STRUTS 2.0

- Struts is web application framework which will be used to develop the web-based applications fastly and easily.
- Struts is a web application framework implemented by Apache and OpenSymphony.
- It is implemented based on two existing framework Apache struts 1.0 and OpenSymphony webwork.
- New features in struts 2
  1. Interceptors – For applying cross cutting concerns.
  2. Annotations – For reducing XML configuration.
  3. OGNL – For accessing the data from any part of the framework.
  4. New set of tag libraries.
- Struts 2.0 supports two types of configuration
  1. XML based configuration.
  2. Annotation based configuration.
- Like strut 1.x, JSF, Spring MVC, struts 2.0 is also implemented based on two well known design patterns called Front Controller and MVC design pattern.

1. Front Controller design pattern:-
   Problem: I want a centralized access point for presentation tier request handling.

   Description of the problem: Always system requires a centralized access point for request handling. Without centralized access point, code that is common across multiple requests are duplicated in various places. This gives you code duplication problem, because of this maintenance will be increased.

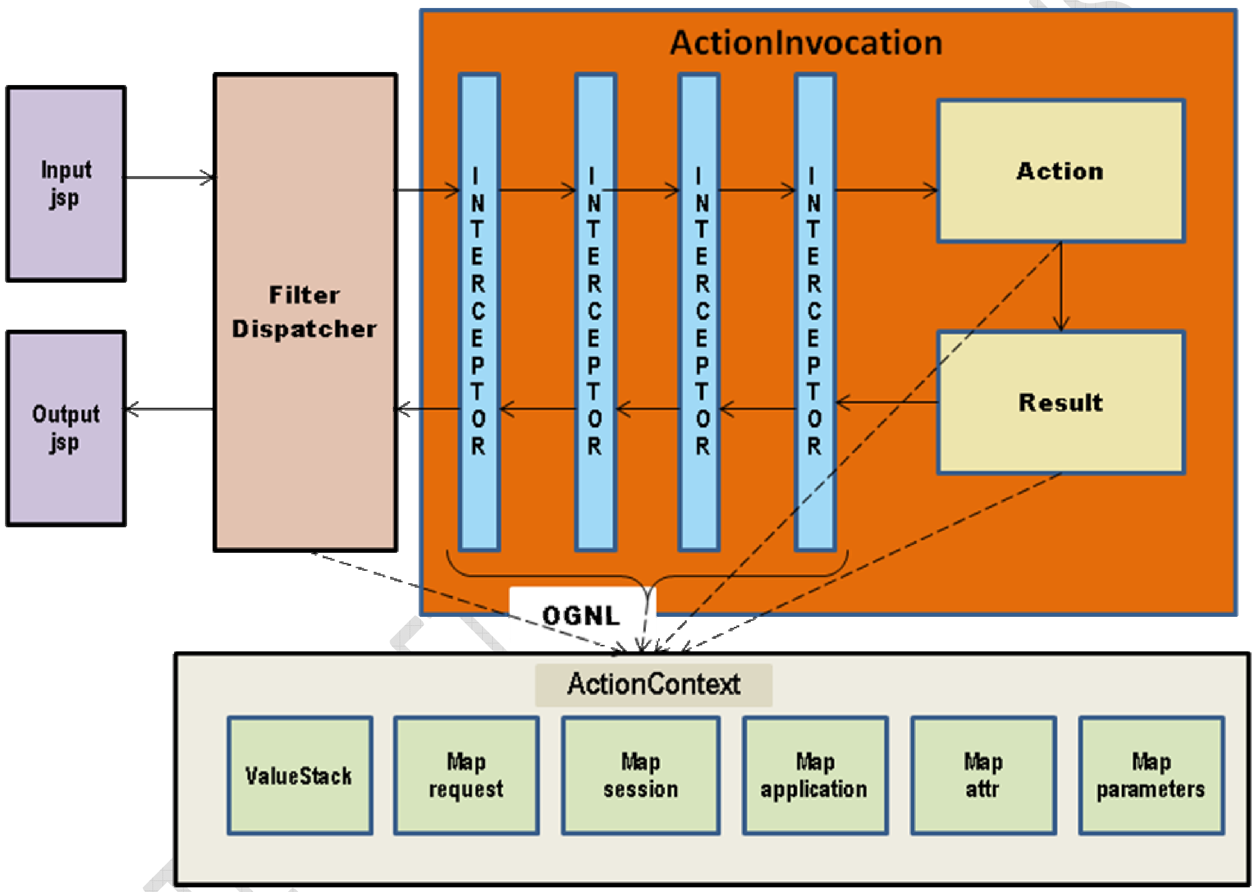
   Solution: Use the Front Controller as the initial point of contact for handling all the requests. Front Controller centralizes controller logic and avoids the code duplication problem.

2. MVC design pattern(Model, View and Controller) :- is used for decoupling business, controller and presentation logic.
   In struts 2, it is implemented by action, JSP and FilterDispatcher.
   1) Controller – FilterDispatcher
      - The role of the controller is played by FilterDispatcher.
      - The controller's job is to map requests to actions.
      - You just need to inform the framework which request URL maps to which of your actions. You can do this with XML based configuration files, or with Java annotations.
   2) Model– Action
      - Model is the application logic and contains both data and business logic.
   3) View – JSP
      - This page is the user interface which presents a representation of the application's state to the user.

Because of MVC design pattern we have 3 layers in struts based application.

I. Presentation Layer.
II. Controller Layer.
III. Model Layer.

Struts 2.Architecture and request processing diagram



Note: OGNL targets ValueStack as the default object for accessing or processing values. If any value is required from any other object then we should explicitly specify the map object name.

- Following are the various components available in struts 2 architecture.

1. FilterDispatcher
2. Interceptors.
3. Actions.
4. ActionContext.
5. ValueStack.
6. OGNL (Object Graph Navigation Language).
7. Results
8. ActionInvocation.

❖ <u>Struts 2 Request processing</u>

1. When client submits the request, FilterDispatcher receives the request first.
2. FilterDispatcher identifies the following things
   a) Corresponding action for the incoming request URL.
   b) The set of interceptors that has to be invoked for the current action and prepares the stack of interceptors.
   c) The view for the current request.
3. FD now creates a new object of ActionInvocation and adds all the action, interceptors and result into the ActionInvocation object and delegates the request processing to the ActionInvocation object.
   Note: A new object of ActionInvocation gets created for every request.
4. ActionInvocation invokes all the interceptors in the stack one by one.
5. ActionInvocation **<u>starts the life cycle of identified action.</u>**
6. Once action class execute() method is completed all the interceptors invoked previously will be invoked now in the reverse order.
7. Once ActionInvocation gets the result object and identifies the corresponding jsp for the result and forwards the control to that jsp.
8. FilterDispatcher destroys the ActionInvocation object.

❖ <u>Life cycle of Action</u>

1. Action class will be loaded.
2. Action class will be instantiated.
3. If Action class is implementing ServletRequestAware interface then setServletRequest() method will be called.
4. If your Action is implementing ServletResponseAware interface then setServletResponse() method will be called.
5. If your Action is implementing ParameterAware interface then setParameters() method will be called.
6. If your Action is implementing RequestAware interface then setRequest() method will be called.
7. If your Action is implementing SessionAware interface then setSession() method will be called.
8. If your Action is implementing ApplicationAware interface then setApplication() method will be called.
9. If your Action is implementing Preparable interface then prepare () method will be called.
   (This method is used for writing pre-processing or resource initialization code)
10. If your Action is implementing ModelDriven interface then getModel () method will be called.
11. Request parameters will be collected and will be stored or set into the properties (Action class or separate java bean class) by calling setter methods.
12. If your Action is implementing Validatable interface then validate() method will be called.
13. After completion of validate() method, error messages will be verified. If any action level errors or field level errors are added then control will return to input jsp.
14. If no error messages are added then next component (either interceptor method or action class execute() method) will be invoked.

❖ Identifying the Action

1) In jsp

```
<html>
<body bgcolor="orange">
<s:form action="cluster/doLogin" method="GET">
</body>
</html>
```

2) in url request

$$\texttt{http://localhost:8090/myapp/}\ \underbrace{\texttt{cluster/}}_{\text{package namespace}}\ \ \underbrace{\texttt{doLogin.action}}_{\text{action name.action}}$$

3) in struts.xml file

```
<struts>
<package name="action" namespace="/cluster" extends="struts-default">
   <action name="doLogin" class="com.cluster.action.CheckUser" >
       <result name="failure">/jsp/loginFailure.jsp</result>
       <result name="success">/jsp/loginSuccess.jsp</result>
   </action>
    </package>
</struts>
```

1. Upon receiving the request , FilterDispatcher takes the package namespace associated with the current url request and verifies in the struts.xml file.
2. If namespace is not found with the current url then FilterDispatcher tries to process previous request package namespace.
3. If that also not found then errors will be given to client.
4. If package namespace found, then that package will be verified to find the action with the action name which is associated with the current request.
5. If action is not found in the package then error message will be given to client.
6. If action is found then action class will be taken and its life cycle will be started.

- **<u>Following are the various components of struts 2 framework and their corresponding roles.</u>**

1. FilterDispatcher

- The role of the controller is played by FilterDispatcher.
- It creates an object of ActionInvocation which encapsulates the execution process of an action with its associated interceptors and results. It then delegates the request processing to ActionInvocation.

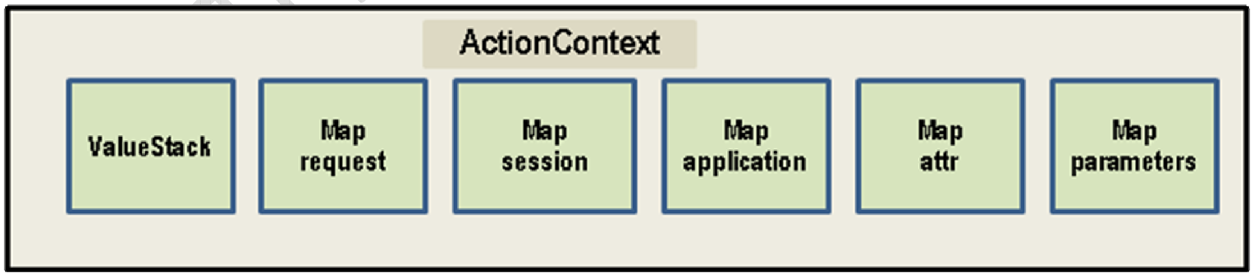
2. Interceptors.

- Struts 2 uses interceptors to both separate and reuse crosscutting concerns from the core request processing.
- There is a stack of interceptors in front of the action.
- The invocation of the action must travel through this stack.
- These interceptors are invoked both before and after the action.

3. Actions.

- Action class contains both data and business logic and is always placed inside the ValueStack.
- Action will always do three things.
  1) It contains the logic to be executed for a particular request.
  2) It stores and carries the data so that we can access the data from the request till the view.
  3) It decides the view for a particular request.
- The Action serves as a centralized data transfer object that can be used to make the application data available in all tiers of the framework.
- The use of Actions as data transfer objects should probably ring some alarms in the minds of alert Struts 1 developers. In Struts 1, the action classes were singletons i.e. there is only one instance of the Action class.
  In a multi-threaded environment, such as a web application, it would be very problematic to store data as instance fields in the Action class.
- But struts 2 solves this problem by creating a new instance of an action for each request that maps to it. This allows Struts 2 objects to exist as dedicated data transfer objects for each request.
- Even if we are concerned about the performance issue because a new action object is created for every request, just consider the fact that even with a singleton action object we still would have to create some sort of data transfer object for the request. So, object instantiation is certainly not higher in Struts 2.

4. ActionContext.

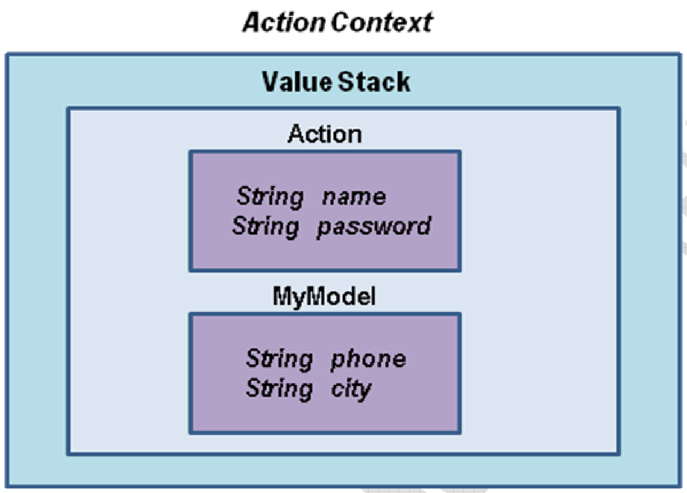


- The ActionContext holds all the important data objects related to a given action invocation and OGNL can target any of them.

- Following are the different objects of ActionContext.

Name	Description
1) Parameters	Map of request parameters.
2) Request	Map of request attributes.
3) Session	Map of session attributes.
4) Application	Map of context attributes.
5) Attr	Gives value of an attribute starting from page, request, session or application.
6) ValueStack	Contains the Action class object (properties or data of the action class) and domain data (Model data)

5. ValueStack.



- The ValueStack is one of the object in ActionContext.
- The Action class object which is the carrier of data is placed in the ValueStack so that its properties or data will be accessible via OGNL. If the action class is implementing ModelDriven interface then the domain object (TO) will be stored in the ValueStack.
- The ValueStack is the default object against which all OGNL expressions are resolved.
- Struts 2 uses the ValueStack as a storage area for all application domain data that will be needed during the processing of a request.
- When a request hits the framework, one of the first things Struts 2 does is it will store all the important data for the request in the ValueStack.
- Rather than passing the data around, Struts 2 keeps it in a convenient, central location – the ValueStack.
- Data is moved to the ValueStack in preparation of request processing, it is manipulated there during action execution, and it is read from there when the results render their response pages.

6. OGNL (Object Graph Navigation Language).

- OGNL is a powerful technology that has been integrated in struts 2 framework to help with data transfer i.e either to set the properties from the incoming parameters or for rendering values in the jsp.
- All OGNL expressions must resolve against one of the objects contained in the ActionContext. By default, the ValueStack will be chosen for OGNL resolution, but we can specifically choose other objects such as session map or application map etc.
- The ValueStack is the default object against which all OGNL expressions are resolved if no specific object is chosen.
- If we want to target other objects using OGNL then we must specify the object as follows.

  #session.STRNAME

  or

  #session['STRNAME']

  Here OGNL tries to resolve the value from session map object.

7. Results.

- This is the jsp or the user interface which presents a representation of the application's state to the user.
- We can use the struts 2 tag libraries here for displaying the jsps.
- The struts 2 tag libraries are divided into two groups general purpose tags and UI component tags. General purpose tags provide support for all sort of things from conditional logic to ValueStack manipulation, etc.

  The UI component tags generate only HTML form fields.

8. ActionInvocation

- Whenever a request is received the FilterDispatcher creates an object of ActionInvocation which encapsulates the Action, all of the interceptors and the result which has to be executed for a particular request.

What is package tag in struts.xml?

- The package element is used for organizing your actions, results and other component elements.
- It also contains a namespace element which is used to map the current request url.

What is `struts-default` value in the `<package>` tag and why does the developer extend it?

- Many of the predefined components are declared in struts-default.xml, found in struts2-core.jar.
- This xml file contains the `struts-default` package. (This package contains `defaultStack`)
- The developer generally extends this package to reuse the built in components of the framework. When we do not extend this package, we are rejecting many features of the framework. The struts-default package, defined in the struts-default.xml, contains all the commonly needed Struts 2 components ranging from complete interceptor stacks to all the common result types. Most of the interceptors that we use are found in the *struts-default* package which is in the struts-default.xml file.
- By not extending the *struts-default* we would be losing lots of functionality of the struts framework.

- <u>**Struts 2 API has one main interface and one class.**</u>

1)
```
public interface Action {
   public static final String ERROR = "error";
   public static final String INPUT = "input";
   public static final String LOGIN = "login";
   public static final String NONE = "none";
   public static final String SUCCESS = "success";

   String execute();
}
```
When our action class implements Action interface then we are unable to make use of struts 2 existing functionality like validation, internationalization, accessing message bundles etc.

2)
```
public class ActionSupport implements
Action,Validateable,ValidationAware,TextProvider,LocaleProvider  {

   String execute();
   addActionError(String anerrormessage);
   addFieldError(String fieldname, String errormessage);
   getText(String textname);
   validate();
}
```
When our action class extends ActionSupport then we get lots of functionality like validation, internationalization, accessing message bundles etc.

❖ <u>**Writing an action class in struts 2**</u>

- In struts 2 you can implement action in 3 ways:-
  1. Writing POJO without extending or implementing any class.
  2. Writing action by implementing Action interface which is available in com.opensymphony.xwork2 package.
  3. Writing an action by extending ActionSupport class which is available in com.opensymphony.xwork2 package.

1. Writing POJO without extending or implementing any class
   - In this approach your action class will not extend or implements any built-in components.
   - In this approach your action must override execute() method with the following signature.

```
public class CheckUser
{
         public String execute(){}
}
```
- With this approach we have following limitations

1) We cannot access locale directly which is associated with the current request.
2) Your action cannot perform any validations.
3) Your action cannot access message bundles directly.
4) Your action cannot add action level or field level error messages.

2. Implementing Action Interface
   - In this approach your action implements Action interface and overrides execute() method.
   - In this approach also we have various limitations which are same as previous one.

```
public class CheckUser implements Action
{
      public String execute(){}
}
```

3. Extending ActionSupport Class

- When your class extends ActionSupport then your class can access overidden methods of interfaces called Action, Validatable, ValidationAware, TextProvider, LocaleProvider

```
public class CheckUser extends ActionSupport
{
      public String execute()
      {}
}
```

- This is considered to be the best approach since we get most of the features of the framework like validation, internationalization, accessing message bundles etc.

How can I access ServletRequest in an Action?

1. Your action class has to implement ServletRequestAware interface.
2. Declare HttpServletRequest type variable in action class and write the setter method.

❖ **Transferring data into domain objects**

- Struts 2 will automatically transfer request data parameters onto the properties that your Action class exposes.
- The params interceptor, in the `defaultStack` interceptor stack, automatically transfers data from the request parameters onto our action properties. The developer needs only to provide properties of the actions with the same names as the form input fields which are used for submitting the request.
- The problem with this mechanism is after receiving these request values into the action properties, we will have to explicitly create our own domain object and then transfer the data from the action properties to the domain object.
- But struts 2 provides a mechanism, to automatically expose the domain object to the data transfer mechanism and automatically initialize the domain object.
- There are two approaches for doing this
  1. Object backed properties.
  2. Model driven actions

1. Using an object backed property to receive deep data transfers

```
    public class CheckUser extends ActionSupport {

        UserTO userTO = null;

        public UserTO getUserTO() {
                return userTO;
        }
        public void setUserTO(UserTO userTO) {
                this.userTO = userTO;
        }
        public String execute() {}
}
```

- In this approach we can expose the domain object as a property and we should have setter/getter methods for the domain object.
- The problem with this approach is we have to use deeper notation to access the values from the domain object.
  Eg in the jsp to collect the input parameters.

```
<s:textfield name="userTO.username" key="login.username"/>
```

2. Using ModelDriven action

```
public class CheckUser extends ActionSupport implements
ModelDriven,Preparable {

        UserTO userTO;
        //userTO = new UserTO(); 1st way
        public void prepare() throws Exception { //2nd way
                System.out.println("Inside prepare()");
                userTO = new UserTO();
        }

        public Object getModel() {
                System.out.println("Inside getModel()");
                return userTO;
        }
```

- In the model driven action approach we have to declare the domain object as the property of the action class but there is no need of writing the setter/getter method for it.
- The domain object is exposed via the getModel() method.
- In the model driven actions approach we again have two approaches to initialize the domain object
  1) Explicitly the developer has to create the instance of the domain object.
  2) We can implement the Preparable interface and override the prepare() method.

- The advantage of using ModelDriven action approach is we can now access the values in a simpler way and there is no deeper access notation.

  Eg in the jsp to collect the input parameters.

```
<s:textfield name="username" key="login.username"/>
```

❖ **Accessing message bundles or i18n (Internationalization)**

- When developing any web application, we get the following requirement.
  1. Centralizing all the messages in message bundles.
  2. Supporting multiple languages.
  3. Centralizing error messages.
- Srtuts 2 supports all the above requirements.
- ActionSupport is implementing LocalProvider and TextProvider interfaces. Every time the action class receives a request, first it invokes the getLocale() method of the LocaleProvider. Once it gets the locale, then the action class calls the getText() method and retrieves the text message from the appropriate message bundles or properties file (Eg hindi, English, kannada, etc)

Following are the steps to implement the above requirement.

Step 1) Write a properties file with key and value pairs as follows.

```
login.title=Welcome to home page
login.username=Enter UserName
login.login=Login
login.username.required= Please enter your Username
login.username.required = $\{getText(login.username)} is mandatory.
```

For other languages write another properties file.
Eg: For Hindi write ApplicationResources_hi.properties
For Kannada write ApplicationResources_kn.properties

Step 2) Register the properties files in the framework as follows.

```
<constant name="struts.custom.i18n.resources"
            value="MyApp.ApplicationResources_hi" />
<constant name="struts.custom.i18n.resources"
            value="MyApp.ApplicationResources_kn" />
```

Step 3) Use the messages in your application in various components as follows.

a) Using messages inside jsp.

```
1) <s:text name="login.title"/>
2) <s:textfield name="username" key= ="login.username" />
3) <s:submit method="execute" key="login.login"/></h2>
```

b) Using in action class.

```
1) addFieldError("username", getText("login.username.required"));
2) addActionError(getText("login.username.required"));
```

Note : Both are methods present in ActionSupport class.

c) Using messages when we are doing validation.

1) in validation.xml file.

```
<field-validator type="requiredstring">
   <message key="login.username.required"></message>
</field-validator>
```

2) in properties file

```
login.username.required = $\{getText(login.username)} is mandatory.
```

d) Using the messages in custom validator class.

```
public class CustomValidator extends FieldValidatorSupport {
      ActionContext actionContext = ActionContext.getContext();
      String msg = actionContext.get("login.username").toString();
}
```

❖ Struts 2 Validation framework

- You can perform validations in two different ways

1. Struts 2 basic validations
2. Struts 2 framework validations

1. Struts 2 basic validations

You need to do the following to perform basic validations

Step 1: Your action also must extend ActionSupport class and you must override validate() method in your action class.

a) Verify the client submitted data to check whether it is following the given validation rules or not.

b) If not following, invoke the addFieldError() method with field name and the key specified in the property file or type the message.

```
public class CheckUser extends ActionSupport {

      public String execute() {}

      public void validate()
      {
1a          if (getUsername().length() == 0)
            {
1b                addFieldError("username", "Please enter the Username");
            }
            else if (getPassword().length() == 0)
            {
                  addFieldError("password",
1b                getText("login.password.required"));
            }
      }
   }
```

Step 2) Your action configuration should have a result with name input (strus-config file). This is the page which gets displayed if validation fails.

```
<result name="input">/jsp/login.jsp</result>
```

Step 3) Write the error message in the property file with the corresponding key

```
login.password.required= Please enter your Password
```

2. Struts 2 framework validations

- When we are developing any web based application we need to perform the validation in client submitted data.
- Already we have seen the way to implement validations by overriding the validate() method in the action class.
- validate() method has the following limitations:-

1. Developer is responsible to write the validation code inside the validate() method for all action classes.
2. Sometimes you may write the same validation code in multiple validate methods belonging to multiple action classes. This gives code duplication problem.

Struts 2 does the validation in the following way.

1. The params interceptor will move the request parameters onto our action object.
2. Next the workflow interceptor is fired for checking basic validations. This interceptor does two things. First it checks if the action is implementing Validateable interface. If it implements then the workflow interceptor will invoke the validate() method for basic validations. Second if the validation fails then an error message is created and added. Then the workflow interceptor will check if there are any error messages. If the error exists, the workflow interceptor will stop execution, returns a control string with the a value "input" and displays the input jsp.
3. If we are using the validation framework then validation interceptor will be fired and adds the error messages if any validation fails. Then the workflow interceptor will check if there are any error messages. If the error exists, the workflow interceptor will stop execution, returns a control string with the a value "input" and displays the input jsp.

You can do framework validation in 2 ways

I. xml based validation
II. Annotation based validation

I. xml based validations.

- Following are the list of built-in validators

  1) required 2) requiredstring 3)int 4)double 5)date 6)expression 7)fieldexpression 8)email 9)url 10)visitor 11)conversion 12)stringlength 13)regex

- These built-in validators are declared in xwork-core-2.jar file in default.xml file

Steps for doing xml validations using validation framework.

Step 1: Write the jsp as follows

```
<html>
      <body bgcolor="lightblue">
            <s:form action="doLogin" method="GET">
                  <s:textfield name="username" key="login.username"/>
                  <s:submit method ="execute" key="login.login"/>
                  </s:form>
      </body>
</html>
```

Step 2: Write the required validation in the validation.xml file. The naming convention of the xml file is ActionClass-validation.xml file. (Eg: CheckUser-validation.xml). This xml file should be placed in package directory structure of the action class itself.

```
   <validators>
      <field name="username">
            <field-validator type="requiredstring">
                  <message key="login.name.required"></message>
            </field-validator>
      </field>
   </validators>
```

Step 3: Your action configuration should have a result with name input (strus-config file). This is the page which gets displayed if validation fails.

```
   <result name="input">/jsp/login.jsp</result>
```

Step 4: Write the error message in the property file with the corresponding key

```
      login.name.required= Please enter your Name
```

Note: We use `short-circuit = "true"` attribute in the `<field-validator>` tag for short circuiting the validations i.e. if the first validation fails then second will not be applied.

II. Annotation based validations

Following are the list of built-in validators

1) ConversionErrorFieldValidator 2)CustomValidator 3) DateRangeFieldValidator 4) EmailValidator 5) ExpressionValidator 6) FieldExpressionValidator 7)IntRangeFieldValidator 8) RegexFieldValidator 9)RequiredFieldValidator 10) RequiredStringValidator 11)StringLengthFieldValidator 12) UrlValidator 13) Validation 14)ValidationParameter 15)Validations 16)ValidatorType 17)VisitorFieldValidator

Steps for doing annotation validations using validation framework.

Step 1: Write the jsp as follows

```
<html>
      <body bgcolor="lightblue">
            <s:form action="doLogin" method="GET">
                  <s:textfield name="username" key="login.username"/>
                  <s:submit method ="execute" key="login.login"/>
                  </s:form>
      </body>
</html>
```

Step 2: Write the required validation in the Action class itself. The validations should be applied at setter methods of the field.

```
@RequiredStringValidator(type = ValidatorType.FIELD, key =
"login.username.required", shortCircuit = true)
@StringLengthFieldValidator(type = ValidatorType.FIELD, minLength = "5",
maxLength = "10", key = "login.length")

public void setUsername(String username) {
            this.username = username;
            }
```

Step 3: Your action configuration should have a result with name input (strus-config file). This is the page which gets displayed if validation fails.

```
   <result name="input">/jsp/login.jsp</result>
```

Step 4: Write the error message in the property file with the corresponding key

```
      login.username.required= Please enter your Name
      login.length=Your Password is nice, a valid password must be between
      ${minLength} and ${maxLength} characters long
```

<u>Steps for doing validations of the domain object using validation framework.</u>

- This approach is the most recommended approach because we can reuse a TO for many action classes.

Step 1: Write the jsp as follows

```
<html>
      <body bgcolor="lightblue">
            <s:form action="doLogin" method="GET">
                  <s:textfield name="username" key="login.username"/>
                  <s:submit method ="execute" key="login.login"/>
                  </s:form>
      </body>
</html>
```

Step 2: Write a validation.xml file for the action class but this will not contain any validation logic. It will only contain the code to delegate the validation to the domain object or TO's validation.xml file. This is done with the help of "visitor" validator. The naming convention of the xml file is ActionClass-validation.xml file. (Eg: CheckUser-validation.xml). This xml file should be placed in package directory structure of the action class itself.

```
   <validators>
      <field name="model">
            <field-validator type="visitor">
                  <param name="appendPrefix">false</param>
                  <message>Hello</message>
            </field-validator>
      </field>
  </validators>
```

Step 3: Write another validation.xml file which contains the validation logic. This file should be placed in package directory structure of the TO or domain object. The naming convention of this file will be - TOname-validation.xml (Eg: CheckUserTO-validation.xml)

```
<validators>
      <field name="username">
            <field-validator type="requiredstring">
                  <message key="login.name.required"></message>
            </field-validator>
      </field>
</validators>
```

Step 4: Your action configuration should have a result with name input (struts-config file). This is the page which gets displayed if validation fails.

```
   <result name="input">/jsp/login.jsp</result>
```

Step 5: Write the error message in the property file with the corresponding key

```
      login.name.required= Please enter your Name
```

Steps to write custom validations

1. Write the validator class by extending FieldValidatorSupport class which is available in com.opensymphony.xwork2.validator.validators package or by implementing FieldValidator interface which is available in com.opensymphony.xwork2.validator package
2. Override the validate (Object obj) method.
3. Implement the required validation logic inside the validate() method

```java
public class PasswordIntegrityValidator extends FieldValidatorSupport {

public void validate(Object object) throws ValidationException {
      // implement the validation logic
}
}
```

4. Register your custom validator by writing the validator tag inside the validators.xml. This validators.xml file must be under src folder.

```xml
<validators>
<validator name="passwordintegrity"
class="com.cluster.customvalidate.PasswordIntegrityValidator"/>
</validators>
```

5. Use the validator now as xml or as annotation

a) Validating using xml.

```xml
<validators>
      <field name="password">
      <field-validator type="passwordintegrity">
            <message key="login.invalidFormat"></message>
      </field-validator>
      </field>
</validators>
```

b) Validating using annotation

```java
@CustomValidator(type = "passwordintegrity", fieldName = "Password",
key = "login.invalidFormat")

public String getPassword() {
      return password;
}
```

6. Write the properties file.

```
login.invalidFormat= Your password must contain one letter and one number
```

❖ Handling exceptions in struts 2

1. Write your userdefined exception.

   Eg:-
```
public class UserNotFoundException extends RuntimeException{
}
```

2. Whenever u get the errors in Action class throw the exception and also add the ActionError message.

   For eg:

```
public class CheckStudent extends ActionSupport implements ModelDriven{

	public String execute() throws Exception {
		try {
			StudentService studentService = new StudentService();
			List list = studentService.getAllPhones(studentTO);
			return "success";
		} catch (Exception e) {
			addActionError(getText("login.nophone"));
			throw new UserNotFoundException("No phones");
		}
	}
}
```

3. Write your `<exception-mapping>` tag in struts.xml file as follows.

```
<global-results>
     <result name="red">/jsp/noPhones.jsp</result>
</global-results>

<global-exception-mappings>
<exception-mapping exception =
"com.cluster.exception.UserNotFoundException" result="red"/>
</global-exception-mappings>
```

4. Write the ActionError message in the property file for the given key.

```
login.nophone=No phone numbers to display
```

5. Add the `<s:actionerror/>` tag in the jsp to display the error management

```
<body bgcolor="wheat">
		<h2><s:actionerror/></h2>
</body>
```

❖ Struts 2 Tiles

index.jsp

```
<html>
<body bgcolor="yellow">
<table border="2">
      <tr height="10%">
            <td><jsp:include page="header.jsp" /></td>
      </tr>
      <tr height="80%">
            <td>
            <table border="2">
                  <tr>
                        <td><jsp:include page="menu.jsp" /></td>
                        <td><jsp:include page="home.jsp" /></td>
                  </tr>
            </table>
            </td>
      </tr>
      <tr height="10%">
            <td><jsp:include page="footer.jsp" /></td>
      </tr>
</table>
</body>
</html>
```

- When we are developing a web based application we can write many jsps inside that and in every jsp we can have some design layout.
- Sometimes we can have different design layouts for different jsps.
- We can develop this kind of layout using <jsp:include> but the problem is:-
  1. Hard coding the jsp names inside the jsp. Because of hardcoding when we change one jsp name we need to modify all the jsps in the application, which gives maintenance problem.
  2. We will be duplicating the layout code in all the jsps. This gives code duplication problem.
- To solve the above problems apache has provided a special framework called tiles framework.

Steps to integrate tiles in struts 2

1. Enable the tiles framework by writing the following is web.xml

```
<web-app >
<context-param>
      <param-name>tilesDefinition</param-name>
      <param-value>/WEB-INF/tiles.xml</param-value>
</context-param>
<filter>
      <filter-name>struts2</filter-name>
      <filter-class> org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
</filter>
```

```
<filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
      <listener-class>org.apache.struts2.tiles.StrutsTilesListener
      </listener-class>
</listener>
 </web-app>
```

2. Design the layout jsp as follows.

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<html>
      <body>
            <table width="80%" height="90%" align="center">
                  <tr>
                        <td height="10%" bgcolor="blue" colspan="3">
                              <tiles:insertAttribute name="header"/>
                        </td>
                  </tr>

                  <tr>
                        <td width="20%" bgcolor="green">
                              <tiles:insertAttribute name="lmenu"/>
                        </td>

                        <td width="60%" bgcolor="red">
                              <tiles:insertAttribute name="body"/>
                        </td>

                        <td width="20%" bgcolor="pink">
                              <tiles:insertAttribute name="rmenu"/>
                        </td>
                  </tr>

                  <tr>
                        <td height="10%" bgcolor="yellow" colspan="3">
                              <tiles:insertAttribute name="footer"/>
                        </td>
                  </tr>
            </table>
      </body>
   </html>
```

3. Develop the other content jsps (header.jsp, footer.jsp, menu.jsp, etc)

4. Write the tiles definition in tiles.xml file.

```
<tiles-definitions>

 <definition name="BaseDef" template="/jsp/clusterlayout.jsp">
 <put-attribute name="header" value="/jsp/header.jsp"/>
 <put-attribute name="lmenu" value="/jsp/lmenu.jsp"/>
 <put-attribute name="body" value=""/>
 <put-attribute name="rmenu" value="/jsp/rmenu.jsp"/>
 <put-attribute name="footer" value="/jsp/footer.jsp"/>
 </definition>


 <definition name="HomeDef" extends="BaseDef">
     <put-attribute name="body" value="/jsp/welcome.jsp" />
 </definition>
</tiles-definitions>
```

5. Include the above definition in the jsp.

```
<html>
      <body>
            <tiles:insertDefinition name="HomeDef"/>
      </body>
</html>
```

❖ Interceptors

- Struts 2 uses interceptors to separate and reuse commonly required tasks or crosscutting concerns for all action classes.
- Some interceptors only do work before the action has been executed, and others only do work after request processing.
- Interceptors don't necessarily have to do something both times they fire but they do have the opportunity.
- There is a stack of interceptors in front of the action.
- The invocation of the action must travel through this stack.
- The built-in interceptors handle most of the fundamental tasks ranging from data transfer, validation, exception handling, etc.
- We generally always extend the *defaultStack* and by extending this we inherit many features of the struts framework.
- While the *defaultStack* provides a common set of interceptors, arranged in a common sequence, to serve the common functional needs of most requests, we can easily re-arrange them to meet varying requirements. We can even remove and add particular interceptors as we like. We can even do this on a per action basis, but this is seldom necessary.
- Most of the interceptors that we use are found in the *struts-default* package which is in the struts-default.xml file.

- Following are few of the list of built-in interceptors which are configured in *struts-default* package of struts-default.xml

1. alias
2. autowiring
3. chain
4. conversionError
5. cookie
6. createSession
7. debugging
8. externalRef
9. execAndWait
10. exception
11. fileUpload
12. i18n
13. logger
14. modelDriven
15. scopedModelDriven
16. params
17. prepare
18. staticParams
19. scope
20. servletConfig
21. sessionAutowiring
22. timer
23. token
24. tokenSession
25. validation
26. workflow
27. store
28. checkbox
29. profiling
30. roles

- There are many stacks defined in the *struts-default* packages of which two are important.
  1. *basicStack*
  2. *defaultStack*
- Struts also provides the facility to implement your own interceptors according to your application requirement.

Following are the steps to write a custom interceptor

Step 1: Write your own interceptor class by implementing Interceptor interface which is available in com.opensymphony.xwork2.interceptor package or by extending AbstractInterceptor class.

```
public interface Interceptor {
void init();
void destroy();
String intercept(ActionInvocation invocation) throws Exception;
}
```

Step 2: Implement the required logic in your custom interceptor class in the intercept() method and you should have to call the invoke() method which is responsible to invoke the next component in the stack (the next component can be another interceptor or action class). The other two are lifecycle methods which are used for initializing and cleaning up of resources.

```
public class MyInterceptor extends AbstractInterceptor {
      public void init(){
            //resource initialization code
      }
      public String intercept(ActionInvocation actionInvocation) {
      //pre proccessing code
      String result = actionInvocation.invoke();//call the invoke() method.
      //post processing code
      return result;
      }
      public void destroy(){
            //resource clean up code
      }}
```

Step 3: The intercept method should return a String value which will be used by another interceptor.

Step 4: Register the interceptor implemented by you in the `<package>` tag using the tag `<interceptors>` and use the interceptor for your action class as follows :-

```
<package name="action" namespace="/cluster" extends="struts-default">
<interceptors>
      <interceptor name="myIntercep"
      class="com.cluster.interceptor.MyInterceptor" />

            <interceptor-stack name="myStack">
                  <!--  <interceptor-ref name="defaultStack"/> -- > 1 option

                        <interceptor-ref name="exception" />          }
                        <interceptor-ref name="servletConfig" />      }
                        <interceptor-ref name="i18n" />               }
                        <interceptor-ref name="chain" />              }
                        <interceptor-ref name="params" />             }
                        <interceptor-ref name="validation">           }
                        <param name="excludeMethods">                 } 2 option
input,back,cancel,browse,retrieve,previous</param>                    }
                        </interceptor-ref>                            }
                        <interceptor-ref name="workflow">             }
                        <param name="excludeMethods">                 }
input,back,cancel,browse</param>                                      }
                        </interceptor-ref>                            ↓

                        <interceptor-ref name="myIntercep" />
                  </interceptor-stack>
            </interceptors>

            <default-interceptor-ref name="myStack" />

            <action name="ShowLogin">
                  <result>/jsp/login.jsp</result>
            </action>

            <action name="doLogin" class="com.cluster.action.CheckUser">
                  <result name="FAILURE">/jsp/loginFailure.jsp
                  </result>
                  <result name="SUCCESS">/jsp/loginSuccess.jsp
                  </result>
            </action>
      </package>
```

Note: The `<default-interceptor-ref name="myStack" />` tag is used for reusing the stack.

❖ **Studying the built-in interceptors.**

1. params :- interceptor collects the incoming parameters and transfers into properties of the action class and then stores the action class in the ValueStack.

2. conversionError:- interceptor is used for converting the HTTP string values into the corresponding java primitive or wrapper classes type. If any HTTP string value cannot be converted to a java type then default error message will be reported which can be customized to display our own message.

3. workflow :- interceptor checks for basic validations. This interceptor does two things. First it checks if the action is implementing Validateable interface. If it implements then the workflow interceptor will invoke the validate() method for basic validations. Second if the validation fails then an error message is created and added. Then the workflow interceptor will check if there are any error messages. If the error exists, the workflow interceptor will stop execution, returns a control string with the a value “input” and displays the input jsp.

```
public String intercept(ActionInvocation invocation)
   throws Exception {

   Action action = invocation.getAction();
   if (action instanceof Validateable) {
      Validateable validateable = (Validateable) action;
      validateable.validate();
|#2
   }
   if (action instanceof ValidationAware) {
      ValidationAware validationAwareAction =
      ValidationAware) action;
|#3

      if (validationAwareAction.hasErrors()) {
         return Action.INPUT;
|#4
      }
   }
   return invocation.invoke();
}
```

4. validation:- interceptor is used for using validation framework. If we are using the validation framework then validation interceptor will be fired and adds the error messages if any validation fails. Then the workflow interceptor will check if there are any error messages. If the error exists, the workflow interceptor will stop execution, returns a control string with the a value “input” and displays the input jsp. To use this interceptor your action must implement ValidationAware interface.

5. exception:- This important interceptor is responsible for doing exception handling in the application. The exception interceptor comes first in the defaultStack, and should probably come first in any custom stacks you might create yourself. The exception interceptor will catch exceptions and map them, by type, to user defined error pages. It's position at the top of the stack guarantees that it will be able to catch all exceptions that may be generated during all phases of the action invocation.

6. prepare:- interceptor looks for prepare() method and executes the method if your action class implements Preparable interface. We generally override prepare() method if we want to write any pre-processing or initializing code.

7. modelDriven:- This interceptor checks whether the action class is implementing ModelDriven or not. If implementing then getmodel() will be called.

8. token and tokenSession:- The token and token-session interceptors can be used to prevent duplicate form submissions. Duplicate form posts can occur when users click the Back button to go back to a previously submitted form and then click the button again, or when they click the button more than once while waiting for a response. The token interceptors work by passing a token in with the request which is checked by the interceptor. If the unique token comes to the interceptor a second time, the request is considered as a duplicate. These two interceptors both do the same thing, differing only in the richness of their handling of the duplicate request. You can either show an error page or save the original result to be re-rendered for the user.
To use the token interceptor your form should contain `<s:token>` tag .

9. servletConfig:- This interceptor checks whether the action class is implementing the different Aware interfaces. It the action class is implementing the Aware interfaces then this interceptor will do dependency injection of all the objects into the action class in the pre-processing phase.

`ServletContextAware` – sets the `ServletContext`
`ServletRequestAware` – sets the `HttpServletRequest`
`ServletResponseAware` – sets the `HttpServletResponse`
`ParameterAware` – sets a map of the request parameters
`RequestAware` – sets a map of the request attributes
`SessionAware` – sets a map of the session attributes
`ApplicationAware` – sets a map of application scope properties
`PrincipalAware` – sets the `Principal` object (security)

10. i18n:- This interceptor sets the locale for the current request and helps to support doing internationalization.

11. autowiring:- This interceptor provides integration with Spring.

12. createSession:- This interceptor creates a new HttpSession object.

13. fileUpload:- It is used for uploading a file.

14. staticParams: interceptor also moves parameters onto properties exposed on the ValueStack for a particular action. The difference is the origin of the parameters. The parameters that this interceptor moves are defined in the `<action>` tag.

```
<action name="exampleAction" class="example.ExampleAction">
   <param name="firstName">John</param>
   <param name="lastName">Doe</param>
</action>
```

15. timer:- This interceptor logs the time taken for an execution. If you place this interceptor at the heart of your stack, just before the action, then it will time the action's execution itself. If you place it at the outer most layer of the stack, it will be timing the execution of the entire stack, as well as the action.

16. execAndWait:- While running a long action users may get impatient in case of long delayed response. To avoid this execAndWait interceptor is used. This runs the long running action in the background and displays the page with progress block to the user. It also prevents the HttpRequest time outs. To use this we have to specify two parameters
    1. delay
    2. delayInterval

17. alias:- In case of action chaining if you want to copy the one action variables data into another action with different name we have to use this interceptor. When you are using this interceptor you must explicitly configure this under the required action with following parameters

```
<action>
     <param name="aliases">
            #{‘username’ : ‘userId’ , ‘email’ : ’emailId’}
      </param>
      <interceptor-ref name="alias"/>
</action>
```

18. chain:- This interceptor is used to copy all objects in the value stack of currently executing Action class to the value stack of next Action class to be executed in the action chaining.
    Following is the way to configure the result to invoke another action.

```
<action>
      <result name="success" type="chain">
            doLogin
      </result>
</action>
```