**Custom Actions:**

Where standard actions which could be defined by the Jsp Technology as part of the JSP API.

* To perform Jsp Actions we have already standard actions but these are in limited manner, in bounded functionality that’s why with these standard actions we are unable to achieve the complete client requirement. Even though if you use standard actions in our Jsp Applications still there is a requirement to write java code inside the jsp pages.
* To overcome the above drawbacks we should go for custom actions.
* Custom actions are the actions which could be defined by the developers as per their application requirements.
* Custom actions could be provided by the developers by defining a set of tags called as Custom Actions.

***Tag Syntax:***

***<prefixname:tagname [Attribute-List] >***

***//Body***

***</prefixname:tagname>***

To design Custom Tags we should require the fallowing components.

1. ***A Jsp Page with Taglib directive***
2. ***A TLD File***
3. ***Tag Handler Class***

Where Taglib directive can be used to specify the location where we have respective TLD File and define prefix names of the Custom tags.

***<%@taglib uri=” “ prefix=” “ />***

Where TLD File is an XML file, which can be used to provide the mapping between custom tag names and respective tag handler classes and provide some description about Custom tags attributes.

To provide the mapping between custom tag and the respective tag handler classes we should use the fallowing XML Tags in respective TLD Files.

<taglib>

<tlib-version>------ </tlib-version>

<jsp-version>-------</jsp-version>

<short-name>------</short-name>

<description>-------</description>

<tag>

<name>-------</name>

<tag-class>-----</tag-class>

<body-content>-----</body-content>

<short-name>------</short-name>

<description>------</description>

</tag>

---

----

</taglib>

* Where <tlib-version> can be used to specify user the user defined TLD File version number.
* Where <jsp-version> can be used to specify the Jsp Version Number. Note: jsp-version number should have the value either jsp1.1 or above version.
* Where <short-name> can be used to specify a short name for the TLD File.
* Where <description> can be used to specify the generalized description about the present TLD File.
* Where <name> can be used to specify name of the Custom Tag.
* Where <tag-class> can be used to specify the fully qualified name of the Tag Handler class.
* Where <body-content> can be used to specify which type of body we may provide for the custom tag like empty, jsp—etc.
* Where <short-name> can be used to specify a short name for the Custom Tag.
* Where <description> can be used to specify the generalized description of the present custom tag.
* Here <short-name> and <description> tags are optional.

***Tag Handler Classes:***

These are the normal java classes, which will provide the basic functionality of the Custom tags.

When container encounters a custom tag then container will pick up the **custom tag name and its prefix name**, now container will identify respective Taglib directive, when container is trying to match with **prefix attribute value.**

If container gets match then container will pick up the **uri** attribute value, by this container **will get the location where we have the respective TLD File.**

From the TLD File container will get the **mapping between custom tag name and respective tag handler class** by this container will identify location where we have respective Tag Handler Class.

When container identifies the respective tag handler class **.class** File then **container will load its byte code to the memory** i.e. THC Loading container will **get an object for the loaded THC called as THC instantiation** and container will execute THC life cycle methods.

To define Custom Tags Tag Handler Class JSP API has provided the fallowing predefined library as part of “***javax.servlet.jsp.tagext***” package.

***JspTag (i)***

***extends extends***

***Tag (i) SimpleTag (i)***

***extends***

***implements implements***

***IteratorTag (i) Tagsupport (c) SimpleTagSupport (c)***

***extends***

***implements***

***BodyTag (i) BodyTagSupport (c)***

***Simple Classic Tag:***

It is a Custom tag without body and without attributes. To design Simple Custom Tag the respective Tag Handler Class must implement **Tag interface** directive.

***public interface Tag extends JspTag***

***{***

***public static final int EVAL\_BODY\_INCLUDE;***

***public static final int SKIP\_BODY;***

***public static final int EVAL\_PAGE;***

***public static final int SKIP\_PAGE;***

***public void setPageContext(PageContext pagecontext);***

***public void setParent(Tag t);***

***public Tag getParent();***

***public int doStartTag() throws JspException;***

***public int doEndTag() throws JspException;***

***public void release();***

***}***

***Public class MyHandler implements Tag***

***{***

***//Body***

***}***

* When container encounter a custom tag then container will execute setPageContext(), by this container will provide PageContext object to the throughout the Tag Handler Class.
* Where setParent() is a method, which could be executed by the container after the setPageContext(), which can be used to specify the parent tags Tag Handler Class Object, if the present tag is a **child** tag.
* Where getParent() is a method which can be used to return Parent tags Tag Handler Class object reference. This method could be executed by the container when the present tag is a child tag to the parent tag, otherwise container will not execute.

Where doStartTag() is a method which could be executed by the container when container encounters start tag of the custom tag.

There are two possible return values from doStartTag()

1. EVAL\_BODY\_INCLUDE
2. SKIP\_BODY

If doStartTag() returns EVAL\_BODY\_INCLUDE constant then container will evaluate the body of the custom tag.

If doStartTag() returns SKIP\_BODY as a constant then container will skip out the custom tag body and bypass flow of control to the doEndTag() of the custom tag.

Where doEndTag() is a method, which could be executed by the container when it encounter end tag of the custom tag. There are two possible return values from doEndTag().

1. EVAL\_PAGE
2. SKIP\_PAGE

If doEndTag() returns EVAL\_PAGE constant then container will evaluate the remaining jsp page.

If doEndTag() returns SKIP\_PAGE as a constant then container should not evaluate the remaining jsp page.

* Where release() is a method, which could be executed by the container to destroy tag handler class object after the tag handling. This method is same as destroy() in servlet interface.

***setPageContext(PageContext pc)***

***setParent(Tag t)***

***doStartTag()***

***doEndTag()***

***Evaluate Body***

***Evaluate Page***

***release()***

***EVAL\_BODY\_INCLUDE***

***SKIP\_BODY***

***EVAL\_PAGE***

***SKIP\_PAGE***

***myjsp.jsp:***

<%@taglib uri="/WEB-INF/mytld.tld" prefix="mytags" %>

<h1>

<mytags:hello/> ***( jsp file in application folder)***

</h1>

***mytld.tld:***

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version>jsp 1.2</jsp-version>

<tag>  ***( mytld.tld file in WEB-INF Folder )***

<name>hello</name>

<tag-class>com.dss.mytags.MyHandler</tag-class>

<body-content>empty</body-content>

</tag>

</taglib>

***MyHandler.java:***

package com.dss.mytags;

import javax.servlet.jsp.tagext.\*;

import javax.servlet.jsp.\*;

import java.io.\*; ***( THC is in classes folder )***

public class MyHandler implements Tag

{ PageContext pagecontext;

public void setPageContext(PageContext pagecontext)

{ this.pagecontext=pagecontext; }

public void setParent(Tag t)

{ }

public Tag getParent()

{ return null; }

public int doStartTag() throws JspException

{ try

{pagecontext.getOut().println("<h1> Hello </h1>");

}

catch(Exception e)

{ System.out.println(e); }

return SKIP\_BODY;

}

public int doEndTag() throws JspException

{ return SKIP\_PAGE; }

public void release()

{ }

}

In the above example if we specify empty is a value to <body-content> in the tld file but if you provide any body for the custom tag then container will raise an exception like

org.apache.jasper.JasperException:myjsp.jsp (5,0) according to TLD, tag mytags:hello must be empty, but it is not.

***Attributes to Custom Tags:***

If you want to provide attributes for the custom tags then we should use the fallowing Three Actions.

1. Define an attribute and provide a value to that attribute in the jsp page.

Eg: <mytags:hello name=”abc” />

1. Provide description about the custom tag attribute in the respective tld file.

Eg:

<taglib>

--

----

<tag>

--

--

<attribute>

<name>name</name>

<required>true</requires>

<rtexprvalue>true</rtexprvalue>

</attribute>

--

--

</tag>

--

--

</taglib>

* Where name attribute can be used to specify attribute name.
* Where required tag is a Boolean tag, which can be used to specify whether the attribute in the custom tag is mandatory or optional.
* Where <rtexprvalue> is a Boolean tag, which can be used to specify whether the attribute is accepting runtime evaluated expression value or not.

1. Define the property and the respective setter method with the same name of the attribute name defined in the custom tag.

Eg:

-----import-----

Public class MyHandler implements Tag

{

--

--

--

Public String name;

Public void setName(String name)

{

this.name=name;

}

--

--

--

}

When container encounter a custom tag with an attribute then container will pick up the value of the attribute and execute the respective setter method by passing attribute value as a parameter after the setParent() as part of the Tag Handler Class life cycle.

***Iteration tags:***

Iteration tags are the custom tags, which will provide number of iterations on their body.

To design iteration tags respective tag handler class must implement IteratorTag interface either directly or indirectly.

***public interface IteratorTag extends Tag***

***{***

***public static final int EVAL\_BODY\_INCLUDE;***

***public static final int SKIP\_BODY;***

***public static final int EVAL\_PAGE;***

***public static final int SKIP\_PAGE;***

***public static final int EVAL\_BODY\_AGAIN;***

***public void setPageContext(PageContext pagecontext);***

***public void setParent(Tag t);***

***public Tag getParent();***

***public int doStartTag() throws JspException;***

***public int doAfterBody() throws JspException;***

***public int doEndTag() throws JspException;***

***public void release();***

***}***

***Public class MyHandler implements IteratorTag***

***{***

***//Body***

***}***

* ***If*** you want to provide iterations on the custom tag body then always we should return EVAL\_BODY\_INCLUDE as a constant from doStartTag().
* *When we* return EVAL\_BODY\_INCLUDE constant from doStartTag() then container will evaluate custom tag body one time, at the end of the body evaluation container will execute doAfterBody().
* There are two possible return values from doAfterBody()

1. ***EVAL\_BODY\_AGAIN***
2. ***SKIP\_BODY***

* If doAfterBody() returns SKIP\_BODY as a constant then container should skip out the body evaluation and bypass flow of execution to the custom tags end tag.
* If doAfterBody() returns EVAL\_BODY\_AGAIN constant then container will evaluate body once again and call doAfterBody().
* In the iteration tags the above step will be repeated until doAfterBody() returns SKIP\_BODY constant.

***setPageContext(PageContext pc)***

***setParent(Tag t)***

***doStartTag()***

***doEndTag()***

***Evaluate Body***

***Evaluate Page***

***release()***

***All the setter methods***

***doAfterBody()***

***EVAL\_BODY\_INCLUDE***

***SKIP\_BODY*** ***SKIP\_BODY***

***EVAL\_BODY\_AGAIN***

***EVAL\_PAGE***

***SKIP\_PAGE***

***TagSupport Class:***

In the case of simple classic tags and iteration tags to design a Tag Handler Class we should take one user defined class and which must implement either Tag Interface or IteratorTag Interface. In this case we should provide the implementation for all the methods in the respective interface with or without the requirement.

To overcome this drawback we should go for an alternative, which was provided by JSP API in the form of the TagSupport Class.

* TagSupport is a class, which was implemented directly IteratorTag interface and provide the default implementation for all the methods got declared in the IteratorTag interface.
* To design a Tag Handler class with TagSupport Class we should use one user defined class, which is extending from TagSupport Class and over ride the respective methods.

***public class TagSupport implements IteratorTag***

***{***

***public static final int EVAL\_BODY\_INCLUDE;***

***public static final int SKIP\_BODY;***

***public static final int EVAL\_PAGE;***

***public static final int SKIP\_PAGE;***

***public static final int EVAL\_BODY\_AGAIN;***

***public PageContext pagecontext;***

***public Tag t;***

***public void setPageContext(PageContext pagecontext)***

***{***

***this.pagecontext=pagecontext;***

***}***

***public void setParent(Tag t)***

***{ this.t=t;***

***}***

***public Tag getParent()***

***{ return t;***

***}***

***public int doStartTag() throws JspException***

***{ return SKIP\_BODY;***

***}***

***public int doAfterBody() throws JspException***

***{ return SKIP\_BODY;***

***}***

***public int doEndTag() throws JspException***

***{ return EVAL\_PAGE;***

***}***

***public void release()***

***{ pagecontext=null;***

***t=null;***

***}***

***}***

***Public class MyHandler extends TagSupport***

***{***

***//over ride the required method***

***}***

***myjsp.jsp:***

<%@taglib uri="/WEB-INF/mytld.tld" prefix="mytags" %>

<h1>

<mytags:iterate times="10">

Challa <br>

</mytags:iterate>

</h1>

***mytld.tld:***

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version> 1.2</jsp-version>

<tag>

<name>iterate</name>

<tag-class>com.dss.mytags.MyHandler</tag-class>

<body-content>jsp</body-content>

<attribute>

<name>times</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

</taglib>

***MyHandler.java:***

package com.dss.mytags;

import javax.servlet.jsp.tagext.\*;

import javax.servlet.jsp.\*;

import java.io.\*;

public class MyHandler extends TagSupport

{ public int times;

int count=1;

public void setTimes(int times)

{ this.times=times;

}

public int doStartTag() throws JspException

{ return EVAL\_BODY\_INCLUDE; }

public int doAfterBody() throws JspException

{ if(count<=times)

{ count++;

return EVAL\_BODY\_AGAIN; }

else

{ return SKIP\_BODY; }

}

}

***Body Tags:***

As part of the Jsp Actions, in custom actions so far we have simple classic tags and iteration tags, these are capable of getting the custom tags body and display that on the client browser as it is without having any modifications.

As per the application requirements we need to modify the custom tags body explicitly at the respective tag handler classes.

To achieve the above simple classic tags and iteration tags are not sufficient so that we should go for body tags custom tag.

To design body tags the respective tag handler class must implement BodyTag Interface either directly or indirectly.

***public interface BodyTag extends IteratorTag***

***{ public static final int EVAL\_BODY\_INCLUDE;***

***public static final int SKIP\_BODY;***

***public static final int EVAL\_PAGE;***

***public static final int SKIP\_PAGE;***

***public static final int EVAL\_BODY\_AGAIN;***

***public static final int EVAL\_BODY\_BUFFERED;***

***public void setPageContext(PageContext pagecontext);***

***public void setParent(Tag t);***

***public Tag getParent();***

***public int doStartTag() throws JspException;***

***public void setBodyContent(BodyContent bodycontent);***

***public void doInitBody() throws JspException;***

***public int doAfterBody() throws JspException;***

***public int doEndTag() throws JspException;***

***public void release();***

***}***

***Public class MyHandler implements BodyTag***

***{***

***//Body***

***}***

In general in the simple classic tags and in the iteration tags there are two possible return values from doStartTag() at the respective tag handler classes. But in case of the BodyTag there are three possible return values from doStartTag().

1. ***EVAL\_BODY\_INCLUDE***
2. ***SKIP\_BODY***
3. ***EVAL\_BODY\_BUFFERED***

When doStartTag() returns EVAL\_BODY\_BUFFERED as a constant then container will pick up the custom tags body and stored on to a buffer then container should access setBodyContent().

To call setBodyContent() container must create body content object, which will maintain the buffer having custom tags body.

After the setBodyContent() execution container should access doInitBody() for preparing body content object for allowing manipulations on the custom tag body.

***BodyTagSupport Class:***

To design BodyTag custom tags if you use the above approach i.e. implementing BodyTag interface directly then we should provide the implementation for all the methods got declared in BodyTag interface with or without the requirement.

To overcome the above drawback we should go for an alternative, here the alternative was provided by sun micro systems in the form of BodyTagSupport class as part of ***javax.servlet.jsp.tagext.\**** package.

BodyTagSupport class is a concrete class which was implemented directly BodyTag interface and provide default implementation for all the methods got declared in BodyTag interface.

To design Tag Handler classes in body tags with BodyTagSupport class then we should take one user defined class, which is extending from BodyTagSupport class and over ride the required method.

***public class BodyTagSupport extends TagSupport implements BodyTag***

***{ public static final int EVAL\_BODY\_INCLUDE;***

***public static final int SKIP\_BODY;***

***public static final int EVAL\_PAGE;***

***public static final int SKIP\_PAGE;***

***public static final int EVAL\_BODY\_AGAIN;***

***public static final int EVAL\_BODY\_BUFFERED;***

***public PageContext pagecontext;***

***public Tag t;***

***public BodyContent bodycontent;***

***public void setPageContext(PageContext pagecontext)***

***{***

***this.pagecontext=pagecontext;***

***}***

***public void setParent(Tag t)***

***{***

***this.t=t;***

***}***

***public Tag getParent()***

***{***

***return t; }***

***public int doStartTag() throws JspException***

***{ return EVAL\_BODY\_BUFFERED;***

***}***

***public void setBodyContent(BodyContent bodycontent)***

***{ this.bodycontent=bodycontent;***

***}***

***public void doInitBody() throws JspException***

***{ }***

***public int doAfterBody() throws JspException***

***{ return SKIP\_BODY;***

***}***

***public int doEndTag() throws JspException***

***{***

***return SKIP\_PAGE;***

***}***

***public void release()***

***{ pagecontext=null;***

***bodycontext=null;***

***}***

***}***

***Public class MyHandler extends BodyTagSupport***

***{***

***//Body***

***}***

***setPageContext(PageContext pc)***

***setParent(Tag t)***

***doStartTag()***

***Evaluate Body***

***doAfterBody()***

***setBodyContent(BodyContent bc)***

***doInitBody()***

***doAfterBody()***

***doEndTag()***

***Evaluate Page***

***release()***

***EVAL\_BODY\_INCLUDE EVAL\_BODY\_BUFFERED***

***SKIP\_BODY***

***SKIP\_BODY***

***SKIP\_BODY***

***EVAL\_BODY\_AGAIN***

***EVAL\_PAGE***

***SKIP\_PAGE***

***reverse.jsp:***

<%@taglib uri="/WEB-INF/reverse.tld" prefix="mytags" %>

<h1><mytags:reverse>

Challa Software Solutions

</mytags:reverse></h1>

***reverse.tld:***

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version>jsp 1.2</jsp-version>

<tag>

<name>reverse</name>

<tag-class>com.dss.mytags.Reverse</tag-class>

<body-content>jsp</body-content>

</tag>

</taglib>

***Reverse.java:***

package com.dss.mytags;

import javax.servlet.jsp.tagext.\*;

import javax.servlet.jsp.\*;

import java.io.\*;

public class Reverse extends BodyTagSupport

{ public int doEndTag() throws JspException

{ try { String data=bodyContent.getString();

StringBuffer sb=new StringBuffer(data);

bodyContent.getEnclosingWriter().println(sb.reverse());

}

catch(Exception e)

{ System.out.println(e); }

return EVAL\_PAGE;

}

}

* Where ***getString()*** is a method which can be used to return custom tag body as a string from ***bodycontent Object***.
* Where ***getEnclosingWriter()*** is a method from ***bodycontent Object*** which can be used to return a writer object to carry response from ***bodycontent Object to the response object.***

***Simple Tags:***

***Q) What are the differences between classic tags and simple tags?***

1. Classic Tags is a concept along with the jsp 1.1 version where as Simple Tags is a concept which came along with jsp 2.0 version.
2. In case of the Classic Tags to design any custom tag we should remember many more number of life cycle’s like Tag, IteratorTag, BodyTag ..etc. In case of Simple Tags we should remember only one life cycle to design any custom tag.
3. Classic tags will not provide easy of design where as Simple Tags will provide easy of design.
4. Classic Tag’s Tag Handler Class Objects are cached i.e. when container encounter a Classic Tag first time then container will create the respective THC Object and keep aside for the sake of the feature reference (cache). i.e. when container encounter the same classic tag then container should not create a new THC Object, simply it will reuse the existed THC Object. Simple Tags THC Objects are never cached. i.e. when container encounter the same Simple Tag every time container should create a separate THC Object.
5. In case of the Classic Tags we should use doStartTag(), doEndTag() for every minimum custom tag. But in case of Simple Tag we should use only doTag() for every minimum Simple Tag.
6. Simple Tags will have all most all the power of Body Tags in classic tags.
7. In case of the classic tags to design any custom tag the respective THC must implement Tag Interface either directly or indirectly. In case of Simple Tags to design any Custom Tag the respective THC must implement SimpleTag interface either directly or indirectly.

***SimpleTag Custom Tag:***

To design SimpleTag Custom Tags THC we should take one user defined class, which is implementing SimpleTag interface.

***public interface SimpleTag extends JspTag***

***{***

***public void setJspContext(JspContext jspContext);***

***public void setParent(JspTag jt);***

***public JspTag getParent();***

***public void setJspBody(JspFragment jf);***

***public void doTag();***

***}***

***public class MyHandler implements SimpleTag***

***{***

***--***

***--***

***}***

* To design SimpleTags if you use the above approach then we should provide the implementation for all the number of methods declared in the SimpleTag interface with or without the requirement.
* To overcome the above drawback we should go for an alternative here the alternative was provided by the sun micro systems in the form of SimpleTagSupport concrete class in JSP API as part of ***“javax.servlet.jsp.tagext”*** package.
* To design SimpleTag with SimpleTagSupport class then we should take one user defined class which is extending from SimpleTagSupport class where we should over ride the required method.

***setJspContext(JspContext jc)***

***setParent(JspTag jt)***

***All the setter methods***

***setJspBody(JspFragment jf)***

***doTag()***

* Note: In case of the Simple Tags the setParent() & getParent() will be executed by the container as part of the SimpleTag life cycle, whenever the present tag is a child tag for any parent tag.
* ***Where setJspBody() will be executed by the container when it identify a body for the custom tag otherwise container should not execute.***

***public class SimpleTagSupport implements SimpleTag***

***{ public JspTag jt;***

***public JspFragment jf;***

***public JspContext jspContext;***

***public void setJspContext(JspContext jspContext)***

***{***

***this.jspContext=jspContext;***

***}***

***public void setParent(JspTag jt)***

***{***

***this.jt=jt;***

***}***

***public JspTag getParent()***

***{***

***Return jt;***

***}***

***public void setJspBody(JspFragment jf)***

***{***

***this.jf=jf;***

***}***

***public JspContext getJspContext()***

***{***

***return JspContext;***

***}***

***public JspFragment getJspBody()***

***{***

***return jf;***

***}***

***public void doTag()***

***{***

***}***

***protected JspTag findAncestor withclass(JspTag jt, Class class)***

***{***

***return jt;***

***}***

***}***

***public class MyHandler extends SimpleTagSupport***

***{***

***--***

***--***

***}***

***hello.jsp:***

<%@taglib uri="/WEB-INF/abc.tld" prefix="mytags"%>

<h1>

<mytags:hello/>

</h1>

***abc.tld:***

<taglib>

<tlib-version>1.2</tlib-version>

<jsp-version>jsp 2.0</jsp-version>

<tag>

<name>hello</name>

<tag-class>com.dss.mytags.Hello</tag-class>

<body-content>empty</body-content>

</tag>

</taglib>

***Hello.java:***

**package com.dss.mytags;**

**import javax.servlet.jsp.\*;**

**import javax.servlet.jsp.tagext.\*;**

**import java.io.\*;**

**public class Hello extends SimpleTagSupport**

**{ JspContext c;**

**public void setJspContext(JspContext c)**

**{**

**this.c=c;**

**}**

**public void doTag() throws JspException**

**{**

**try**

**{**

**c.getOut().println("hello");**

**}**

**catch(Exception e)**

**{**

**System.out.println(e);**

**}**

**}**

**}**