Java Server Pages (JSP)

By
Apparao G

Java Server Pages (JSP) is a technology for developing Webpages that supports dynamic content.

A Java Server Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

Advantages of JSP over Servlet:

1) Extension to Servlet:

JSP technology is the extension to Servlet technology. We can use all the features of the Servlet in JSP.

2) Easy to maintain:

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

3) Fast Development: No need to recompile and redeploy

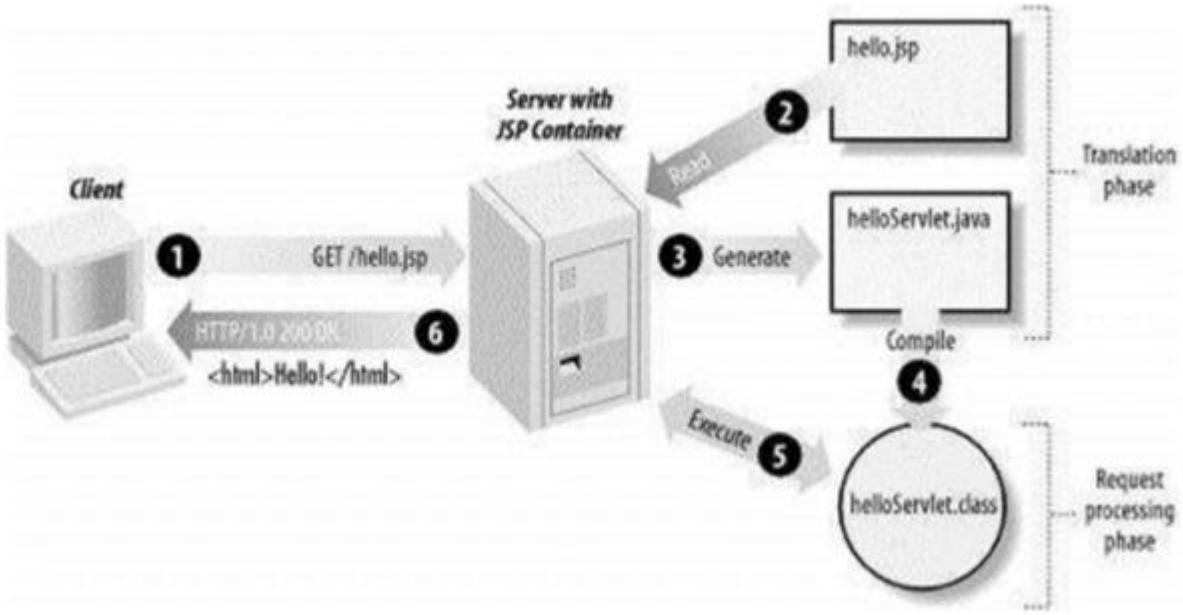
If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

4) Less code than Servlet:

In JSP, we can use many tags such as action tags, JSTL, custom tags, etc. that reduces the code. Moreover, we can use EL, implicit objects, etc.

JSP Processing:

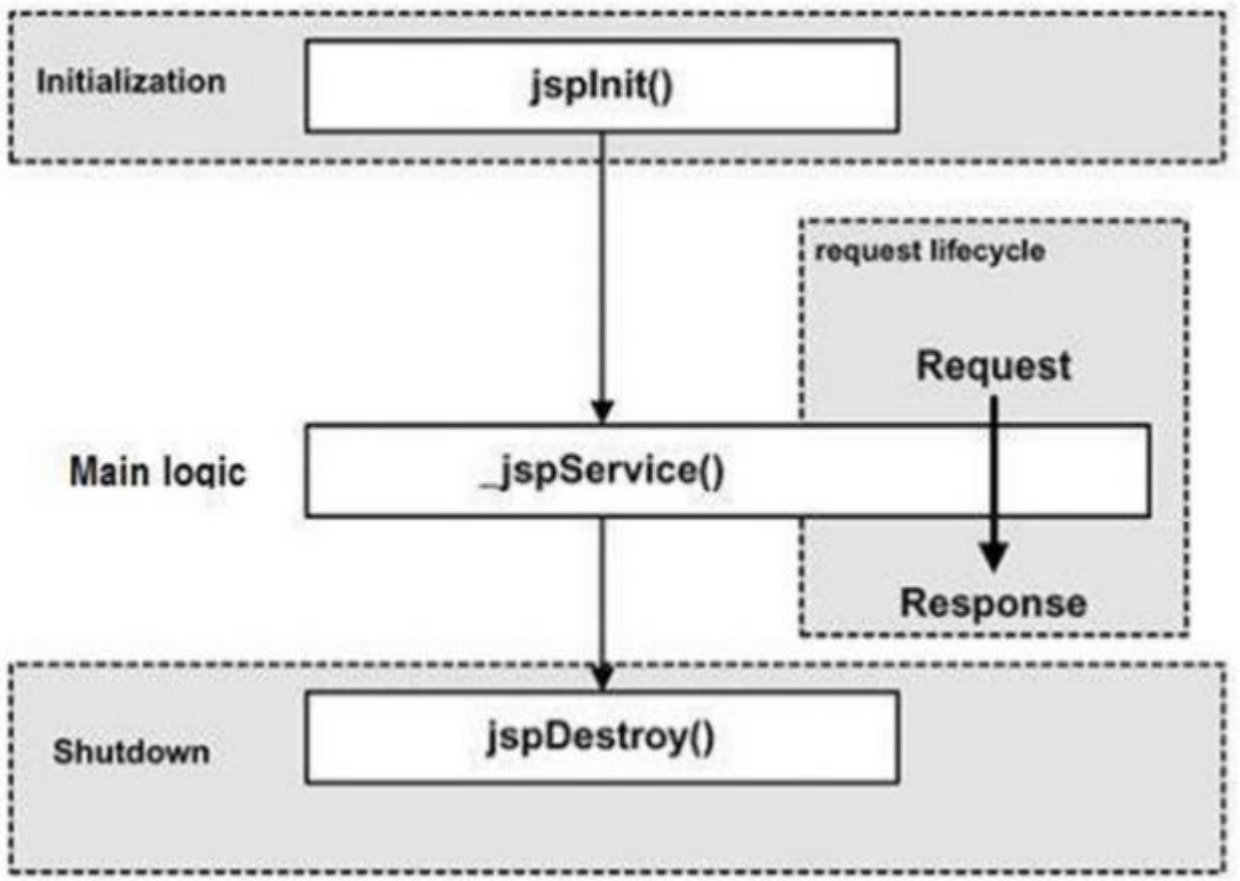
The web server needs a JSP engine, i.e., a container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages.



Life cycle of JSP page:

- Pre-translated: Before the JSP file has been translated and compiled into the Servlet.
- Translated: The JSP file has been translated and compiled as a Servlet.
- Initialized: Prior to handling the requests in the service method the container calls the jspInit() to initialize the Servlet. Called only once per Servlet instance.
- Servicing: Services the client requests. Container calls the _jspService() method for each request.
- Out of service: The Servlet instance is out of service. The container calls the jspDestroy() method.

The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle. The four phases have been described below:



JSP Compilation:

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps:

Parsing the JSP.

Turning the JSP into a servlet.

Compiling the servlet.

JSP Initialization:

When a container loads a JSP it invokes the jspInit() method before servicing any requests.

If you need to perform JSP-specific initialization, override the jspInit() method:

```
public void jspInit(){
  // Initialization code...
}
```

Typically, initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP Execution:

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.

The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows:

```
void _jspService(HttpServletRequest request, HttpServletResponse response)
{
  // Service handling code...
}
```

The _jspService() method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e., GET, POST, DELETE, etc.

JSP Cleanup:

- The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.
- The jspDestroy() method is the JSP equivalent of the destroy method for servlets.
- Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

Signature of jspDestroy() method:

```
public void jspDestroy()
{
  // write cleanup code here.
}
```

Basic tags of JSP:

Scriptlet tag: A scriptlet tag is used to execute java source code in JSP.

The declaration of scriptlet tag is placed inside the _jspService() method.

syntax of Scriptlet:

<% code fragment %>

XML equivalent of the above syntax as follows:

<jsp:scriptlet>

code fragment

</jsp:scriptlet>

Example:

```
<html>
<head><title>Hello World</title></head>
<body>
        Hello World!<br/>
        <%
        out.println("Welcome to JSP world");
        %>
</body>
</html>
```

Declarations tag:

A declaration tag is used to declares variables and methods in the JSP file.

The declaration of jsp declaration tag is placed outside the _jspService() method.

<%! declaration; %>

XML equivalent of the above syntax as follows:

<jsp:declaration>

 code fragment

</jsp:declaration>

Example for JSP Declarations tag:

<%! int i = 0; %>

<%! int a, b, c; %>

<%! Date date = new Date(); %>

Expression tag:

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.

Syntax of Expression tag:

<%= expression %>

XML equivalent of the above syntax as follows:

<jsp:expression>

expression

</jsp:expression>

Example:

```
<html>
<head><title>A Comment Test</title></head>
<body>
        <p>
                Today's date: <%= (new java.util.Date()).toLocaleString()%>
        </p>
</body>
</html>
```

JSP Comments tag:

JSP comment marks the text or the statements that the JSP container should ignore.

Syntax of the JSP comments:

<%-- This is JSP comment --%>

Example:

```
<html>
<head><title>A Comment Test</title></head>
<body>
<h2>A Test of Comments</h2>
   <%-- This comment will not be visible in the page source --%>
</body>
</html>
```

JSP Implicit Objects:

Objects	Description
request	HttpServletRequest object associated with the request.
response	HttpServletResponse object associated with the response to the client.
out	PrintWriter object used to send output to the client.
session	HttpSession object is used to get the session.
application	ServletContext object associated with application context.
config	ServletConfig object associated with the page
pageContext	The pageContext object can be used to set,get or remove attribute from page, resquest,session,application scopes
page	Used to call the methods defined by the translated servlet class.
Exception	The pageContext object can be used to set,get or remove attribute from

JSP Directives:

JSP directive intimates to the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

page directive

include directive

taglib directive

JSP Page directive:

<%@ page attribute="value" %>

Attributes of page directive:

import	contentType
extends	info
buffer	language
isELIgnored	isThreadSafe
autoFlush	session
pageEncoding	errorPage
isErrorPage	

Example:

```
<html>

<body>

<%@ page import="java.util.Date" %>

Today's Date is: <%= new Date() %>

</body>

</html>
```

Jsp Include Directive:

The include directive is used to include the contents of any resource it may be jsp file, html file or text file. The include directive includes the original content of the included resource at page translation time

Advantage:

Code Reusability

Syntax:

```
<%@ include file="resourceName" %>
```

Example:

```
<html>

<body>

<%@ include file="header.jsp" %>

Today is: <%= new java.util.Date() %>

</body>

</html>
```

JSP Taglib directive:

The JSP taglib directive is used to define a tag library for custom defines tags.

We will use the TLD (Tag Library Descriptor) file to define the tags

syntax:

<%@ taglib uri="urlofthetaglibrary" prefix="prefixoftaglibrary" %>

JSP Actions tags:

JSP Action Tags	Description
jsp:forward	forwards the request and response to another resource.
jsp:include	includes another resource.
jsp:useBean	creates or locates bean object.
jsp:setProperty	sets the value of property in bean object.
jsp:getProperty	get the value of property of the bean.
jsp:plugin	embeds another components tag, such as applet.
jsp:param	sets the parameter value. It is used in forward and include operation .
jsp:element	Defines XML elements dynamically.

Example:

Follow below steps for creating web project:

1. open eclipse
2. go to File --> New --> select "Dynamic Web Project"
3. Fill below values

   Project name: JspActionTagExample

   Dynamic web module version: 3.0 0r later version

   click Next

   select Generate web.xml deployment descriptor
4. Create registration.html file under "WebContent" folder

```
<!DOCTYPE html>
<html><body>
<form action="/JspActionTagExample/registrationProcess.jsp" method="post">
 Fill below details for Applying the voter card : <br>
 <table>
    <tr>
                                <td>First Name                          : </td>
                                <td><input type="text" name="fname" id ="fname"></td>
                </tr>
                <tr>
                                <td>Last Name                                           : </td>
                                <td><input type="text" name="lname" id ="lname"> </td>
                </tr>
                <tr>
                                <td>Age                                                                         : </td>
                                <td><input type="text" name="age" id ="age"> </td>
                </tr>
                <tr>
                                <td>Address                                             : </td>
                                <td><input type="text" name="address" id ="address"> </td>
                </tr>
 </table>
 <input type="submit" value="Submit">
 </form>
</body> </html>
```

5. Create registrationProcess.jsp under webContent folder

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<body>
                <%
                        int age = Integer.parseInt(request.getParameter("age"));
                  if(age > 18) {
                %>
                <jsp:forward page="confirmation.jsp">
                        <jsp:param name="message" value="Voter card registration process is successful"/>
                </jsp:forward>
                <%
                  } else {
                        out.println("You are not eligible for voter card");
                   }
                 %>
</body>
</html>
```

6. Create confirmation.jsp under webContent folder

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<body>
  <%
            String message = request.getParameter("message");
            out.println(message);
  %>
</body>
</html>
```

Note: Deploy the application in tomcat server and access the application using below url.
http://localhost:8080/JspActionTagExample

<jsp:useBean> tag:

<jsp:useBean id= "instanceName" scope= "page | request | session | application"

class= "packageName.className" type= "packageName.className" beanName="packageName.className >

</jsp:useBean>

Attribute of useBean tag:

id: is used to identify the bean in the specified scope.

scope: represents the scope of the bean. It may be page, request, session or application. The default scope is page.

class: instantiates the specified bean class (i.e. creates an object of the bean class) but it must have no-arg constructor.

type: provides the bean a data type if the bean already exists in the scope. It is mainly used with class or beanName attribute. If you use it without class or beanName, no bean is instantiated.

beanName: instantiates the bean using the java.beans.Beans.instantiate() method.

Example:

<jsp:useBean id="student" class="com.test.bean.Student" scope="session" />

jsp:setProperty tag:

<jsp:setProperty name="instanceOfBean" property= "*" |
property="propertyName" param="parameterName" |
property="propertyName" value="{ string | <%= expression %>}"
/>

Example:

<jsp:setProperty name="student" property="name" param="name">
(Or)
<jsp:setProperty name="student" property="name" value="Ramesh">

jsp:getProperty tag:

<jsp:getProperty name="instanceOfBean" property="propertyName" />

Example:

<jsp:getProperty name="student" property="name" />

Example:

1. Create dynamic web project name as “JspBeanTagExample” and paste below files in WebContent folder.

2 student.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
		pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
		<form action="/JspBeanTagExample/registration.jsp">
			Enter your name: <input type="text" name="name"> <br>
			Enter your address: <input type="text" name="address"> <br>
			Enter your email:<input type="text" name="email"> <br>
			<input type="submit" value="Send" name="send">
		</form>
</body>
</html>
```

3. registration.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
                <jsp:useBean id="studentBean" scope="session" class="com.test.model.Student"></jsp:useBean>
                <jsp:setProperty name="studentBean" property="name" param="name"/>
                <jsp:setProperty name="studentBean" property="address" param="address"/>
                <jsp:setProperty name="studentBean" property="email" param="email"/>
                Thanks for Registration
                <br>
                <a href="/JspBeanTagExample/confirmation.jsp">Show My Details</a>
</body>
</html>
```

4. confirmation.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
 Your details are : <br>
 <jsp:useBean id="studentBean" scope="session" class="com.test.model.Student"></jsp:useBean>
 Name : <jsp:getProperty property="name" name="studentBean"/> <br>
 Address : <jsp:getProperty property="address" name="studentBean"/> <br>
 Email : <jsp:getProperty property="email" name="studentBean"/> <br>
 Have a nice day
</body>
</html>
```

Custom Tags:

Custom tag are the user defined tags in jsp application to insert some content as response to the requested client.

Advantages:

- Separation of business logic from JSP The custom tags separate the the business logic from the JSP page so that it may be easy to maintain.
- Re-usability The custom tags makes the possibility to reuse the same business logic again and again.

syntax:
1.custom tag having no body

<prefix:tagname attribute1=value1....attributen=valuen />

2.Custom tag having body:

<prefix:tagname attribyte1=value1....attributen=valuen >

body code

</prefix:tagname>

Steps to create custom tag:

We need to follow the following 3 steps to define custom tags in jsp application.

1.we need to define a class which is called as tag handler class for respective custom tag. so that this tag handler class will handle the respective custom tag functionality in the application.

This tag handler class should be placed inside the class folder of the application.

2.Create a TLD(Tag Library Descriptor) file and provide the description about the custom tag name and related tag handler class names. So that JSP engine will create an object of the respective tag handler class when ever custom tag is encountered.

This TLD file should be places inside WEB-INF folder.

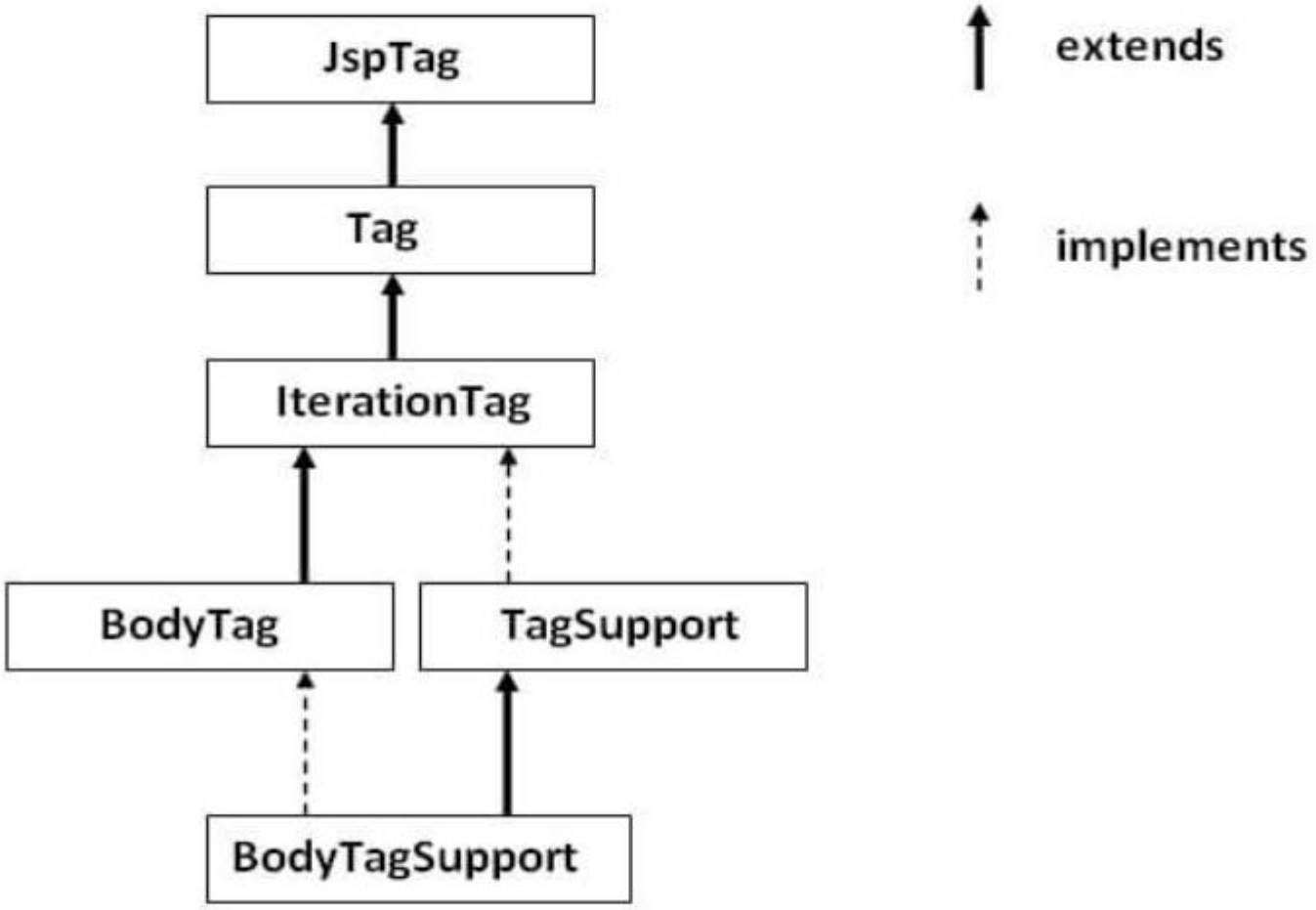
3.Add directive tag in the jsp file, this will provides the information to jsp engine regarding the location and name of the TLD file, prefix of custom tag.

So that we can use the respective custom tag many number of times in respective jsp file.

- Note: Custom tag can be defined in the jsp application as a simple custom tag without body content or custom tag with body content.

JSP Custom Tag API:

The javax.servlet.jsp.tagext package contains classes and interfaces for JSP custom tag API.



JspTag interface:

The JspTag is the root interface for all the interfaces and classes used in custom tag. It is a marker interface.

Tag interface:

The Tag interface is the sub interface of JspTag interface. It provides methods to perform action at the start and end of the tag.

Fields of Tag interface:

There are four fields defined in the Tag interface.

public static int EVAL_BODY_INCLUDE: it evaluates the body content.

public static int EVAL_PAGE: it evaluates the JSP page content after the custom tag.

public static int SKIP_BODY: it skips the body content of the tag.

public static int SKIP_PAGE: it skips the JSP page content after the custom tag.

Methods of Tag interface:

public void setPageContext(PageContext pc): it sets the given PageContext object.

public void setParent(Tag t): it sets the parent of the tag handler.

public Tag getParent(): it returns the parent tag handler object.

public int doStartTag()throws JspException: it is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the start of the tag.

Default return value is SKIP_BODY.

public int doEndTag()throws JspException: it is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the end of the tag.

Default return value is EVAL_PAGE

public void release(): it is invoked by the JSP page implementation object to release the state.

IterationTag interface:

The IterationTag interface is the sub interface of the Tag interface. It provides an additional method to reevaluate the body.

Field of IterationTag interface:

One field available in the IterationTag interface.

public static int EVAL_BODY_AGAIN: it reevaluates the body content.

Method of Tag interface:

One method avilable in IterationTag interface.

public int doAfterBody()throws JspException

It is invoked by the JSP page implementation object after the evaluation of the body.

If this method returns EVAL_BODY_INCLUDE, body content will be reevaluated, if it returns SKIP_BODY, no more body content will be evaluated.

Default return value is SKIP_BODY

BodyTag interface:

This is extension of IterationTag interface. It has one method and below is the signature of the method.

Public BodyContent getBodyContent();

Where BodyContent is class defined in the same package.

TagSupport class: It is the implementation class of IterationTag interface.
This class implements all methods available in IterationTag interface.

For example doStartTag(), doEndTag(),doAfterBody() etc.

BodyTagSupport class: It is the implementation class of BodyTag interface. This class implements all methods available in BodyTag interface.

For example doStartTag(),doEndTag(),doAfterBody(),doBodyContent() etc.

Example1:

1. Create a dynamic project and provide name as "JspCustomTag"

2. create HelloTag handler class put in source folder.

```
package com.test.taghandler;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
public class HelloTag extends TagSupport {
		private static final long serialVersionUID = 1L;
		public int doStartTag() {
			JspWriter out = pageContext.getOut();
			try {
				out.println("Hello client, this is for custom tag");
				out.println("<br>");
				out.println("I hope you are fine");
				out.println("<br>");
				out.println("Bye Bye");
			} catch (IOException e) {
				e.printStackTrace();
			}
			return SKIP_BODY;
		}
}
```

3. create mytld.tld file under WEB-INF folder

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>

            <tlib-version>1.0</tlib-version>
            <jsp-version>1.2</jsp-version>
            <short-name>simple</short-name>

            <tag>
                        <name>hello</name>
                        <tag-class>com.test.taghandler.HelloTag</tag-class>
            </tag>
</taglib>
```

4. create first.jsp file under WebContent folder

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
		pageEncoding="ISO-8859-1"%>

<%@ taglib uri="/WEB-INF/mytag.tld" prefix="mytag"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
		This is the custom tag hello
		<br> Custom Tag content:
		<br>
		<mytag:hello />
		<br> Have a nice day
</body>
</html>
```

5. web.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID"
version="4.0">
  <display-name>JspCustomTag</display-name>
</web-app>
```

Note: Deploy above application in tomcat server and run the tomcat server.

Access the application using below url

http://localhost:8080/JspCustomTag/first.jsp

Note: If you want to insert after end of the tag, then implement doEndTag() method.

```
public class HelloTag extends TagSupport {
		private static final long serialVersionUID = 1L;
		public int doStartTag() {
				JspWriter out = pageContext.getOut();
				try {
						out.println("Hello client, this is for custom tag");  out.println("<br>");
						out.println("I hope you are fine");  out.println("<br>");
						out.println("Bye Bye");
				} catch (IOException e) {
						e.printStackTrace();
				}
				return SKIP_BODY;
		}
		public int doEndTag() {
				JspWriter out = pageContext.getOut();
				try {
						out.println("<br>");
						out.println("**********************");
				} catch (IOException e) {
						e.printStackTrace();
				}
				return EVAL_PAGE;
	}
}
```

Example2 : custom tag with the body content.

1.create dynamic web project (JspCustomTagWithBody)

2. create mytag.tld file under WEB-INF folder

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>

            <tlib-version>1.0</tlib-version>
            <jsp-version>1.2</jsp-version>
            <short-name>simple</short-name>

            <tag>
                        <name>caps</name>
                        <tag-class>com.test.taghandler. CapitalTagHandler </tag-class>
                        <body-content>jsp</body-content>
            </tag>
</taglib>
```

3. Create CapitalTagHandler.java class under src folder.

```
package com.test.taghandler;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class CapitalTagHandler extends BodyTagSupport {
		public int doAfterBody() {
				BodyContent body = getBodyContent();
				String text = body.getString();
				JspWriter out = body.getEnclosingWriter();
				try {
						out.println(text.toUpperCase());
				} catch (IOException e) {
						e.printStackTrace();
				}
				return SKIP_BODY;
		}

}
```

4. Create index.jsp file under WebContent folder

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
		pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/mytag.tld" prefix="caps"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
		Welcome to client <br>
		<mytag:caps>
			Hello Boss
			<br>
			How are you
  </mytag:caps>
</body>
</html>
```

5. Create web.xml file under WEB-INF folder

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

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID" version="4.0">

  <display-name>JspCustomTagWithBody</display-name>

</web-app>
```

Note: Deploy above application in tomcat server and run the tomcat server.

Access the application using below url:

http://localhost:8080/JspCustomTagWithBody/index.jsp

Example3: Custom tag with attributes.

1. create dynamic web project (JspCustomTagWithAttributes)

2. create custtagAttribute.jsp in WebContent folder

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
		pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/mytag.tld" prefix="filter"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
		<mytag:filter replace="bad" with="good">
			Tom is a bad boy
		</mytag:filter>
		<br>
		<mytag:filter replace="he" with="she">
			He is cooking
		</mytag:filter>
		<br> Wish you all the best.
</body>
</html>
```

2. create FilterTagHandler.java under src folder

```
public class FilterTagHandler extends BodyTagSupport {
		String replace;  String with;
		public void setReplace(String replace) {
			this.replace = replace;
		}
		public void setWith(String with) {
			this.with = with;   }
		public int doAfterBody() {
			String filteredText = "";
			BodyContent body = getBodyContent();
			String text = body.getString();
			String[] textArray = text.split(" ");
			for (String word : textArray) {
				if (word.equalsIgnoreCase(replace)) {
					word = with;    }
				filteredText = filteredText + " " + word;
			}
			JspWriter out = body.getEnclosingWriter();
			try {
				out.println(filteredText.toUpperCase());
			} catch (IOException e) {
				e.printStackTrace();
			}
			return SKIP_BODY;
		}   }
```

3. Create mytag.tld file under WEB-INF folder

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
                <tlib-version>1.0</tlib-version>
                <jsp-version>1.2</jsp-version>
                <short-name>simple</short-name>
                <tag>
                        <name>filter</name>
                        <tag-class>com.test.taghandler.FilterTagHandler</tag-class>
                        <body-content>jsp</body-content>
                        <attribute>
                                <name>replace</name>
                                <required>true</required>
                        </attribute>
                        <attribute>
                                <name>with</name>
                                <required>true</required>
                        </attribute>
                </tag>
</taglib>
```

4. create web.xml under WEB-INF folder

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

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID"
version="4.0">

  <display-name>JspCustomTagWithAttributes</display-name>


</web-app>
```

Note: Access the application using below url

http://localhost:8080/JspCustomTagWithAttributes/custtagAttribute.jsp

Data base integration:

1. Create dynamic web application (JspDBIntegration)
2. Create student.jsp file under WebContent folder.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="registration" method="post">
        Student Registration Form: <br>
        <table>
                <tr>
                   <td>First Name</td>
                   <td>:</td>
                   <td> <input type="text" name="fname" id="fname"></td>
                </tr>
```

```
            <tr>
                <td>Last Name</td>
                <td>:</td>
                <td> <input type="text" name="lname" id="lname"></td>
            </tr>
            <tr>
                <td>Age</td>
                <td>:</td>
                <td> <input type="text" name="age" id="age"></td>
            </tr>
            <tr>
                <td>Address</td>
                <td>:</td>
                <td> <input type="text" name="address" id="address"></td>
            </tr>
            <tr>
                <td>Class:</td>
                <td>:</td>
                <td> <input type="text" name="stand" id="stand"></td>
            </tr>
    </table>
    <input type="submit" value="Regsistration">
</form>
</body>
</html>
```

2. Create registration.jsp page

```
<%@page import="com.test.business.StudentBusiness"%>
<%@page import="com.test.business.StudentBusinessImpl"%>
<jsp:useBean id="student" class="com.test.model.Student" scope="request"></jsp:useBean>
<jsp:setProperty property="*" name="student"/>

<%
	StudentBusiness studentBusiness = new StudentBusinessImpl();
		try {
			boolean status = studentBusiness.createStudent(student);
			if (status) {
				response.sendRedirect("success.html");
			} else {
				response.sendRedirect("error.html");
			}
		} catch (ApplicationException e) {
			response.sendRedirect("error.html");
		}
%>
```

Create success.jsp under WebContent folder

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    Your Registration is successful.
</body>
</html>
```

Create error.jsp under WebContent folder

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    There was some problem in the application, please try again.
</body>
</html>
```

2. Create the bean class/model class

```
package com.test.model;

public class Student implements Serializable {

    private String name;

    private int age;

    private String address;

    private String stand;

    private int id;

    public Student() {
    }
    // generate setters and getter methods.
}
```

3. Create business interface and its implementation class.

```
package com.test.business;
import com.test.exception.ApplicationException;
import com.test.model.Student;
public interface StudentBusiness {
	boolean createStudent(Student student) throws ApplicationException;
}
package com.test.business;
import com.test.dao.StudentDao;
import com.test.dao.StudentDaoImpl;
import com.test.exception.ApplicationException;
import com.test.model.Student;
public class StudentBusinessImpl implements StudentBusiness {
	@Override
	public boolean createStudent(Student student) throws ApplicationException {
			StudentDao studentDao = new StudentDaoImpl();
			// Generate the id.
			int maxId = studentDao.getMaxStudentId();
			if (maxId > 0) {
				student.setId(maxId + 1);
			}
			boolean status = studentDao.createStudent(student);
		return status;
	}
```

4. Create Dao interface and its implementation class.

```
public interface StudentDao {
    boolean createStudent(Student student) throws ApplicationException;
    int getMaxStudentId() throws ApplicationException;

}

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.test.exception.ApplicationException;
import com.test.model.Student;
import com.test.util.ConnectionUtil;

public class StudentDaoImpl implements StudentDao {
```

```
@Override
    public boolean createStudent(Student student) throws ApplicationException {
                Connection conn = ConnectionUtil.getConnection();
                try {
                            PreparedStatement psmt = conn.prepareStatement("insert into student(id,name,age,address,stand)
    values(?,?,?,?,?)");
                            psmt.setInt(1, student.getId());
                            psmt.setString(2, student.getName());
                            psmt.setInt(3, student.getAge());
                            psmt.setString(4, student.getAddress());
                            psmt.setString(5, student.getStand());
                            int result = psmt.executeUpdate();
                            return result > 0 ? true:false;
                } catch (SQLException e) {
                            throw new ApplicationException(e.getMessage());
                }finally {
                            ConnectionUtil.closeResources(conn);
                }
    }
```

```
@Override
   public int getMaxStudentId() throws ApplicationException {
            Connection conn = ConnectionUtil.getConnection();
            try {
                     Statement stmt = conn.createStatement();
                     ResultSet rs = stmt.executeQuery("select max(id) from student");
                     while (rs.next()) {
                              return rs.getInt(1);
                     }
            } catch (Exception e) {
                     throw new ApplicationException(e.getMessage());
            }
            return 0;
   }

}
```

4. create connection util class

```
package com.test.util;
Import java.sql.*;
public class ConnectionUtil {
    public static Connection getConnection() throws ApplicationException {
            Connection conn = null;
            try {
                    Class.forName("oracle.jdbc.driver.OracleDriver");
                    conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");
            } catch (Exception e) {
                    throw new ApplicationException(e.getMessage());
            }
            return conn;
    }
    public static void closeResources(Connection conn) {
            if(conn != null) {
                    try {
                        conn.close();
                    } catch (SQLException e) {
                            e.printStackTrace();
                    }
            }
    }
}
```

5. Create the ApplicationException class.

```
package com.test.exception;

public class ApplicationException extends Exception {

    private static final long serialVersionUID = 1L;

    public ApplicationException(String message) {
            super(message);
    }

    public ApplicationException(Throwable t) {
            super(t);
    }
}
```

Note : Access the above application using below url:

http://localhost:8081/JspDBIntegration/student.jsp