --- |
| Struts 2 Login and Logout Example   1. [Struts 2 Login and Logout Example](https://www.javatpoint.com/struts-2-login-and-logout-example)   Before creating the login and logout application using struts 2, you must clear the concepts of aware interfaces in struts 2. In this example, we have used the **SessionAware** interface to put the information in the session scope and **ServletActionContext** class to get the information from the session scope.  This example contains three links **login**, **logout** and **profile**. The end user cannot click on the profile page until he/she is logged in. After getting logged in, he/she may go the profile page. If the end user clicks on the logout page, he will not be able to access the profile page.  Here, we assume that you have a table in oracle database named user3333 that contains records. The table query is:   1. CREATE TABLE  "USER3333" 2. (    "ID" NUMBER, 3. "NAME" VARCHAR2(4000), 4. "PASSWORD" VARCHAR2(4000), 5. "EMAIL" VARCHAR2(4000), 6. CONSTRAINT "USER3333\_PK" PRIMARY KEY ("ID") ENABLE 7. ) 8. /   Example of creating login and logout application using struts 2  In this example, we are need following pages   1. **index.jsp** for providing links to the login, logout and profile. 2. **struts.xml** for defining the result and action. 3. **Login.java** for defining login and logout logic. 4. **LoginDao.java** for matching username and password in the database. 5. **Profile.java** for checking if the user is logged in or not. 6. **View components** for the displaying results.   1) Create index.jsp for input  This jsp page creates three links for login, logout and profile.  **index.jsp**   1. <hr/> 2. <a href="login">login</a>| 3. <a href="logout">logout</a>| 4. <a href="profile">profile</a>   2) Define action and result in struts.xml  This xml file defines one package and 4 actions. Each action defines at least one result page.  For the loginprocess and logout actions, we are using the same action class but there invocation methods are different.  **struts.xml**   1. <?xml version="1.0" encoding="UTF-8" ?> 2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD 3. Struts Configuration 2.1//EN" 4. "http://struts.apache.org/dtds/struts-2.1.dtd"> 5. <struts> 6. <**package** name="abc" **extends**="struts-default"> 8. <action name="login"> 9. <result >login.jsp</result> 10. </action> 12. <action name="loginprocess" **class**="com.javatpoint.Login"> 13. <result name="success"  >loginsuccess.jsp</result> 14. <result name="error" >loginerror.jsp</result> 15. </action> 17. <action name="logout" **class**="com.javatpoint.Login" method="logout"> 18. <result name="success" >logoutsuccess.jsp</result> 19. </action> 21. <action name="profile" **class**="com.javatpoint.Profile"> 22. <result name="success" >profilesuccess.jsp</result> 23. <result name="error" >profileerror.jsp</result> 24. </action> 26. </**package**> 27. </struts>   3) Create the action class for login and logout  This action class implements the SessionAware interface and overrides the setSession method to store the information in the session scope.  For logout, we are simply calling the **invalidate()** method of **SessionMap**.  **Login.java**   1. **package** com.javatpoint; 3. **import** java.util.Map; 4. **import** org.apache.struts2.dispatcher.SessionMap; 5. **import** org.apache.struts2.interceptor.SessionAware; 7. **public** **class** Login **implements** SessionAware{ 8. **private** String username,userpass; 9. SessionMap<String,String> sessionmap; 11. **public** String getUsername() { 12. **return** username; 13. } 15. **public** **void** setUsername(String username) { 16. **this**.username = username; 17. } 19. **public** String getUserpass() { 20. **return** userpass; 21. } 23. **public** **void** setUserpass(String userpass) { 24. **this**.userpass = userpass; 25. } 27. **public** String execute(){ 28. **if**(LoginDao.validate(username, userpass)){ 29. **return** "success"; 30. } 31. **else**{ 32. **return** "error"; 33. } 34. } 36. **public** **void** setSession(Map map) { 37. sessionmap=(SessionMap)map; 38. sessionmap.put("login","true"); 39. } 41. **public** String logout(){ 42. sessionmap.invalidate(); 43. **return** "success"; 44. } 46. }   4) Create the Dao class to authenticate user  This class simply validates the user from the table stored in the oracle database.  **LoginDao.java**   1. **package** com.javatpoint; 3. **import** java.sql.Connection; 4. **import** java.sql.DriverManager; 5. **import** java.sql.PreparedStatement; 6. **import** java.sql.ResultSet; 8. **public** **class** LoginDao { 10. **public** **static** **boolean** validate(String username,String userpass){ 11. **boolean** status=**false**; 12. **try**{ 13. Class.forName("oracle.jdbc.driver.OracleDriver"); 14. Connection con=DriverManager.getConnection( 15. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle"); 17. PreparedStatement ps=con.prepareStatement( 18. "select \* from user3333 where name=? and password=?"); 19. ps.setString(1,username); 20. ps.setString(2,userpass); 21. ResultSet rs=ps.executeQuery(); 22. status=rs.next(); 23. }**catch**(Exception e){e.printStackTrace();} 24. **return** status; 25. } 26. }   5) Create the Profile class  This class gets the information from the session scope, if any information is found in the session scope with login name, it returns success otherwise false.  **Profile.java**   1. **package** com.javatpoint; 3. **import** javax.servlet.http.HttpServletRequest; 4. **import** javax.servlet.http.HttpSession; 5. **import** org.apache.struts2.ServletActionContext; 7. **public** **class** Profile { 9. **public** String execute(){ 10. HttpServletRequest request=ServletActionContext.getRequest(); 11. HttpSession session=request.getSession(); 13. String s=(String)session.getAttribute("login"); 14. **if**(s!=**null** && !s.equals("")){ 15. **return** "success"; 16. } 17. **else**{ 18. **return** "error"; 19. } 21. } 22. }   6) Create view components  There are many view components:   * login.jsp * loginsuccess.jsp * loginerror.jsp * logoutsuccess.jsp * profilesuccess.jsp * profileerror.jsp   view components for login  **login.jsp**  This page creates the login form.   1. <jsp:include page="index.jsp"></jsp:include> 2. <hr/> 3. <%@ taglib uri="/struts-tags" prefix="s" %> 5. <s:form action="loginprocess"> 6. <s:textfield name="username" label="Name"></s:textfield> 7. <s:password name="userpass" label="Password"></s:password> 8. <s:submit value="login"></s:submit> 9. </s:form>   **loginsuccess.jsp**  This page prints the welcome message with the username.   1. <jsp:include page="index.jsp"></jsp:include> 2. <hr/> 3. <%@ taglib uri="/struts-tags" prefix="s" %> 5. <br/>Welcome, <s:property value="username"/>   **loginerror.jsp**  This page displays the error message.   1. Sorry username or password error! 2. <jsp:include page="login.jsp"></jsp:include>   view components for logout  **logoutsuccess.jsp**  This page simply displays the successfully logged out message.   1. <jsp:include page="index.jsp"></jsp:include> 2. <hr/> 3. You are successfully logged out!   view components for profile  **profilesuccess.jsp**  This page prints the welcome to profile message.   1. <jsp:include page="index.jsp"></jsp:include> 2. <hr/> 3. <br/>Welcome to profile   **profileerror.jsp**  This page prints the message to login first and includes the login.jsp page.   1. Please login first to see profile 2. <jsp:include page="login.jsp"></jsp:include> |

Struts 2 Fetching all records of a table

1. [Struts 2 Fetching all records of a table](https://www.javatpoint.com/struts-2-fetching-all-records-of-a-table)
2. [Example to fetch and display records](https://www.javatpoint.com/struts-2-fetching-all-records-of-a-table)

To fetch all the records, we have stored all the records in a collection (using List), and displaying the data of the collection using the iterator tag of struts2.

Here, we assume that you have a table in oracle database named user3333 that contains records. The table query is:

1. CREATE TABLE  "USER3333"
2. (    "ID" NUMBER,
3. "NAME" VARCHAR2(4000),
4. "PASSWORD" VARCHAR2(4000),
5. "EMAIL" VARCHAR2(4000),
6. CONSTRAINT "USER3333\_PK" PRIMARY KEY ("ID") ENABLE
7. )

Example to fetch all the records of the table

In this example, we are creating 5 pages :

1. **index.jsp** invoking action.
2. **Register.java** for storing data of the table in the collection.
3. **User.java** for representing table.
4. **struts.xml** for defining the action and result.
5. **welcome.jsp** for the view component to display records.

1) Create index.jsp for invoking action (optional)

This jsp page creates a link to invoke the action. But you can direct invoke the action class.

**index.jsp**

1. <a href="viewrecords">View All Records</a>

2) Create the action class

This action class contains ArrayList object as the datamember and execute method.

**Register.java**

1. **package** com.javatpoint;
2. **import** java.sql.\*;
3. **import** java.util.ArrayList;
5. **public** **class** FetchRecords {
6. ArrayList<User> list=**new** ArrayList<User>();
8. **public** ArrayList<User> getList() {
9. **return** list;
10. }
11. **public** **void** setList(ArrayList<User> list) {
12. **this**.list = list;
13. }
14. **public** String execute(){
15. **try**{
16. Class.forName("oracle.jdbc.driver.OracleDriver");
17. Connection con=DriverManager.getConnection(
18. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
20. PreparedStatement ps=con.prepareStatement("select \* from user3333");
21. ResultSet rs=ps.executeQuery();
23. **while**(rs.next()){
24. User user=**new** User();
25. user.setId(rs.getInt(1));
26. user.setName(rs.getString(2));
27. user.setPassword(rs.getString(3));
28. user.setEmail(rs.getString(4));
29. list.add(user);
30. }
32. con.close();
33. }**catch**(Exception e){e.printStackTrace();}
35. **return** "success";
36. }
37. }

3) Create the class to represent table

This is the simple bean class containing 4 fields.

**User.java**

1. **package** com.javatpoint;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,password,email;
6. //getters and setters
7. }

4) Create struts.xml

This xml file defines action and result.

**struts.xml**

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD
3. Struts Configuration 2.1//EN"
4. "http://struts.apache.org/dtds/struts-2.1.dtd">
5. <struts>
7. <**package** name="anbc" **extends**="struts-default">
8. <action name="viewrecords" **class**="com.javatpoint.FetchRecords">
9. <result name="success">displayrecords.jsp</result>
10. </action>
11. </**package**>
13. </struts>

5) Create view component

It is the simple jsp file displaying the information of the user.

**welcome.jsp**

1. <%@ taglib uri="/struts-tags" prefix="s" %>
3. <h3>All Records:</h3>
4. <s:iterator  value="list">
5. <fieldset>
6. <s:property value="id"/><br/>
7. <s:property value="name"/><br/>
8. <s:property value="password"/><br/>
9. <s:property value="email"/><br/>
10. </fieldset>
11. </s:iterator>