1. **What is Struts2?**

Struts2 is popular and mature web application framework based on the MVC design pattern. Struts2 is not just the next version of Struts 1, but it is a complete rewrite of the Struts architecture.

1. **Name some of the features of Struts2.**

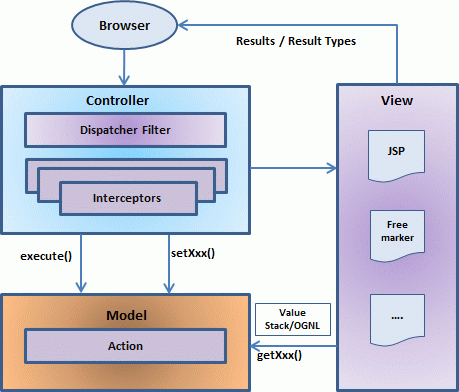
Here are some of the great features that may force you to consider Struts2 −

* **POJO forms and POJO actions** − Struts2 has done away with the Action Forms that were an integral part of the Struts framework. With Struts2, you can use any POJO to receive the form input. Similarly, you can now see any POJO as an Action class.
* **Tag support** − Struts2 has improved the form tags and the new tags allow the developers to write less code.
* **AJAX support** − Struts2 has recognised the take over by Web2.0 technologies, and has integrated AJAX support into the product by creating AJAX tags, that function very similar to the standard Struts2 tags.
* **Easy Integration** − Integration with other frameworks like Spring, Tiles and SiteMesh is now easier with a variety of integration available with Struts2.
* **Template Support** − Support for generating views using templates.
* **Plugin Support** − The core Struts2 behaviour can be enhanced and augmented by the use of plugins. A number of plugins are available for Struts2.

1. **What are the core components of a Struct2 based application?**

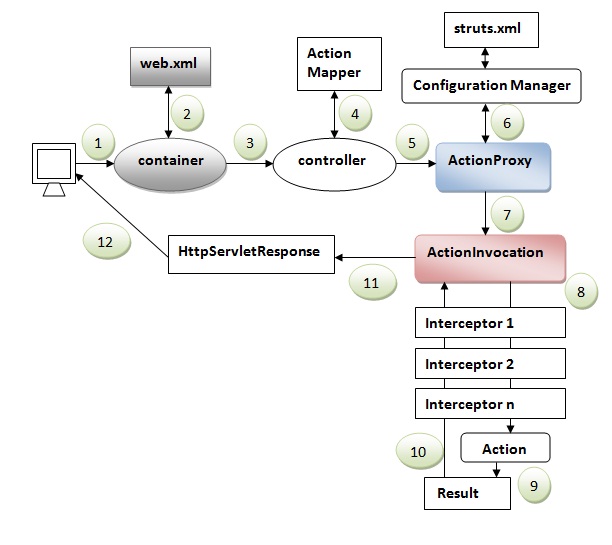
The Model-View-Controller pattern in Struts2 is realized with following five core components −

* Actions
* Interceptors
* Value Stack / OGNL
* Results / Result types
* View technologies



1. **Explain the life cycle of a request in Struct2 application.**

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

<web-app id="WebApp\_9" version="2.4"

    xmlns="<http://java.sun.com/xml/ns/j2ee>"

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

    xsi:schemaLocation="<http://java.sun.com/xml/ns/j2ee>

<http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd>">

<!-- servlet definition -->

<servlet>

<servlet-name>ErrorHandler</servlet-name>

<servlet-class>ErrorHandler</servlet-class>

</servlet>

<!-- servlet mappings -->

<servlet-mapping>

<servlet-name>ErrorHandler</servlet-name>

<url-pattern>/ErrorHandler</url-pattern>

</servlet-mapping>

    <filter>

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

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

</filter-class>

    </filter>

    <filter-mapping>

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

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

    </filter-mapping>

    <!-- ... -->

</web-app>

1. **What is the purpose of struts.xml in Struct2?**

The struts.xml file contains the configuration information that you will be modifying as actions are developed. This file can be used to override default settings for an application, for example struts.devMode = false and other settings which are defined in property file. This file can be created under the folder WEB-INF/classes.

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

<!DOCTYPE struts PUBLIC

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

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

<struts>

<constant name="struts.devMode" value="true" />

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

<action name="hello"

class="com.tutorialspoint.struts2.HelloWorldAction"

method="execute">

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

</action>

</package>

</struts>

Few words about the above configuration file.

**Constant Tag**

The constant tag along with name and value attributes will be used to override any of the following properties defined in default.properties, like we just set struts.devMode property.

Setting struts.devMode property allows us to see more debug messages in the log file.Here we set the constant **struts.devMode** to **true**, because we are working in development environment and we need to see some useful log messages.

**Package Tag**

Then, we defined a package called **helloworld**.

Creating a package is useful when you want to group your actions together.

**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:

/actionName.action

If you specify any namespace, you need this URI:

/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.

**Action Tag:**

In our example, we named our action as "hello" which is corresponding to the URL **/hello.action** and is backed up by the **HelloWorldAction.class**. The **execute** method of  **HelloWorldAction.class** is the method that is run when the URL **/hello.action** is invoked. If the outcome of the **execute** method returns "success", then we take the user to **HelloWorld.jsp**.

* **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:

<action name=" hello ">

* **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:

<action name="hello"

class="com.tutorialspoint.struts2.HelloWorldAction">

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

</action>

will be same as:

<action name="hello"

class="com.tutorialspoint.struts2.HelloWorldAction"

**method="execute">**

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

</action>

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

**Result element:**

It is the sub element of action that specifies where to forward the request for this action. Results determine what gets returned to the browser after an action is executed. The string returned from the action should be the name of a result. Results are configured per-action as above, or as a "global" result, available to every action in a package. Results have optional name and type attributes. The default name value is "success".

**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.

1. **What is the purpose of struts.properties in Struct2?**

This configuration file provides a mechanism to change the default behavior of the framework. Actually all of the properties contained within the struts.properties configuration file can also be configured in the web.xml using the init-param, as well using the constant tag in the struts.xml configuration file. But if you like to keep the things separate and more struts specific then you can create this file under the folder WEB-INF/classes. The values configured in this file will override the default values configured in default.properties which is contained in the struts2-core-x.y.z.jar distribution.

### When set to true, Struts will act much more friendly for developers

struts.devMode = true

### Enables reloading of internationalization files

struts.i18n.reload = true

### Enables reloading of XML configuration files

struts.configuration.xml.reload = true

### Sets the port that the server is run on

struts.url.http.port = 8080

1. **What are interceptors in Struts 2?**

Interceptors are conceptually the same as servlet filters or the JDKs Proxy class. Interceptors allow for crosscutting functionality to be implemented separately from the action as well as the framework. You can achieve the following using interceptors −

* Providing preprocessing logic before the action is called.
* Providing postprocessing logic after the action is called.
* Catching exceptions so that alternate processing can be performed.

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

<!DOCTYPE struts PUBLIC

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

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

<struts>

<constant name="struts.devMode" value="true" />

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

<action name="hello"

class="com.tutorialspoint.struts2.HelloWorldAction"

method="execute">

<interceptor-ref name="params"/>

<interceptor-ref name="timer" />

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

</action>

</package>

</struts>

1. **How can you create your custom interceptor in Struts 2?**

Creating a custom interceptor is easy; the interface that needs to be extended is the Interceptor interface.

public interface Interceptor extends Serializable{

void destroy();

void init();

String intercept(ActionInvocation invocation)

throws Exception;

}

As the names suggest, the init() method provides a way to initialize the interceptor, and the destroy() method provides a facility for interceptor cleanup. Unlike actions, interceptors are reused across requests and need to be thread-safe, especially the intercept() method.

The **ActionInvocation** object provides access to the runtime environment. It allows access to the action itself and methods to invoke the action and determine whether the action has already been invoked.

If you have no need for initialization or cleanup code, the **AbstractInterceptor** class can be extended. This provides a default no-operation implementation of the init() and destroy() methods.

Let us create following **MyInterceptor.java** in **Java Resources > src** folder:

package com.tutorialspoint.struts2;

import java.util.\*;

import com.opensymphony.xwork2.ActionInvocation;

import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

public class MyInterceptor extends AbstractInterceptor {

public String intercept(ActionInvocation invocation)throws Exception{

/\* let us do some pre-processing \*/

String output = "Pre-Processing";

System.out.println(output);

/\* let us call action or next interceptor \*/

String result = invocation.invoke();

/\* let us do some post-processing \*/

output = "Post-Processing";

System.out.println(output);

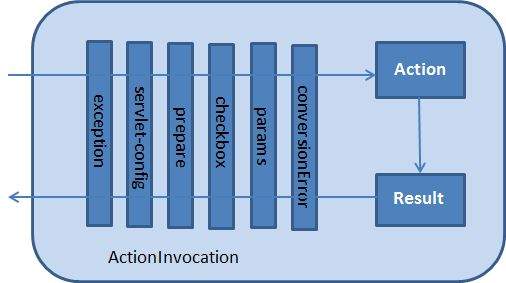
return result;

}

}

As you notice, actual action will be executed using the interceptor by**invocation.invoke()** call. So you can do some pre-processing and some post-processing based on your requirement.

The framework itself starts the process by making the first call to the ActionInvocation object's invoke(). Each time invoke() is called, ActionInvocation consults its state and executes whichever interceptor comes next. When all of the configured interceptors have been invoked, the invoke() method will cause the action itself to be executed. Following digram shows the same concept through a request flow:



1. **List few basic interceptors?**

Few of the important interceptors are listed below:

|  |  |
| --- | --- |
| **SN** | **Interceptor & Description** |
| 1 | **alias**  Allows parameters to have different name aliases across requests. |
| 2 | **checkbox**  Assists in managing check boxes by adding a parameter value of false for check boxes that are not checked. |
| 3 | **conversionError**  Places error information from converting strings to parameter types into the action's field errors. |
| 4 | **createSession**  Automatically creates an HTTP session if one does not already exist. |
| 5 | **debugging**  Provides several different debugging screens to the developer. |
| 6 | **execAndWait**  Sends the user to an intermediary waiting page while the action executes in the background. |
| 7 | **exception**  Maps exceptions that are thrown from an action to a result, allowing automatic exception handling via redirection. |
| 8 | **fileUpload**  Facilitates easy file uploading. |
| 9 | **i18n**  Keeps track of the selected locale during a user's session. |
| 10 | **logger**  Provides simple logging by outputting the name of the action being executed. |
| 11 | **params**  Sets the request parameters on the action. |
| 12 | **prepare**  This is typically used to do pre-processing work, such as setup database connections. |
| 13 | **profile**  Allows simple profiling information to be logged for actions. |
| 14 | **scope**  Stores and retrieves the action's state in the session or application scope. |
| 15 | **ServletConfig**  Provides the action with access to various servlet-based information. |
| 16 | **timer**  Provides simple profiling information in the form of how long the action takes to execute. |
| 17 | **token**  Checks the action for a valid token to prevent duplicate formsubmission. |
| 18 | **validation**  Provides validation support for actions |

1. **What is interceptor stack?**

As you can imagine, having to configure multiple interceptor for each action would quickly become extremely unmanageable. For this reason, interceptors are managed with interceptor stacks. Here is an example, directly from the struts-default.xml file:

<interceptor-stack name="basicStack">

<interceptor-ref name="exception"/>

<interceptor-ref name="servlet-config"/>

<interceptor-ref name="prepare"/>

<interceptor-ref name="checkbox"/>

<interceptor-ref name="params"/>

<interceptor-ref name="conversionError"/>

</interceptor-stack>

The above stake is called basicStack and can be used in your configuration as shown below. This configuration node is placed under the <package .../> node. Each <interceptor-ref .../> tag references either an interceptor or an interceptor stack that has been configured before the current interceptor stack. It is therefore very important to ensure that the name is unique across all interceptor and interceptor stack configurations when configuring the initial interceptors and interceptor stacks.

We have already seen how to apply interceptor to the action, applying interceptor stacks is no different. In fact, we use exactly the same tag:

<action name="hello" class="com.tutorialspoint.struts2.MyAction">

<interceptor-ref name="basicStack"/>

<result>view.jsp</result>

</action

The above registration of "basicStack" will register complete stake of all the six interceptors with hello action. This should be noted that interceptors are executed in the order, in which they have been configured. For example, in above case, exception will be executed first, second would be servlet-config and so on.

1. **What are Result types in Struts?**

When an action class method completes, it returns a String. The value of the String is used to select a result element. An action mapping will often have a set of results representing different possible outcomes. A standard set of result tokens are defined by the ActionSupport base class.

**Predefined result names**

String SUCCESS = "success";

String NONE = "none";

String ERROR = "error";

String INPUT = "input";

String LOGIN = "login";

The Action class manages the application's state, and the Result Type manages the view.

Often there is some navigation rules attached with the results. For example, if the action method is to authenticate a user, there are three possible outcomes. (a) Successful Login (b) Unsuccessful Login - Incorrect username or password (c) Account Locked.

In this scenario, the action method will be configured with the three possible outcome strings and three different views to render the outcome.

Struts2 does not tie you up with using JSP as the view technology. Afterall the whole purpose of the MVC paradigm is to keep the layers separate and highly configurable.

Struts comes with a number of predefined **result types** and whatever we've already seen that was the default result type **dispatcher**, which is used to dispatch to JSP pages.

The **dispatcher** result type is the default type, and is used if no other result type is specified. It's used to forward to a servlet, JSP, HTML page, and so on, on the server. It uses the *RequestDispatcher.forward()* method.

We saw the "shorthand" version in our earlier examples, where we provided a JSP path as the body of the result tag.

<result name="success">

/HelloWorld.jsp

</result>

We can also specify the JSP file using a <param name="location"> tag within the <result...> element as follows:

<result name="success" type="dispatcher">

<param name="location">

/HelloWorld.jsp

</param >

</result>

We can also supply a parse parameter, which is true by default. The parse parameter determines whether or not the location parameter will be parsed for OGNL expressions.

1. **What is the purpose of redirect result type?**

The redirect result type calls the standard response.sendRedirect() method, causing the browser to create a new request to the given location. We can provide the location either in the body of the <result...> element or as a <param name="location"> element.

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

<!DOCTYPE struts PUBLIC

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

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

<struts>

<constant name="struts.devMode" value="true" />

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

<action name="hello"

class="com.tutorialspoint.struts2.HelloWorldAction"

method="execute">

<result name="success" type="redirect">

<param name="location">

/NewWorld.jsp

</param >

</result>

</action>

<action name="index">

<result >/index.jsp</result>

</action>

</package>

</struts>

1. **What is Value Stack?**

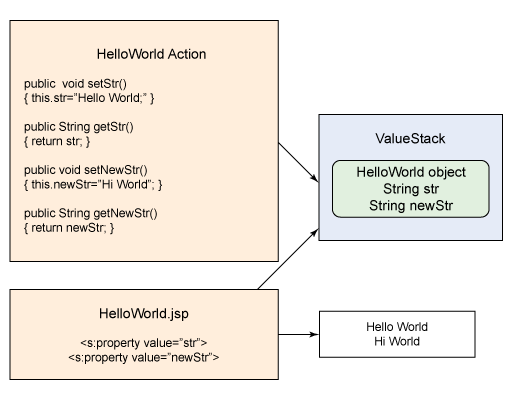
The ActionContext is a global storage area that holds all the data associated with the processing of a request.

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

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

ValueStack, one of the powerful features of Struts 2, holds all the data during processing of a request. Action classes also store data in the form of beans. (In Struts 2 you use getter and setter methods.) Once the data is stored in the action class objects, Struts 2 puts that data onto the ValueStack in the form of the Action object. ValueStack stores a stack of objects, and this data will be exposed to the other parts of the framework.

In Figure, the action class sets the values of two variables using getter and setter methods. This data will be saved to the ValueStack during processing of requests. HelloWorld.jsp will get the values of these variables from the ValueStack and display the values using the <s:property> tag from the Struts 2 tag library.

****

The value stack is a set of several objects which keeps the following objects in the provided order −

* **Temporary Objects** − There are various temporary objects which are created during execution of a page. For example the current iteration value for a collection being looped over in a JSP tag.
* **The Model Object** − If you are using model objects in your struts application, the current model object is placed before the action on the value stack.
* **The Action Object** − This will be the current action object which is being executed.
* **Named Objects** − These objects include #application, #session, #request, #attr and #parameters and refer to the corresponding servlet scopes.

<http://www.ibm.com/developerworks/library/wa-dynamicstruts/>

1. **What is OGNL?**

The Object-Graph Navigation Language (OGNL) is a powerful expression language that is used to reference and manipulate data on the ValueStack. OGNL also helps in data transfer and type conversion.

1. **Which components are available using ActionContext map?**

The ActionContext map consists of the following −

* **application** − application scoped variables.
* **session** − session scoped variables.
* **root / value stack** − all your action variables are stored here.
* **request** − request scoped variables.
* **parameters** − request parameters.
* **atributes** − the attributes stored in page, request, session and application scope.

1. **Which interceptor is responsible for file upload support?**

File uploading in Struts is possible through a pre-defined interceptor called FileUpload interceptor which is available through the org.apache.struts2.interceptor.FileUploadInterceptor class and included as part of the defaultStack.

1. **What are the Struts2 configuration properties that control file uploading process?**

Following are the Struts2 configuration properties that control file uploading process −

* **struts.multipart.maxSize** − The maximum size (in bytes) of a file to be accepted as a file upload. Default is 250M.
* **struts.multipart.parser** − The library used to upload the multipart form. By default is jakarta.
* **struts.multipart.saveDir** − The location to store the temporary file. By default is javax.servlet.context.tempdir.

1. **What are the Struts2 error message keys that can come during file uploading process?**

The fileUplaod interceptor uses several default error message keys −

* **struts.messages.error.uploading** − A general error that occurs when the file could not be uploaded.
* **struts.messages.error.file.too.large** − Occurs when the uploaded file is too large as specified by maximumSize.
* **struts.messages.error.content.type.not.allowed** − Occurs when the uploaded file does not match the expected content types specified.

1. **How to override the default error message that can come during file uploading process?**

You can override the text of these messages in WebContent/WEB-INF/classes/messages.properties resource files.

1. **What is Structs 2 validation framework?**

At Struts's core, we have the validation framework that assists the application to run the rules to perform validation before the action method is executed. Action class should extend the ActionSupport class, in order to get the validate method executed.

1. **How Struts 2 validation works?**

When the user presses the submit button, Struts 2 will automatically execute the validate method and if any of the if statements listed inside the method are true, Struts 2 will call its addFieldError method. If any errors have been added then Struts 2 will not proceed to call the execute method. Rather the Struts 2 framework will return input as the result of calling the action.

So when validation fails and Struts 2 returns input, the Struts 2 framework will redisplay the view file. Since we used Struts 2 form tags, Struts 2 will automatically add the error messages just above the form filed.

These error messages are the ones we specified in the addFieldError method call. The addFieldError method takes two arguments. The first is the form field name to which the error applies and the second is the error message to display above that form field.

1. **What is XML Based Validation in struts2?**

The second method of doing validation is by placing an xml file next to the action class. Struts2 XML based validation provides more options of validation like email validation, integer range validation, form validation field, expression validation, regex validation, required validation, requiredstring validation, stringlength validation and etc.

1. **What should be the name of xml file used for validation in struts?**

The xml file needs to be named '[action-class]'-validation.xml.

1. **What types of validations are available in xml based validation in struts2?**

Following is the list of various types of field level and non-field level validation available in Struts2 −

* date validator
* double validator
* email validator
* expression validator
* int validator
* regex validator
* required validator
* requiredstring validator
* stringlength validator
* url validator

1. **What is Internationalization?**

Internationalization (i18n) is the process of planning and implementing products and services so that they can easily be adapted to specific local languages and cultures, a process called localization. The internationalization process is sometimes called translation or localization enablement.

1. **How struts2 supports Internationalization?**

Struts2 provides localization ie. internationalization (i18n) support through resource bundles, interceptors and tag libraries in the following places −

* The UI Tags.
* Messages and Errors.
* Within action classes.

1. **What is the naming convention for a resource bundle file in struts2?**

The simplest naming format for a resource file is −

bundlename\_language\_country.properties

Here bundlename could be ActionClass, Interface, SuperClass, Model, Package, Global resource properties. Next part language\_country represents the country locale for example Spanish (Spain) locale is represented by es\_ES and English (United States) locale is represented by en\_US etc. Here you can skip country part which is optional.

1. **In which order Struts framework searches for a message bundle?**

When you reference a message element by its key, Struts framework searches for a corresponding message bundle in the following order −

* ActionClass.properties
* Interface.properties
* SuperClass.properties
* model.properties
* package.properties
* struts.properties
* global.properties

1. **Which class of struts is responsible to converts data types from string and vice versa?**

StrutsTypeConverter class tells Struts how to convert Environment to a String and vice versa by overriding two methods convertFromString() and convertToString().

1. **What inbuilt themes are provided by Struts2?**

Struts 2 comes with three built-in themes −

* **simple theme** − A minimal theme with no "bells and whistles". For example, the textfield tag renders the HTML <input/> tag without a label, validation, error reporting, or any other formatting or functionality.
* **xhtml theme** − This is the default theme used by Struts 2 and provides all the basics that the simple theme provides and adds several features like standard two-column table layout for the HTML, Labels for each of the HTML, Validation and error reporting etc.
* **css\_xhtml theme** − This theme provides all the basics that the simple theme provides and adds several features like standard two-column CSS-based layout, using <div> for the HTML Struts Tags, Labels for each of the HTML Struts Tags, placed according to the CSS stylesheet.

1. **How to handle exceptions in Structs?**

Struts makes the exception handling easy by the use of the "exception" interceptor. The "exception" interceptor is included as part of the default stack, so you don't have to do anything extra to configure it. It is available out-of-the-box ready for you to use.

1. **What is the purpose of @Results annotation?**

A @Results annotation is a collection of results. Under the @Results annotation, we can have multiple @Result annotations.

@Results({

@Result(name="success", value="/success.jsp"),

@Result(name="error", value="/error.jsp")

})

public class Employee extends ActionSupport{

...

}

1. **What is the purpose of @Result annotation?**

The @result annotations have the name that correspond to the outcome of the execute method. They also contain a location as to which view should be served corresponding to return value from execute().

@Result(name="success", value="/success.jsp")

public class Employee extends ActionSupport{

...

}

1. **What is the purpose of @Action annotation?**

This is used to decorate the execute() method. The Action method also takes in a value which is the URL on which the action is invoked.

public class Employee extends ActionSupport{

private String name;

private int age;

@Action(value="/empinfo")

public String execute()

{

return SUCCESS;

}

}

1. **What is the purpose of @After annotation?**

The @After annotation marks a action method that needs to be called after the main action method and the result was executed. Return value is ignored.

public class Employee extends ActionSupport{

@After

public void isValid() throws ValidationException {

// validate model object, throw exception if failed

}

public String execute() {

// perform secure action

return SUCCESS;

}

}

1. **What is the purpose of @Before annotation?**

The @Before annotation marks a action method that needs to be called before the main action method and the result was executed. Return value is ignored.

public class Employee extends ActionSupport{

@Before

public void isAuthorized() throws AuthenticationException {

// authorize request, throw exception if failed

}

public String execute() {

// perform secure action

return SUCCESS;

}

}

1. **What is the purpose of @BeforeResult annotation?**

The @BeforeResult annotation marks a action method that needs to be executed before the result. Return value is ignored.

public class Employee extends ActionSupport{

@BeforeResult

public void isValid() throws ValidationException {

// validate model object, throw exception if failed

}

public String execute() {

// perform action

return SUCCESS;

}

}

1. **What is the purpose of @ConversionErrorFieldValidator annotation?**

This validation annotation checks if there are any conversion errors for a field and applies them if they exist.

public class Employee extends ActionSupport{

@ConversionErrorFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true)

public String getName() {

return name;

}

}

1. **What is the purpose of @DateRangeFieldValidator annotation?**

This validation annotation checks that a date field has a value within a specified range.

public class Employee extends ActionSupport{

@DateRangeFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true,

min = "2005/01/01", max = "2005/12/31")

public String getDOB() {

return dob;

}

}

1. **What is the purpose of @DoubleRangeFieldValidator annotation?**

This validation annotation checks that a double field has a value within a specified range. If neither min nor max is set, nothing will be done.

public class Employee extends ActionSupport{

@DoubleRangeFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true,

minInclusive = "0.123", maxInclusive = "99.987")

public String getIncome() {

return income;

}

}

1. **What is the purpose of @EmailValidator annotation?**

This validation annotation checks that a field is a valid e-mail address if it contains a non-empty String.

public class Employee extends ActionSupport{

@EmailValidator(message = "Default message",

key = "i18n.key", shortCircuit = true)

public String getEmail() {

return email;

}

}

1. **What is the purpose of @ExpressionValidator annotation?**

This non-field level validator validates a supplied regular expression.

@ExpressionValidator(message = "Default message", key = "i18n.key",

shortCircuit = true, expression = "an OGNL expression" )

1. **What is the purpose of @IntRangeFieldValidator annotation?**

This validation annotation checks that a numeric field has a value within a specified range. If neither min nor max is set, nothing will be done.

public class Employee extends ActionSupport{

@IntRangeFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true,

min = "0", max = "42")

public String getAge() {

return age;

}

}

1. **What is the purpose of @RegexFieldValidator annotation?**

This annotation validates a string field using a regular expression.

@RegexFieldValidator( key = "regex.field", expression = "yourregexp")

1. **What is the purpose of @RequiredFieldValidator annotation?**

This validation annotation checks that a field is non-null. The annotation must be applied at method level.

public class Employee extends ActionSupport{

@RequiredFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true)

public String getAge() {

return age;

}

}

1. **What is the purpose of @RequiredStringValidator annotation?**

This validation annotation checks that a String field is not empty (i.e. non-null with a length > 0).

public class Employee extends ActionSupport{

@RequiredStringValidator(message = "Default message",

key = "i18n.key", shortCircuit = true, trim = true)

public String getName() {

return name;

}

}

1. **What is the purpose of @StringLengthFieldValidator annotation?**

This validator checks that a String field is of the right length. It assumes that the field is a String. If neither minLength nor maxLength is set, nothing will be done.

public class Employee extends ActionSupport{

@StringLengthFieldValidator(message = "Default message",

key = "i18n.key", shortCircuit = true,

trim = true, minLength = "5", maxLength = "12")

public String getName() {

return name;

}

}

1. **What is the purpose of @UrlValidator annotation?**

This validator checks that a field is a valid URL.

public class Employee extends ActionSupport{

@UrlValidator(message = "Default message",

key = "i18n.key", shortCircuit = true)

public String getURL() {

return url;

}

}

1. **What is the purpose of @Validations annotation?**

If you want to use several annotations of the same type, these annotation must be nested within the @Validations() annotation.

public class Employee extends ActionSupport{

@Validations(

requiredFields =

{@RequiredFieldValidator(type = ValidatorType.SIMPLE,

fieldName = "customfield",

message = "You must enter a value for field.")},

requiredStrings =

{@RequiredStringValidator(type = ValidatorType.SIMPLE,

fieldName = "stringisrequired",

message = "You must enter a value for string.")}

)

public String getName() {

return name;

}

}

1. **What is the purpose of @CustomValidator annotation?**

This annotation can be used for custom validators. Use the ValidationParameter annotation to supply additional params.

@CustomValidator(type ="customValidatorName", fieldName = "myField")

1. **What is the purpose of @Conversion Annotation annotation?**

This is a marker annotation for type conversions at Type level. The Conversion annotation must be applied at Type level.

@Conversion()

public class ConversionAction implements Action {

}

1. **What is the purpose of @CreateIfNull Annotation annotation?**

This annotation sets the CreateIfNull for type conversion. The CreateIfNull annotation must be applied at field or method level.

@CreateIfNull( value = true )

private List<User> users;

1. **What is the purpose of @Element Annotation annotation?**

This annotation sets the Element for type conversion. The Element annotation must be applied at field or method level.

@Element( value = com.acme.User )

private List<User> userList;

1. **What is the purpose of @Key Annotation annotation?**

This annotation sets the Key for type conversion. The Key annotation must be applied at field or method level.

@Key( value = java.lang.Long.class )

private Map<Long, User> userMap;

1. **What is the purpose of @KeyProperty Annotation annotation?**

This annotation sets the KeyProperty for type conversion. The KeyProperty annotation must be applied at field or method level.

@KeyProperty( value = "userName" )

protected List<User> users = null;

1. **What is the purpose of @TypeConversion Annotation annotation?**

This annotation annotation is used for class and application wide conversion rules. The TypeConversion annotation can be applied at property and method level.

@TypeConversion(rule = ConversionRule.COLLECTION,

converter = "java.util.String")

public void setUsers( List users ) {

this.users = users;

}