**JSP**

1. **What is a web browser?**
2. A web browser (commonly referred to as a browser) is a software application for retrieving, presenting and traversing information resources on the World Wide Web.

An information resource is identified by a Uniform Resource Identifier (URI/URL) and may be a web page, image, video or other piece of content. Hyperlinks present in resources enable users easily to navigate their browsers to related resources.

Although browsers are primarily intended to use the World Wide Web, they can also be used to access information provided by web servers in private networks or files in file systems.

The major web browsers are Firefox, Internet Explorer, Google Chrome, Opera, and Safari.

1. **What is HTML?**
2. HyperText Markup Language, commonly referred to as HTML, is the standard [markup language](https://en.wikipedia.org/wiki/Markup_language) used to create [web pages](https://en.wikipedia.org/wiki/Web_page). [Web browsers](https://en.wikipedia.org/wiki/Web_browser) can read HTML files and render them into visible or audible web pages.

HTML describes the structure of a [website](https://en.wikipedia.org/wiki/Website) [semantically](https://en.wikipedia.org/wiki/Semantic) along with cues for presentation, making it a markup language, rather than a [programming language](https://en.wikipedia.org/wiki/Programming_language).

HTML elements form the building blocks of all websites. HTML allows [images and objects](https://en.wikipedia.org/wiki/Img_(HTML_element)) to be embedded and can be used to create [interactive forms](https://en.wikipedia.org/wiki/Fieldset). It provides a means to create [structured documents](https://en.wikipedia.org/wiki/Structured_document) by denoting structural semantics for text such as headings, paragraphs, lists, [links](https://en.wikipedia.org/wiki/Hyperlink), quotes and other items.

The language is written in the form of [HTML elements](https://en.wikipedia.org/wiki/HTML_element) consisting of tags enclosed in [angle brackets](https://en.wikipedia.org/wiki/Bracket#Angle_brackets) (like <html>). Browsers do not display the HTML tags and scripts, but use them to interpret the content of the page.

HTML can embed [scripts](https://en.wikipedia.org/wiki/Scripting_language) written in languages such as [JavaScript](https://en.wikipedia.org/wiki/JavaScript) which affect the behavior of HTML web pages. Web browsers can also refer to [Cascading Style Sheets](https://en.wikipedia.org/wiki/Cascading_Style_Sheets) (CSS) to define the look and layout of text and other material.

The [World Wide Web Consortium](https://en.wikipedia.org/wiki/World_Wide_Web_Consortium) (W3C), maintainer of both the HTML and the CSS standards, has encouraged the use of CSS over explicit presentational HTML since 1997.

1. **How HTML pages are transmitted between server and browser?**
2. The World Wide Web is composed primarily of HTML documents transmitted from web servers to web browsers using the Hypertext Transfer Protocol (HTTP). However, HTTP is used to serve images, sound, and other content, in addition to HTML. To allow the web browser to know how to handle each document it receives, other information is transmitted along with the document. This meta data usually includes the MIME type (e.g. text/html or application/xhtml+xml) and the character encoding (see Character encoding in HTML).

In modern browsers, the MIME type that is sent with the HTML document may affect how the document is initially interpreted. A document sent with the XHTML MIME type is expected to be well-formed XML; syntax errors may cause the browser to fail to render it. The same document sent with the HTML MIME type might be displayed successfully, since some browsers are more lenient with HTML.

The W3C recommendations state that XHTML 1.0 documents that follow guidelines set forth in the recommendation's Appendix C may be labeled with either MIME Type.[64] XHTML 1.1 also states that XHTML 1.1 documents should[65] be labeled with either MIME type.[66]

1. **What is JSP?**
2. Java Server Page (JSP) is a technology for controlling the content or appearance of Web pages through the use of servlets, small programs that are specified in the Web page and run on the Web server to modify the Web page before it is sent to the user who requested it.

It is a technology that helps software developers to create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP uses the Java programming language.

A JSP page is a text document that contains two types of text:

**Static Contents:** Static contents include static html/xml, images, client side javascript etc. Static contents are called static because, there is no change in these components with each page load. E.g. there may be a logo image in a page, this image does not get changed frequently. Hence it can be called as static content.

**Dynamic Contents:** These contents may get changed with every page load. Basically, we are looking at the data part of a page presented to user. Mostly application web front ends display data sent by the server. Server side processing generates data through middle tier or server side processing. Server sends this data to the client i.e. the web browser. Web browser displays this data along with the static contents. Such contents are called as dynamic contents.

The recommended file extension for the source file of a JSP page is .jsp.

The page can be composed of a top file that includes other files that contain either a complete JSP page or a fragment of a JSP page.

The recommended extension for the source file of a fragment of a JSP page is .jspf.

To deploy and run JavaServer Pages, a compatible web server with a servlet container, such as Apache Tomcat or Jetty, is required.

1. **What is JSP Architecture?**
2. Architecturally, JSP may be viewed as a high-level abstraction of Java servlets. JSPs are translated into servlets at runtime; each JSP servlet is cached and re-used until the original JSP is modified.

JSP can be used independently or as the view component of a server-side model–view–controller design, normally with JavaBeans as the model and Java servlets (or a framework such as Apache Struts) as the controller. This is a type of Model 2 architecture(MVC2).

JSP allows Java code and certain pre-defined actions to be interleaved with static web markup content, such as HTML, with the resulting page being compiled and executed on the server to deliver a document. The compiled pages, as well as any dependent Java libraries, contain Java bytecode rather than machine code. Like any other Java program, they must be executed within a Java virtual machine (JVM) that interacts with the server's host operating system to provide an abstract, platform-neutral environment.

JSPs are usually used to deliver HTML and XML documents, but through the use of OutputStream, they can deliver other types of data as well.

The Web container creates JSP implicit objects like pageContext, servletContext, session, request & response.

Servlet class created from JSP.

1. **What are advantages of using JSP?**
2. JSP offer several advantages as listed below:
3. Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself.
4. JSP are always compiled before it's processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
5. Java Server Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP etc.
6. JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.
7. It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.
8. Embedding of Java code in HTML pages.
9. Platform independence.
10. Creation of database-driven Web applications.
11. Server-side programming capabilities.

Following is the JSP code in hello.jsp:

<HTML>

<HEAD><TITLE>The Hello User JSP</TITLE></HEAD>

<BODY>

<% String user=request.getParameter("user"); %>

<H3>Welcome <%= (user==null) ? "" : user %>!</H3>

<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>

<B>Enter name:</B>

<FORM METHOD=get>

<INPUT TYPE="text" NAME="user" SIZE=15>

<INPUT TYPE="submit" VALUE="Submit name">

</FORM>

</BODY>

</HTML>

Following is the generated page implementation class Java code (\_hello.java), as generated by Oracle JSP 1.1.x.x releases:

package \_test;

import oracle.jsp.runtime.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

import java.io.\*;

import java.util.\*;

import java.lang.reflect.\*;

import java.beans.\*;

public class \_hello extends oracle.jsp.runtime.HttpJsp {

public final String \_globalsClassName = null;

// \*\* Begin Declarations

// \*\* End Declarations

public void \_jspService(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {

/\* set up the intrinsic variables using the pageContext goober:

\*\* session = HttpSession

\*\* application = ServletContext

\*\* out = JspWriter

\*\* page = this

\*\* config = ServletConfig

\*\* all session/app beans declared in globals.jsa

\*/

JspFactory factory = JspFactory.getDefaultFactory();

PageContext pageContext = factory.getPageContext( this, request, response, null, true, JspWriter.DEFAULT\_BUFFER, true);

// Note: this is not emitted if the session directive == false

HttpSession session = pageContext.getSession();

if (pageContext.getAttribute(OracleJspRuntime.JSP\_REQUEST\_REDIRECTED, PageContext.REQUEST\_SCOPE) != null) {

pageContext.setAttribute(OracleJspRuntime.JSP\_PAGE\_DONTNOTIFY, "true", PageContext.PAGE\_SCOPE);

factory.releasePageContext(pageContext);

return;

}

ServletContext application = pageContext.getServletContext();

JspWriter out = pageContext.getOut();

hello page = this;

ServletConfig config = pageContext.getServletConfig();

try {

// global beans

// end global beans

out.print(\_\_jsp\_StaticText.text[0]);

String user=request.getParameter("user");

out.print(\_\_jsp\_StaticText.text[1]);

out.print( (user==null) ? "" : user );

out.print(\_\_jsp\_StaticText.text[2]);

out.print( new java.util.Date() );

out.print(\_\_jsp\_StaticText.text[3]);

out.flush();

}

catch( Exception e) {

try {

if (out != null) out.clear();

}

catch( Exception clearException) {

}

pageContext.handlePageException( e);

}

finally {

if (out != null) out.close();

factory.releasePageContext(pageContext);

}

}

private static class \_\_jsp\_StaticText {

private static final char text[][]=new char[4][];

static {

text[0] = "<HTML>\r\n<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>\r\n<BODY>\r\n".toCharArray();

text[1] = "\r\n<H3>Welcome ".toCharArray();

text[2] = "!</H3>\r\n<P><B> Today is ".toCharArray();

text[3] = ". Have a nice day! :-)</B></P>\r\n<B>Enter name:</B>\r\n<FORM METHOD=get>\r\n<INPUT TYPE=\"text\" NAME=\"user\" SIZE=15>\r\n<INPUT TYPE=\"submit\" VALUE=\"Submit name\">\r\n</FORM>\r\n</BODY>\r\n</HTML>".toCharArray();

}

}

}

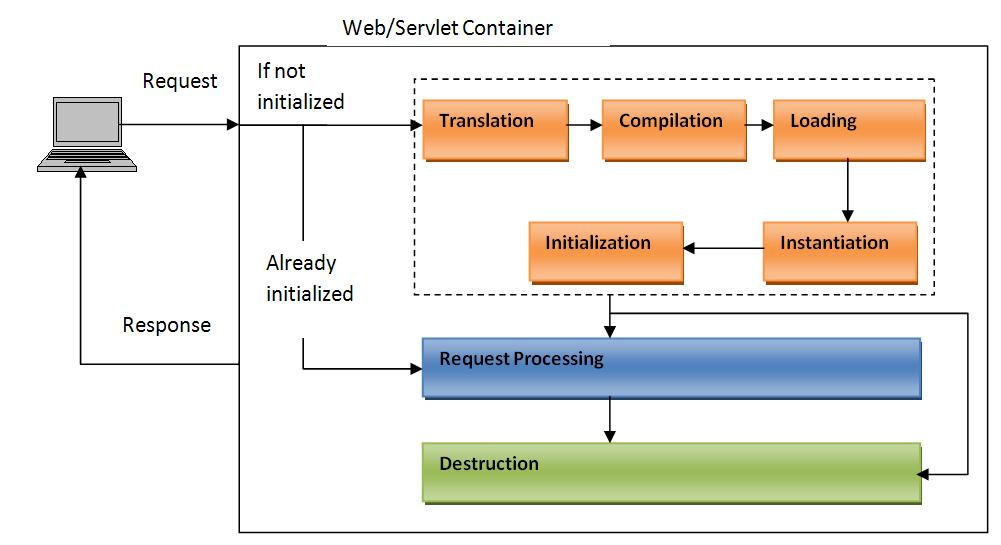
1. **What are the advantages of JSP over JavaScript?**
2. JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.
3. **What are the advantages of JSP over Static HTML?**
4. Regular HTML, of course, cannot contain dynamic information.
5. **Explain lifecycle of a JSP?**
6. A JSP page services requests as a servlet. Thus, the life cycle and many of the capabilities of JSP pages (in particular the dynamic aspects) are determined by Java Servlet technology.

When a request is mapped to a JSP page, the web container first checks whether the JSP page’s servlet is older than the JSP page. If the servlet is older, the web container translates the JSP page into a servlet class and compiles the class. During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.

Following diagram shows the different stages of life cycle of JSP.

Broadly, these stages can be classified into three.

* Instantiation
* Request Processing
* Destruction



**Instantiation:**

When a web container receives a jsp request (may be first or subsequent), it checks for the jsp’s servlet instance. If no servlet instance is available or if it is older than the jsp, then, the web container creates the servlet instance using following stages.

* Translation
* Compilation
* Loading
* Instantiation
* Initialization

***Translation:***

Web container translates (converts) the jsp code into a servlet code. This means that jsp is actually a servlet. After this stage, there is no jsp, everything is a servlet. This task will create a complete jsp page, by considering all included components. Here on, the static content and dynamic contents are treated differently. The resultant is a java class instead of an html page (which we wrote). This is how the structure of a jsp compiled into a java class will be.

package org.apache.jsp.WEB\_002dINF.jsp;

import javax.servlet.\*;

import javax.servlet.http.\*;

import javax.servlet.jsp.\*;

public final class firstJsp\_jsp extends org.apache.jasper.runtime.HttpJspBase

implements org.apache.jasper.runtime.JspSourceDependent {

private static final JspFactory \_jspxFactory = JspFactory.getDefaultFactory();

............

............

public Object getDependants() {

return \_jspx\_dependants;

}

public void \_jspInit() {

............

............

}

public void \_jspDestroy() {

............

............

}

public void \_jspService(HttpServletRequest request,

HttpServletResponse response)

throws java.io.IOException, ServletException {

............

............

}

............

............

}

***Compilation:***

The generated servlet is compiled to validate the syntax. As it is a java class, the compilation is done using javac command. This will generate the byte code to be run on JVM.

***Loading:***

The compiled byte code is loaded by the class loader used by web container. This is a standard process of using any java class.

***Instantiation:***

In this step, instance of the servlet class is created so that it can serve the request.

***Initialization:***

Initialization is done by calling the jspInit() method. This is one time activity at the start of the initialization process. Initialization will make the ServletContext and ServletConfig objects available. One can access many attributes related to the web container and the servlet itself. After initialization the servlet is ready to process requests.

**Request Processing:**

Entire initialization process is done to make the servlet available in order to process the incoming request. jspService() is the method that actually processes the request. It prints the response in html (any other) format, using ‘out’ object.

**Destroy:**

Whenever the server is shutting down or when the server needs memory, the server removes the instance of the servlet. The destroy method jspDestroy() can be called by the server after initialization and before or after request processing. Once destroyed the jsp needs to be initialized again.

Just to summarize, web container handles incoming requests to a jsp by converting it into a servlet and then by using this servlet to generate the response. Also when the server shuts down, the container needs to clear the instances.

<https://docs.oracle.com/javaee/5/tutorial/doc/bnahe.html>

<http://www.deepakgaikwad.net/index.php/2009/04/15/jsp-life-cycle.html>

1. **List down differences between ServletContext vs. ServletConfig.**
2. Both ServletContext and ServletConfig are basically configuration objects which are used by servlet container to initialize various parameter of web application. But they have some difference in terms of **scope and availability**

**ServletConfig** is implemented by the servlet container **to initialize a single servlet** using init(). That is, you can pass initialization parameters to the servlet using the web.xml deployment descriptor. For understanding, this is similar to a constructor in a java class.

**ServletContext** is implemented by the servlet container **for all servlet** to communicate with its servlet container, for example, to get the MIME type of a file, to get dispatch requests, or to write to a log file. That is to get detail about its execution environment. It is applicable only within a single Java Virtual Machine. If a web application is distributed between multiple JVM this will not work. For understanding, this is like a application global variable mechanism for a single web application deployed in only one JVM.

|  |  |
| --- | --- |
| **Servlet Config** | **Servlet Context** |
| Servlet config object represent single servlet | It represent whole web application running on particular JVM and common for all the servlet |
| Its like local parameter associated with particular servlet | Its like global parameter associated with whole application |
| It’s a name value pair defined inside the servlet section of web.xml file so it has servlet wide scope | ServletContext has application wide scope so define outside of servlet tag in web.xml file. |
| getServletConfig() method is used to get theconfig object | getServletContext() method is  used to get the context object. |
| for example shopping cart of a user is a specific to particular user so here we can use servlet config | To get the MIME type of a file or application session related information is stored using servlet context object. |
| <servlet>  <servlet-name>ServletConfigTest</servlet-name>  <servlet-class>com.rid.ServletTest</servlet-class>  <init-param>  <param-name>topic</param-name>  <param-value>Diff btwn Config & Context</param-value>  </init-param>  </servlet> | <context-param>  <param-name>globalVariable</param-name>  <param-value>com.stackoverflow</param-value>  </context-param> |

Source: <http://java67.blogspot.com/2012/09/difference-between-servletconfig-and-servletcontext-j2ee-jsp.html#ixzz3q0FsoUlt>

1. **What is a scriptlet in JSP and what is its syntax?**
2. A **JSP scriptlet** is used to contain any code fragment that is valid for the scripting language used in a page. The syntax for a scriptlet is as follows:

<%

*scripting-language-statements*

%>

When the scripting language is set to **java**, a scriptlet is transformed into a Java programming language statement fragment and is inserted into the service method of the JSP page’s servlet.

A programming language variable created within a scriptlet is accessible from anywhere within the JSP page.

The example shown below contains a scriptlet to retrieve the request parameter named username and test whether it is empty. If the if statement evaluates to true, the response page is included. Because the if statement opens a block, the HTML markup would be followed by a scriptlet that closes the block.

<%

String username = request.getParameter("username");

if ( username != null && username.length() > 0 ) {

%>

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

<%

}

%>

1. **What are JSP declarations?**
2. A **JSP declaration** is used to declare variables and methods in a page’s scripting language. The syntax for a declaration is as follows:

<%! *scripting-language-declaration* %>

When the scripting language is the Java programming language, variables and methods in JSP declarations become declarations in the JSP page’s servlet class.

#### Initializing and Finalizing a JSP Page

You can customize the initialization process to allow the JSP page to read persistent configuration data, initialize resources, and perform any other one-time activities; to do so, you override the jspInit method of the JspPage interface. You release resources using the jspDestroy method. The methods are defined using JSP declarations.

For example, Bookstore application retrieved the object that accesses the bookstore database from the context and stored a reference to the object in the variable bookDBAO in the jspInit method. The variable definition and the initialization and finalization methods jspInit and jspDestroy were defined in a declaration:

<%!

private BookDBAO bookDBAO;

public void jspInit() {

bookDBAO =

(BookDBAO)getServletContext().getAttribute("bookDB");

if (bookDBAO == null)

System.out.println("Couldn’t get database.");

}

%>

When the JSP page was removed from service, the jspDestroy method released the BookDBAO variable.

<%!

public void jspDestroy() {

bookDBAO = null;

}

%>

1. **What are JSP expressions?**
2. A **JSP expression** is used to insert the value of a scripting language expression, converted into a string, into the data stream returned to the client. When the scripting language is the Java programming language, an expression is transformed into a statement that converts the value of the expression into a String object and inserts it into the implicitout object.

The syntax for an expression is as follows:

**<%= *scripting-language-expression* %>**

Note that a semicolon is not allowed within a JSP expression, even if the same expression has a semicolon when you use it within a scriptlet.

In the following scriptlet, which gets the proxy that implements the service endpoint interface. It then invokes the sayHello method on the proxy, passing the user name retrieved from a request parameter:

<%

String resp = null;

try {

Hello hello = new HelloService().getHelloPort();

resp = hello.sayHello(request.getParameter("username"));

} catch (Exception ex) {

resp = ex.toString();

}

%>

A scripting expression is then used to insert the value of resp into the output stream:

<h2><font color="black"><%= resp %>!</font></h2>

1. **What are JSP comments?**
2. JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out" part of your JSP page.

Following is the syntax of JSP comments:

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

1. **What are JSP Directives?**
2. **Directives** are elements that relay messages to the JSP container and affect how it compiles the JSP page. The directives themselves do not appear in the XML output.

|  |  |
| --- | --- |
| **Directive** | **Description** |
| **basic syntax:**  <%@ page attribute="value" %>  **XML equivalent**  <jsp:directive.page attribute="value" /> | Defines page-dependent attributes, such as scripting language, error page, and buffering requirements. |
| **basic syntax:**  <%@ include file="relative url" >  **XML equivalent**  <jsp:directive.include file="relative url" /> | Includes a file during the translation phase. |
| **basic syntax:**  <%@ taglib uri="uri" prefix="prefixOfTag" >  **XML equivalent**  <jsp:directive.taglib uri="uri" prefix="prefixOfTag" /> | Declares a tag library, containing custom actions, used in the page |

1. **How do I include static files within a JSP page?**
2. Static resources should always be included using the JSP **include directive**. This way, the inclusion is performed just once during the translation phase. Do note that you should always supply a relative URL for the file attribute. Although you can also include static resources using the action, this is not advisable as the inclusion is then performed for each and every request.
3. **What are JSP literals?**
4. Literals are the values, such as a number or a text string, that are written literally as part of a program code. The JSP expression language defines the following literals:
5. Boolean: true and false
6. Integer: as in Java
7. Floating point: as in Java
8. String: with single and double quotes; " is escaped as \", ' is escaped as \', and \ is escaped as \\.
9. Null: null

|  |  |  |
| --- | --- | --- |
| **Expression** | **Expected Type** | **Result** |
| Hi | String | Hi |
| true | Boolean | Boolean.TRUE |
| 42 | int | 42 |

1. **What are various attributes of page directive?**
2. Page directive contains the following 13 attributes.
3. **import attribute**: This is one of the most used page directive attribute. It’s used to instruct container to import other java classes, interfaces, enums etc. while generating servlet code. This is similar to import statements in java classes, interfaces. An example of import page directive usage is:

<%@ page import="java.util.Date,java.util.List,java.io.\*" %>

1. **contentType attribute**: This attribute is used to set the content type and character set of the response. The default value of contentType attribute is”text/html; charset=ISO-8859-1″. We can use it like below.

<%@ page contentType="text/html; charset=US-ASCII" %>

contentType attribute sets the character encoding for the JSP page and for the generated response page. The default content type is text/html, which is the standard content type for HTML pages.

The two composite top-level media types are:

* Multipart
* Message

The five discrete top-level media types are:

* Text
* Image
* Audio
* Video
* Application

[**https://msdn.microsoft.com/en-us/library/ms526508(v=exchg.10).aspx**](https://msdn.microsoft.com/en-us/library/ms526508(v=exchg.10).aspx)

1. **pageEncoding attribute**: We can set response encoding type with this page directive attribute, its default value is “ISO-8859-1″.

<%@ page pageEncoding="US-ASCII" %>

The **contentType** charset is how the servlet container which runs the JSP (for example Tomcat) must send to the browser the text generated by the page. If not specified the charset is assume to be the ISO-8859-1, so only western characters can be used in that page.

UTF-8 is the current best practice, but it leads to some other issues usually ignored by coders, specially if they come from the PHP world. You can read more about those UTF-8 encoded form data and the problem with accented/cyrillic/chinese characters.

The **pageEncoding** directive is used to correctly read the JSP from the file system. Since even the JSP is a text but a file is a sequence of bytes on disk, it can be correctly read only knowing the charset to use.

The two encoding are independent and you can save JSP(s) on disk using UTF-8 and ask the container to communicate with the browser using another charset, like the ISO-8859-15.

1. **extends attribute**: This attribute is used to define the super class of the generated servlet code. This is very rarely used and we can use it if we have extended HttpServlet and overridden some of it’s implementations. For example;

<%@ page extends="org.apache.jasper.runtime.HttpJspBase" %>

1. **info attribute**: We can use this attribute to set the servlet description and we can retrieve it using Servlet interface getServletInfo() method. For example;

<%@ page info="Home Page JSP" %>

1. **buffer attribute**: We know that JspWriter has buffering capabilities, we can use this attribute to set the buffer size in KB to handle output generated by JSP page. Default value of buffer attribute is 8kb. We can define 16 KB buffer size as;

<%@ page buffer="16kb" %>

The buffer attribute specifies buffering characteristics for the server output response object.

When buffer is set to “none”, servlet output is immediately directed to the response output object.

1. **language attribute**: language attribute is added to specify the scripting language used in JSP page. It’s default value is “java” and this is the only value it can have. May be in future, JSPs provide support to include other scripting languages like C++ or PHP too.

<%@ page language="java" %>

1. **isELIgnored attribute**: We can ignore the Expression Language (EL) in JSP using this page directive attribute. Its datatype is [**Java Enum**](http://www.journaldev.com/716/java-enum-examples-with-benefits-and-class-usage) and default value is false, so EL is enabled by default. We can instruct container to ignore EL using below directive;

<%@ page isELIgnored="true" %>

1. **isThreadSafe attribute**: We can use this attribute to implement SingleThreadModel interface in generated servlet. Its an Enum with default value as true. If we set it’s value to false, the generated servlet will implement SingleThreadModel and eventually we will loose all the benefits of servlet [**multi-threading**](http://www.journaldev.com/1079/java-thread-tutorial) features. You should never set it’s value to false.

<%@ page isThreadSafe="false" %>

1. **errorPage attribute: This attribute is used to set the error page for the JSP, if the JSP throws exception, the request is redirected to the error handler defined in this attribute. It’s datatype is URI. For example;**

<%@ page errorPage="errorHandler.jsp" %>

1. **isErrorPage attribute: This attribute is used to declare that current JSP page is an error page. It’s of type Enum and default value is false. If we are creating an error handler JSP page for our application, we have to use this attribute to let container know that it’s an error page.**[**JSP implicit attribute**](http://www.journaldev.com/2038/jsp-implicit-objects-with-examples)**exception is available only to the error page JSPs. For example;**

<%@ page isErrorPage="true" %>

1. **autoFlush attribute**: autoFlush attribute is to control the buffer output. Its default value is true and output is flushed automatically when buffer is full.

If we set it to false, the buffer will not be flushed automatically and if it’s full, we will get exception for buffer overflow. We can use this attribute when we want to make sure that JSP response is sent in full or none. For example;

<%@ page autoFlush="false" %>

1. **session attribute**: By default JSP page creates a session but sometimes we don’t need session in JSP page. We can use this attribute to indicate compiler to not create session by default. It’s default value is true and session is created. To disable the session creation, we can use it like below.

<%@ page session ="false" %>

<http://www.jsptut.com/sessions.jsp>

1. **trimDirectiveWhitespaces attribute**: This attribute was added in JSP 2.1 and used to strip out extra white spaces from JSP page output. Its default value is false. It helps in reducing the generated code size, notice the generate servlet code keeping this attribute value as true and false. You will notice noout.write("\n") when it’s true.

<%@ page trimDirectiveWhitespaces ="true" %>

1. **What are JSP actions?**
2. JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin.

Its syntax is as follows:

<jsp:action\_name attribute="value" />

|  |  |
| --- | --- |
| **Syntax** | **Purpose** |
| jsp:include | Includes a file at the time the page is requested |
| jsp:useBean | Finds or instantiates a JavaBean |
| jsp:setProperty | Sets the property of a JavaBean |
| jsp:getProperty | Inserts the property of a JavaBean into the output |
| jsp:forward | Forwards the requester to a new page |
| jsp:plugin | Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin |
| jsp:element | Defines XML elements dynamically. |
| jsp:attribute | Defines dynamically defined XML element's attribute. |
| jsp:body | Defines dynamically defined XML element's body. |
| jsp:text | Use to write template text in JSP pages and documents. |

1. **What do the id and scope attribute mean in the action elements?**
2. **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext

**Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values: (a) page, (b)request, (c)session, and (d) application.

<jsp:useBean id="sessionBean" class="beans.NameBean" scope="session" />

<jsp:setProperty name="sessionBean" property="\*" />

1. **What is the function of <jsp:include> action?**
2. The **jsp:include** action inserts additional static or dynamic resources into the page at request time as the page is displayed. Specify the resource with a relative URL (either page-relative or application-relative).

You must set flush to **true**, which results in the buffer being **flushed** to the browser when a **jsp:include** action is executed. (The flush attribute is mandatory, but a setting of false is currently invalid.)

You can also have an action body with jsp:param settings, as shown in the second example.

Examples:

<jsp:include page="/templates/userinfopage.jsp" flush="true" />

or:

<jsp:include page="/templates/userinfopage.jsp" flush="true" >

<jsp:param name="username" value="Smith" />

<jsp:param name="userempno" value="9876" />

</jsp:include>

**Note:**

1. The following syntax would work as an alternative to the preceding example:

<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876" flush="true" />

1. The jsp:include action can be used only within the same servlet context.

[**https://docs.oracle.com/cd/A97336\_01/buslog.102/a83726/genlovw3.htm**](https://docs.oracle.com/cd/A97336_01/buslog.102/a83726/genlovw3.htm)

1. **What is the difference between include action and include directive?**
2. Unlike the include directive, which inserts the file at the time the JSP page is translated into a servlet, include action inserts the file at the time the page is requested.

**Include Action**

<jsp:include page="relative URL" flush="true" />

**Include directive**

**Basic syntax:**

<%@ include file="relative url" >

**XML equivalent**

<jsp:directive.include file="relative url" />

1. **What are the options in JSP to include files?**
2. In JSP, we can perform inclusion in the following ways:

* **By include directive:** For example:

<%@ include file=”header.jsp” %>

* **By include action:** For example:

<%@ include file=”header.jsp” %>

* **By using pageContext implicit object** For example:

<%

pageContext.include(“/header.jsp”);

%>

* **By using RequestDispatcher object:** For example:

<%

RequestDispatcher rd = request.getRequestDispatcher(“/header.jsp”);

Rd.incliude(request,response);

%>

1. **What is <jsp:useBean> action?**
2. The useBean action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.

The simplest way to load a bean is as follows:

<jsp:useBean id="name" class="package.class" />

<jsp:useBean id="sessionBean" class="beans.NameBean" scope="session" />

<jsp:setProperty name="sessionBean" property="\*" />

1. We have a bean class Details where we are having three variables username, age and password. In order to use the bean class and it’s properties in JSP we have initialized the class like this in the userdetails.jsp page –

<jsp:useBean id="userinfo" class="beginnersbook.com.Details"></jsp:useBean>

We have used useBean action to initialize the class. Our class is in beginnersbook.com package so we have given a fully qualified name**beginnersbook.com.Details**.

1. We have mapped the properties of bean class and JSP using setProperty action tag. We have given ‘\*’ in the property field to map the values based on their names because we have used the same property name in bean class and index.jsp JSP page. In the name field we have given the unique identifier which we have defined in useBean tag.

<jsp:setProperty property="\*" name="userinfo"/>

1. To get the property values we have used getProperty action tag.

<jsp:getProperty property="propertyname" name="userinfo"/>

Details.java

package beginnersbook.com;

public class Details {

public Details() {

}

private String username;

private int age;

private String password;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

index.jsp

<html>

<head><title>

useBean, getProperty and setProperty example

</title></head>

<form action="userdetails.jsp" method="post">

User Name: <input type="text" name="username"><br>

User Password: <input type="password" name="password"><br>

User Age: <input type="text" name="age"><br>

<input type="submit" value="register">

</form>

</html>

userdetails.jsp

<jsp:useBean id="userinfo" class="beginnersbook.com.Details"></jsp:useBean>

<jsp:setProperty property="\*" name="userinfo"/>

You have enterted below details:<br>

<jsp:getProperty property="username" name="userinfo"/><br>

<jsp:getProperty property="password" name="userinfo"/><br>

<jsp:getProperty property="age" name="userinfo" /><br>

[**http://beginnersbook.com/2013/11/jsp-usebean-setproperty-getproperty-action-tags/**](http://beginnersbook.com/2013/11/jsp-usebean-setproperty-getproperty-action-tags/)

1. **What is <jsp:forward> Action?**
2. The jsp:forward action effectively terminates execution of the current page, discards its output, and dispatches a new page--either an HTML page, a JSP page, or a servlet.

The JSP page must be buffered (you cannot set buffer="none") to use a jsp:forward action. The action will clear the buffer (not outputting contents to the browser).

As with jsp:include, you can also have an action body with jsp:param settings, as shown in the second example.

Examples:

<jsp:forward page="/templates/userinfopage.jsp" />

or:

<jsp:forward page="/templates/userinfopage.jsp" >

<jsp:param name="username" value="Smith" />

<jsp:param name="userempno" value="9876" />

</jsp:forward>

**Notes:**

* The difference between the jsp:forward examples here and the jsp:include examples earlier is that the jsp:include examples insert userinfopage.jsp within the output of the current page; the jsp:forward examples stop executing the current page and display userinfopage.jsp instead.
* The jsp:forward action can be used only within the same servlet context.

<https://docs.oracle.com/cd/A97336_01/buslog.102/a83726/genlovw3.htm>

1. **What is <jsp:plugin> Action?**
2. The jsp:plugin action results in the execution of a specified applet or JavaBean in the client browser, preceded by a download of Java plugin software if necessary.

Specify configuration information, such as the applet to run and the codebase, using jsp:plugin attributes. The JSP container might provide a default URL for the download, but you can also specify attributenspluginurl="*url*" (for a Netscape browser) or iepluginurl="*url*" (for an Internet Explorer browser).

Use nested jsp:param actions within <jsp:params> and </jsp:params> start and end tags to specify parameters to the applet or JavaBean. (Note that these jsp:params start and end tags are *not* necessary when usingjsp:param in a jsp:include or jsp:forward action.)

Use <jsp:fallback> and </jsp:fallback> start and end tags to delimit alternative text to execute if the plugin cannot run.

The following example, from the *Sun Microsystems JavaServer Pages Specification, Version 1.1*, shows use of an applet plugin:

<jsp:plugin type=applet code="Molecule.class" codebase="/html" >

<jsp:params>

<jsp:param name="molecule" value="molecules/benzene.mol" />

</jsp:params>

<jsp:fallback>

<p> Unable to start the plugin. </p>

</jsp:fallback>

</jsp:plugin>

Many additional parameters--such as ARCHIVE, HEIGHT, NAME, TITLE, and WIDTH--are allowed in the jsp:plugin action statement as well. Use of these parameters is according to the general HTML specification.

XS:Scope

1. **What are the different scope values for the JSP action?**
2. The scope attribute identifies the lifecycle of the Action element. It has four possible values:

* page
* request
* session
* application

1. **What are JSP implicit objects?**
2. JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

Implicit objects are supported by JSP:

|  |  |  |
| --- | --- | --- |
| **Objects** | **Classes** | **Comments** |
| out | javax.servlet.jsp.JspWriter | This is used for writing content to the client (browser). It has several methods which can be used for properly formatting output message to the browser and for dealing with the buffer. |
| request | javax.servlet.http.HttpServletRequest | The main purpose of request implicit object is to get the data on a JSP page which has been entered by user on the previous JSP page. While dealing with login and signup forms in JSP we often prompts user to fill in those details, this object is then used to get those entered details on an another JSP page (action page) for validation and other purposes. |
| response | javax.servlet.http.HttpServletResponse | It is basically used for modfying or delaing with the response which is being sent to the client(browser) after processing the request. |
| session | javax.servlet.http.HttpSession | It is most frequently used implicit object, which is used for storing the user’s data to make it available on other JSP pages till the user session is active. |
| application | javax.servlet.ServletContext | This is used for getting application-wide initialization parameters and to maintain useful data across whole JSP application. |
| exception | javax.servlet.jsp.JspException | Exception implicit object is used in exception handling for displaying the error messages. This object is only available to the JSP pages, which has isErrorPage set to true. |
| page | java.lang.Object | Page implicit object is a reference to the current Servlet instance (Converted Servlet, generated during translation phase from a JSP page). We can simply use this in place of it. I’m not covering it in detail as it is rarely used and not a useful implicit object while building a JSP application. |
| pageContext | javax.servlet.jsp.PageContext | It is used for accessing page, request, application and session attributes. |
| config | javax.servlet.ServletConfig | This is a Servlet configuration object and mainly used for accessing getting configuration information such as servlet context, servlet name, configuration parameters etc. |

<http://beginnersbook.com/2013/11/jsp-implicit-objects/>

1. **What is the page directive is used to prevent a JSP page from automatically creating a session?**
2. <%@ page session="false">
3. **How can you read request header information?**
4. Using getHeaderNames() method of HttpServletRequest to read the HTTP header infromation. This method returns an Enumeration that contains the header information associated with the current HTTP request.
5. **What is the difference between JspWriter and PrintWriter?**
6. The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions.
7. **How to read form data using JSP?**
8. JSP handles form data parsing automatically using the following methods depending on the situation:

* **getParameter():** You call request.getParameter() method to get the value of a form parameter.
* **getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
* **getParameterNames():** Call this method if you want a complete list of all parameters in the current request.
* **getInputStream():** Call this method to read binary data stream coming from the client.

1. **What are filters?**
2. Servlet and JSP Filters are Java classes that can be used in Servlet and JSP Programming for the following purposes:

* To intercept requests from a client before they access a resource at back end.
* To manipulate responses from server before they are sent back to the client.

There are various types of filters suggested by the specifications:

* Authentication Filters.
* Data compression Filters
* Encryption Filters .
* Filters that trigger resource access events.
* Image Conversion Filters .
* Logging and Auditing Filters.
* MIME-TYPE Chain Filters.
* Tokenizing Filters .
* XSL/T Filters That Transform XML Content.

Filters are deployed in the deployment descriptor file web.xml and then map to either servlet or JSP names or URL patterns in your application's deployment descriptor. The deployment descriptor file web.xml can be found in <Tomcat-installation-directory>\conf directory.

When the JSP container starts up your web application, it creates an instance of each filter that you have declared in the deployment descriptor. The filters execute in the order that they are declared in the deployment descriptor.

<http://www.tutorialspoint.com/jsp/jsp_writing_filters.htm>

1. **What are cookies?**
2. Cookies are text files stored on the client computer and they are kept for various information tracking purpose.
3. **How cookies work?**
4. Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser).If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server.
5. **How do you set cookies in the JSP?**
6. Setting cookies with JSP involves three steps:

* **Creating a Cookie object:** You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.
* **Setting the maximum age:** You use setMaxAge to specify how long (in seconds) the cookie should be valid.
* **Sending the Cookie into the HTTP response headers:** You use response.addCookie to add cookies in the HTTP response header

1. **How to read cookies with JSP?**
2. To read cookies, you need to create an array of javax.servlet.http.Cookie objects by calling the getCookies( ) method of HttpServletRequest. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.
3. **How to delete cookies with JSP?**
4. To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:

* Read an already existing cookie and store it in Cookie object.
* Set cookie age as zero using **setMaxAge()** method to delete an existing cookie.
* Add this cookie back into response header.

1. **How is Session Management done in JSP?**
2. Session management can be achieved by the use of:

* **Cookies:** A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.
* **Hidden Form Fields:** A web server can send a hidden HTML form field along with a unique session ID as follows:

<input type="hidden" name="sessionid" value="12345">

This implies that when the form is submitted, the specified name and value will be getting included in GET or POST method.

* **URL Rewriting:** In URL rewriting some extra information is added on the end of each URL that identifies the session. This URL rewriting can be useful where a cookie is disabled.
* **The session Object:** JSP makes use of servlet provided HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

1. **How can you delete a session data?**
2. When you are done with a user's session data, you have several options:

* **Remove a particular attribute:** You can call *public void removeAttribute(String name)* method to delete the value associated with a particular key.
* **Delete the whole session:** You can call *public void invalidate()*method to discard an entire session.
* **Setting Session timeout:** You can call *public void setMaxInactiveInterval(int interval)* method to set the timeout for a session individually.
* **Log the user out:** The servers that support servlets 2.4, you can call**logout** to log the client out of the Web server and invalidate all sessions belonging to all the users.
* **web.xml Configuration:** If you are using Tomcat, apart from the above mentioned methods, you can configure session time out in web.xml file as follows.

1. **How can you upload a file using JSP?**
2. To upload a single file you should use a single <input .../> tag with attribute type="file".To allow multiple files uploading, include more than one input tags with different values for the name attribute.
3. **Where will be the uploaded files stored?**
4. You can hard code this in your program or this directory name could also be added using an external configuration such as a context-param element in web.xml.
5. **What is JSP page redirection?**
6. Page redirection is generally used when a document moves to a new location and we need to send the client to this new location or may be because of load balancing, or for simple randomization.
7. **What is the difference between <jsp:forward page = ... > and response.sendRedirect(url)?**
8. The <jsp:forward> element forwards the request object containing the client request information from one JSP file to another file. The target file can be an HTML file, another JSP file, or a servlet, as long as it is in the same application context as the forwarding JSP file.

sendRedirect sends HTTP temporary redirect response to the browser, and browser creates a new request to go the redirected page.

|  |  |
| --- | --- |
| **Forward()** | **SendRediret()** |
| When we use forward method request is transfer to other resource within the same server for further processing. | In case of sendRedirect request is transfer to another resource to different domain or different server for further processing. |
| In case of forward Web container handle all process internally and client or browser is not involved. | When you use SendRedirect container transfers the request to client or browser so url given inside the **sendRedirect**method is visible as a new request to the client. |
| When forward is called on **request dispatcher** object we pass request and response object so our old request object is present on new resource which is going to process our request | In case of SendRedirect call old request and response object is lost because it’s treated as new request by the browser. |
| Visually we are not able to see the forwarded address, it is transparent | In address bar we are able to see the new redirected address it’s not transparent. |
| Using forward () method is faster then send redirect. | SendRedirect is slower because one extra round trip is required because completely new request is created and old request object is lost. Two browser requests required. |
| When we redirect using forward and we want to use same data in new resource we can use request.setAttribute () as we have request object available. | But in sendRedirect if we want to use we have to store the data in session or pass along with the URL. |

<http://javarevisited.blogspot.com/2011/09/sendredirect-forward-jsp-servlet.html#ixzz3qIWpevzv>

1. **What is a hit count for a web page?**
2. A hit counter tells you about the number of visits on a particular page of your web site.
3. **How do you impement hit counter in JSP?**
4. To implement a hit counter you can make use of Application Implicit object and associated methods getAttribute() and setAttribute().

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method.

1. **How can you implement hit counter to avoid loss of count data with each restart of the application?**
2. You can follow the below steps:

* Define a database table with a single count, let us say hitcount. Assign a zero value to it.
* With every hit, read the table to get the value of hitcount.
* Increase the value of hitcount by one and update the table with new value.
* Display new value of hitcount as total page hit counts.
* If you want to count hits for all the pages, implement above logic for all the pages.

1. **What is auto refresh feature?**
2. Consider a webpage which is displaying live game score or stock market status or currency exchange rate. For all such type of pages, you would need to refresh your web page regularly using refresh or reload button with your browser.

JSP makes this job easy by providing you a mechanism where you can make a webpage in such a way that it would refresh automatically after a given interval.

1. **How do you implement the auto refresh in JSP?**
2. The simplest way of refreshing a web page is using method setIntHeader() of response object. Following is the signature of this method:

public void setIntHeader(String header, int headerValue)

This method sends back header "Refresh" to the browser along with an integer value which indicates time interval in seconds.

1. **What is a JSP custom tag?**

A custom tag is a user-defined JSP language element. When a JSP page containing a custom tag is translated into a servlet, the tag is converted to operations on an object called a tag handler. The Web container then invokes those operations when the JSP page's servlet is executed.

1. **What is JSP Expression Language?**

JSP Expression Language (EL) makes it possible to easily access application data stored in JavaBeans components. JSP EL allows you to create expressions both (a) arithmetic and (b) logical. A simple syntax for JSP EL is :

## Simple Syntax:

Typically, when you specify an attribute value in a JSP tag, you simply use a string. For example:

<jsp:setProperty name="box" property="perimeter" value="100"/>

JSP EL allows you to specify an expression for any of these attribute values. A simple syntax for JSP EL is as follows:

${expr}

Here expr specifies the expression itself. The most common operators in JSP EL are . and []. These two operators allow you to access various attributes of Java Beans and built-in JSP objects.

For example above syntax <jsp:setProperty> tag can be written with an expression like:

<jsp:setProperty name="box" property="perimeter"

value="${2\*box.width+2\*box.height}"/>

When the JSP compiler sees the ${} form in an attribute, it generates code to evaluate the expression and substitues the value of expresson.

You can also use JSP EL expressions within template text for a tag. For example, the <jsp:text> tag simply inserts its content within the body of a JSP. The following <jsp:text> declaration inserts <h1>Hello JSP!</h1> into the JSP output:

<jsp:text>

<h1>Hello JSP!</h1>

</jsp:text>

You can include a JSP EL expression in the body of a <jsp:text> tag (or any other tag) with the same ${} syntax you use for attributes. For example:

<jsp:text>

Box Perimeter is: ${2\*box.width + 2\*box.height}

</jsp:text>

EL expressions can use parentheses to group subexpressions. For example, ${(1 + 2) \* 3} equals 9, but ${1 + (2 \* 3)} equals 7.

1. **What are the implicit EL objects in JSP ?**

The JSP expression language supports the following implicit objects:

* **pageScope:** Scoped variables from page scope
* **requestScope:** Scoped variables from request scope
* **sessionScope:** Scoped variables from session scope
* **applicationScope:** Scoped variables from application scope
* **param:** Request parameters as strings
* **paramValues:** Request parameters as collections of strings
* **headerHTTP:** request headers as strings
* **headerValues:** HTTP request headers as collections of strings
* **initParam:** Context-initialization parameters
* **cookie:** Cookie values
* **pageContext:** The JSP PageContext object for the current page

1. **How can we disable EL ?**

We can disable using isELIgnored attribute of the page directive:

<%@ page isELIgnored ="true|false" %>

If it is true, EL expressions are ignored when they appear in static text or tag attributes. If it is false, EL expressions are evaluated by the container.

XT: TAGS

1. **What is JSTL?**
2. The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags.

1. **What the different types of JSTL tags are?**
2. Types of JSTL tags are:

<http://beginnersbook.com/2013/11/jstl-curl-core-tag/>

* **Core Tags**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tag** | **Description** | **Example** | **Result** |
| <c:out > | Like <%= ... >, but for expressions. | <c:out value="${'<b>This is a <c:out> example </b>'}"/> | <b>This is a <c:out> example </b> |
|  |  | <c:out value="${'<b>This is a <c:out> example </b>'}" escapeXml="false"/> | This is a example |
|  |  | <%! String str = null; %> <c:out value="${str}" default="default value of c:out"/> | default value of c:out |
| <c:set > | Sets the result of an expression evaluation in a 'scope'  Source: http://javarevisited.blogspot.com.au/2012/02/jstl-tag-examples-in-jsp-java-j2ee.html | <c:set var="currency" value="USD" /> | Above example of <c:set> will create an attribute named "currency" with value "USD" in default scope which is page scope |
|  |  | <c:set var="currency" value="USD" scope="session" /> | Above example of <c:set> will create an attribute named "currency" with value "USD" in session scope which is page scope |
|  |  | <c:set var="currency" scope="session" >  USD,EUR,AUD,INR </c:set> | Another variance of jstl <c:set> tag is that you can supply value in body instead of on attribute line. some time when value is long, giving it on body makes code more readable. here is example of supplying value on jstl set tag body: |
|  |  | <c:set var="currency" value="${user.currency}" /> | Keep in mind that <c:set> can also remove a variable or attribute if value resolves to "null" during runtime.   e.g. in below example<c: set> tag of jstl will remove "currency" attribute from any scope if EL expression ${user.currency} will resolve to null. |
|  |  | <% HashMap<String, String> currencyMap= new HashMap<String, String>();  pageContext.setAttribute("currencyMap", currencyMap);  %>  <c:set target="${currencyMap}" property="${'USA'}" value="${'USD'}" />  <c:out value="${currencyMap.isEmpty()}" />  <c:out value="${currencyMap['USA']}" /> | <c:set> can also be used to set bean properties or map value. in that case instead of "var" we need to use "target" and "property" which will define bean and property name to be set. if "target" is map than "property" is name of key and "value" is value for that key. What is worth noting here is that <c:set target=""> must point to a real object and not the name of object as it was with <jsp:useBean> action. if "target" doesn't resolve into object than web container will throw exception. here is example of setting bean property using JSTL <c:set> tag in JSP:  Result: false USD |
| <c:remove > | Removes a scoped variable (from a particular scope, if specified). | <c:remove var="author"/> | This above code removes an attribute from all the scopes (page, session, application, request) |
|  |  | <c:remove var="author" scope="session"/> | In order to be specific we must need to specify the scope attribute inside <c:remove> tag |
| <c:catch> | Catches any Throwable that occurs in its body and optionally exposes it. | <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %> <html> <head> <title>JSTL c:catch Core Tag Example</title> </head> <body> <%!  int num1=10; int num2=0; %> <c:catch var ="errormsg">  <% int res = num1/num2;  out.println(res);%> </c:catch> <c:if test = "${errormsg != null}">  <p>There has been an exception raised in the above  arithmetic operation. Please fix the error.  Exception is: ${errormsg}</p> </c:if> </body> </html> | Result: There has been an exception raised in the above arithmetic operation. Please fix the error. Exception is: java.lang.ArithmeticException: / by zero |
| <c:if> | Simple conditional tag which evalutes its body if the supplied condition is true. | <c:set var="age" value="26"/> <c:if test="${age >= 18}">  <c:out value="You are eligible for voting!"/> </c:if> <c:if test="${age < 18}">  <c:out value="You are not eligible for voting!"/> </c:if> | You are eligible for voting! |
|  |  | <c:if test="${17 >= 18}" var="res" scope="request"> </c:if> <c:out value="${requestScope.res}"/> | There are two other optional attributes for this tag which are var and scope. Using these attributes you can simply store the test results in a variable within a specified scope.  Storing the test result in variable res in request scope. For printing the value we have given requestScope.res as the variable is stored in request however you can even give variable name(res) alone, it would work fine.  Result: false |
| <c:choose> | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> | <c:set var="number1" value="${222}"/> <c:set var="number2" value="${12}"/> <c:set var="number3" value="${10}"/> <c:choose>  <c:when test="${number1 < number2}">  ${"number1 is less than number2"}  </c:when>  <c:when test="${number1 <= number3}">  ${"number1 is less than equal to number2"}  </c:when>  <c:otherwise>  ${"number1 is largest number!"}  </c:otherwise> </c:choose> | Result: number1 is largest number! |
| <c:when> | Subtag of <choose> that includes its body if its condition evalutes to 'true'. |
| <c:otherwise > | Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'. |
| <c:import> | Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'. | Syntax: **<c:import var="variable\_name" url="relative\_url"/>**  Here variable\_name is a variable which stores the data imported from another url. relative\_url is the address of the file/page which needs to be imported.  Attributes of <c:import> **url:** It’s mandatory attribute and needs to be mentioned always.  **var:** It is an optional attribute if this is not specified then the imported data will be printed on the current page. For e.g. the statement <c:import url=”/file.jsp” /> would print the data of file.jsp on the client (browser).  **scope:** It is also optional. If we are using var attribute then scope can be used along with it to specify the scope of the data stored in the variable. |  |
| <c:forEach > | <c:forEach> tag in JSTL is used for executing the same set of statements for a finite number of times. It’s similar to the for loop in java. This is basically used when we need to perform(execute) set of statements again and again for a specified number of times.  The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality . | <c:forEach> Tag  Syntax of <c:forEach> <c:forEach var="counter\_variable\_name" begin="intial\_value" end="final\_limit">  //Block of statements </c:forEach> The below are the three main attributes of <c:forEach> tag.  **begin:** The initial counter value. **end:** The final limit till which the loop will execute **var:** Counter variable name  example <c:forEach var="counter" begin="1" end="10">  <c:out value="${counter}"/> </c:forEach> | Result:  1 2 3 4 5 6 7 8 9 10 |
| <c:forTokens> | <c:forTokens> is also used for iteration but it only works with delimiter which means using this tag we can break the input data into multiple parts based on the delimiter. We will understand this with the help of an example in this post.  Iterates over tokens, separated by the supplied delimeters. | <c:forTokens> tag <c:forTokens items="value(s)" delims="delimiter" var="variable\_name">  //Set of statements </c:forTokens> The below are the three main attributes of <c:forTokens> tag.  **items:** Set of data value(s). **delims:** The delimiter can have any value. It can be a number, string or special character. **var:** variable name which stores the sub strings.  Example In this example we are splitting the strings into multiple substrings using delimter dot(‘.’).  <c:forTokens items="www.beginnersbook.com" delims="." var="site">  <c:out value="${site}"/> </c:forTokens> | Result:  www beginnersbook com |
| <c:param> | <c:param> JSTL tag is mostly used with <c:url> and <c:redirect> tags. Basically it adds parameter and their values to the output of these tags. | Syntax: **<c:param name="parameter\_name" value="parameter\_value"/> Attributes of <c:param> tag name:** To specify the name of the parameter. **value:** To specify the value of the parameter. **Example of <c:param>** <c:url value="/mypage.jsp" var="completeURL">  <c:param name="Id" value="736"/>  <c:param name="user" value="chaitanya"/> </c:url> ${completeURL} | **Result:** /helloWorldStruts2/mypage.jsp?Id=736&user=chaitanya |
|  |  | **<c:redirect url="/display.jsp" >  <c:param name="UserId" value="222"/>  <c:param name="UserName" value="ChaitanyaSingh"/> </c:redirect> ${completeURL}  display.jsp  USER ID IS: ${param.UserId} USER NAME IS: ${param.UserName}** | USER ID IS: 222USER NAME IS: ChaitanyaSingh |
| <c:redirect > | <c:redirect> is used for redirecting the current page to another URL. | **Syntax:** <c:redirect url="http://www.anydomainhere.com/samplepage.jsp"/>  **Example:**  <c:set var="myurl" value="2" scope="request"/>  <c:if test="${myurl<1}">  <c:redirect url="http://beginnersbook.com"/>  </c:if>  <c:if test="${myurl>1}">  <c:redirect url="http://www.google.com"/>  </c:if> | Result: Since the value of the variable myurl is 2 the page gets directed to the http://www.google.com. |
| <c:url> | <c:url> JSTL tag is used for url formatting or you can say url encoding. This is mainly used when we need to open a JSP page based on the user input or based on the value of a variable. It basically converts a relative url into a application context’s url. | Syntax: Basic syntax looks like this – The attribute “value” is a required attribute for the <c:url> tag  <c:url value="/file1.jsp" /> There are three other optional attributes exist for this tag which are as follows –  var: Variable name to store the formatted url (resultant url). context: Used for specifying the application (or project name). Don’t get it? We will see this with the help of an example later. scope: The scope in which the var attribute would be stored. It can be request, page, application or session. |  |
|  |  | Example 1: value attribute of <c:url>  <c:url value="/file1.jsp"/> | The above code produced below output. Note: BeginnersBook is my project name (in other words JSP application name).  /BeginnersBook/file1.jsp |
|  |  | Example 2: var attribute of <c:url> tag Let’s modify the example 1 like this. We have added a variable myurl in <c:url>. Now the result of <c:url> would be stored in variable myurl.  <c:url var="myurl" value="/file1.jsp"/> ${myurl} | /BeginnersBook/file1.jsp |
|  |  | Example 3: context attribute By default this tag takes the current application as the context, however we can explicitly specify the context in c url as shown in the below example –  Note: The value of the context should always starts with “/” otherwise you will get the below exception message – HTTP Status 500 – javax.servlet.ServletException: javax.servlet.jsp.JspTagException: In URL tags, when the “context” attribute is specified, values of both “context” and “url” must start with “/”.  <c:url var="myurl" value="/file1.jsp" context="/MyJSPProject"/> ${myurl} | /MyJSPProject/file1.jsp |
|  |  | Example 4: scope attribute <c:url var="myurl" value="/file1.jsp" context="/MyJSPProject" scope="session"/> ${requestScope.myurl} Output: Output screen will be blank as we have stored the myurl in session scope and we are trying to print the value after fetching from requestScope.  Correct usage: This is how it should be coded. Here we have stored in session and fetching from sessionScope so it would work fine.  <c:url var="myurl" value="/file1.jsp" context="/MyJSPProject" scope="session"/> ${sessionScope.myurl} | /MyJSPProject/file1.jsp |

* **Formatting tags**

|  |  |
| --- | --- |
| Tag | Description |
| <fmt:formatNumber> | To render numerical value with specific precision or format. |
| <fmt:parseNumber> | Parses the string representation of a number, currency, or percentage. |
| <fmt:formatDate> | Formats a date and/or time using the supplied styles and pattern |
| <fmt:parseDate> | Parses the string representation of a date and/or time |
| <fmt:bundle> | Loads a resource bundle to be used by its tag body. |
| <fmt:setLocale> | Stores the given locale in the locale configuration variable. |
| <fmt:setBundle> | Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable. |
| <fmt:timeZone> | Specifies the time zone for any time formatting or parsing actions nested in its body. |
| <fmt:setTimeZone> | Stores the given time zone in the time zone configuration variable |
| <fmt:message> | To display an internationalized message. |
| <fmt:requestEncoding> | Sets the request character encoding |

* **SQL tags**

|  |  |
| --- | --- |
| Tag | Description |
| <sql:setDataSource> | Creates a simple DataSource suitable only for prototyping |
| <sql:query> | Executes the SQL query defined in its body or through the sql attribute. |
| <sql:update> | Executes the SQL update defined in its body or through the sql attribute. |
| <sql:param> | Sets a parameter in an SQL statement to the specified value. |
| <sql:dateParam> | Sets a parameter in an SQL statement to the specified java.util.Date value. |
| <sql:transaction > | Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction. |

* **XML tags**

|  |  |
| --- | --- |
| Tag | Description |
| <x:out> | Like <%= ... >, but for XPath expressions. |
| <x:parse> | Use to parse XML data specified either via an attribute or in the tag body. |
| <x:set > | Sets a variable to the value of an XPath expression. |
| <x:if > | Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored. |
| <x:forEach> | To loop over nodes in an XML document. |
| <x:choose> | Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise> |
| <x:when > | Subtag of <choose> that includes its body if its expression evalutes to 'true' |
| <x:otherwise > | Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false' |
| <x:transform > | Applies an XSL transformation on a XML document |
| <x:param > | Use along with the transform tag to set a parameter in the XSLT stylesheet |

* **JSTL Functions**

|  |  |
| --- | --- |
| Function | Description |
| fn:contains() | Tests if an input string contains the specified substring. |
| fn:containsIgnoreCase() | Tests if an input string contains the specified substring in a case insensitive way. |
| fn:endsWith() | Tests if an input string ends with the specified suffix. |
| fn:escapeXml() | Escapes characters that could be interpreted as XML markup. |
| fn:indexOf() | Returns the index withing a string of the first occurrence of a specified substring. |
| fn:join() | Joins all elements of an array into a string. |
| fn:length() | Returns the number of items in a collection, or the number of characters in a string. |
| fn:replace() | Returns a string resulting from replacing in an input string all occurrences with a given string. |
| fn:split() | Splits a string into an array of substrings. |
| fn:startsWith() | Tests if an input string starts with the specified prefix. |
| fn:substring() | Returns a subset of a string. |
| fn:substringAfter() | Returns a subset of a string following a specific substring. |
| fn:substringBefore() | Returns a subset of a string before a specific substring. |
| fn:toLowerCase() | Converts all of the characters of a string to lower case. |
| fn:toUpperCase() | Converts all of the characters of a string to upper case. |
| fn:trim() | Removes white spaces from both ends of a string. |

1. **How to pass information from JSP to included JSP?**

Using <%jsp:param> tag.

1. **What are JSTL formatting tags ?**

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites. Following is the syntax to include Formatting library in your JSP:

<%@ taglib prefix="fmt"

uri="http://java.sun.com/jsp/jstl/fmt" %>

1. **What are JSTL SQL tags?**

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server.

Following is the syntax to include JSTL SQL library in your JSP:

<%@ taglib prefix="sql"

uri="http://java.sun.com/jsp/jstl/sql" %>

1. **What are JSTL XML tags?**

The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in your JSP.

<%@ taglib prefix="x"

uri="http://java.sun.com/jsp/jstl/xml" %>

1. **What type of errors you might encounter in a JSP code?**

* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compliation.
* **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

1. **In JSP page how can we handle runtime exception?**

We can use the errorPage attribute of the page directive to have uncaught run-time exceptions automatically forwarded to an error processing page.

Example: <%@ page errorPage="error.jsp" %>

It will redirect the browser to the JSP page error.jsp if an uncaught exception is encountered during request processing. Within error.jsp, will have to indicate that it is an error-processing page, using the directive: <%@ page isErrorPage="true" %>

Following is an example to specifiy an error page for a main.jsp. To set up an error page, use the <%@ page errorPage="xxx" %> directive.

<%@ page errorPage="ShowError.jsp" %>

<html>

<head>

<title>Error Handling Example</title>

</head>

<body>

<%

// Throw an exception to invoke the error page

int x = 1;

if (x == 1)

{

throw new RuntimeException("Error condition!!!");

}

%>

</body>

</html>

Now you would have to write one Error Handling JSP ShowError.jsp, which is given below. Notice that the error-handling page includes the directive <%@ page isErrorPage="true" %>. This directive causes the JSP compiler to generate the exception instance variable.

<%@ page isErrorPage="true" %>

<html>

<head>

<title>Show Error Page</title>

</head>

<body>

<h1>Opps...</h1>

<p>Sorry, an error occurred.</p>

<p>Here is the exception stack trace: </p>

<pre>

<% exception.printStackTrace(response.getWriter()); %>

</pre>

</body>

</html>

1. **What is Internationalization, Internationalization**  and **?**

The development of multilingual software currently goes through two phases: the first phase is internationalization, and the second phase is localization.

**Internationalization** is the process of designing a software application so that it can potentially be adapted to various languages and regions(locale) without engineering changes.

Internationalization is often written i18n, where 18 is the number of letters between i and n in the English word.

**Localization** is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. For example Hindi translation to a web site.

Localization is sometimes written as l10n, where 10 is the number of letters between l and n.

**Also**, Localization (which is potentially performed multiple times, for different locales) uses the infrastructure or flexibility provided by internationalization (which is ideally performed only once, or as an integral part of ongoing development)

**Locale** This is a particular cultural or geographical region. It is usually referred to as a language symbol followed by a country symbol which are separated by an underscore. For example "en\_US" represents english locale for US.

1. **What is difference between <%-- comment --%> and <!-- comment -->?**
2. <%-- comment --%> is JSP comment and is ignored by the JSP engine.

<!-- comment --> is an HTML comment and is ignored by the browser.

1. **Is JSP technology extensible?**
   1. YES. JSP technology is extensible through the development of custom actions, or tags, which are encapsulated in tag libraries.
2. **Can a JSP page process HTML FORM data?**
   1. Yes. However, unlike Servlet, you are not required to implement HTTP-protocol specific methods like doGet() or doPost() within your JSP page. You can obtain the data for the FORM input elements via the request implicit object within a scriptlet or expression.
3. **How do you pass control from one JSP page to another?**
4. Use the following ways to pass control of a request from one servlet to another or one jsp to another:

* The RequestDispatcher object ‘s forward method to pass the control.
* Using the *response.sendRedirect* method

**Request Dispatcher in JSP**

**Explanation**

Request Dispatcher is used to forward a request to another page.

**Example:example.jsp**

<form action="process.jsp" method="post">  
<input type="text" name="name">  
<input type="password" name="pass">  
<input type=submit>  
</form>

**process.jsp**

<%@ page contentType="text/html"%>  
<%@ page import = "javax.servlet.RequestDispatcher" %>  
<%  
String message = "HI Welcome";  
RequestDispatcher rd = request.getRequestDispatcher("index.jsp");  
request.setAttribute("msg",message);  
rd.forward(request, response);  
%>

**index.jsp**

<%  
String message = (String) request.getAttribute("msg");  
out.println(" "+ message); %>

In the above example the request from the "example.jsp" is forwarded to "process.jsp". Using the "request dispatcher" function, an attribute "msg" is sent from process.jsp page to index.jsp. When the process.jsp page starts to execute, the message will be transferred from index.jsp page to process.jsp, using the "get attribute function". Finally, the message is displayed in the process.jsp page.

**response.sendRedirect in JSP**

In the below example we are receiving id and password from login page and then we are matching them with hardcoded correct id/pass. If the credentials are correct the sign-in page redirects to success page else it redirects to sign-in fail JSP page.

index.html

<html>

<head>

<title>Login Page</title>

</head>

<body>

<form action="checkdetails.jsp">

UserId: <input type="text" name="id" /> <br><br>

Password: <input type="text" name="pass" /> <br><br>

<input type="submit" value="Sign In!!"/>

</form>

</body>

</html>

This JSP page verifies the input id/pass against hard-coded values.

checkdetails.jsp

<html>

<head><title>Check Credentials</title>

</head>

<body>

<%

String uid=request.getParameter("id");

String password=request.getParameter("pass");

session.setAttribute("session-uid", uid);

if(uid.equals("Chaitanya") && password.equals("BeginnersBook"))

{

response.sendRedirect("success.jsp");

}

else

{

response.sendRedirect("failed.jsp");

}

%>

</body>

</html>

This JSP page would execute if id/pass are matched to the hardcoded userid/password.

success.jsp

<html>

<head><title>Success Page</title>

</head>

<body>

<%

String data=(String)session.getAttribute("session-uid");

out.println("Welcome "+ data+"!!");

%>

</body>

</html>

The control will be redirected to this page if the credentials entered by user are wrong.

failed.jsp

<html>

<head><title>Sign-in Failed Page</title>

</head>

<body>

<%

String data2=(String)session.getAttribute("session-uid");

out.println("Hi "+ data2+". Id/Password are wrong. Please try Again.");

%>

</body>

</html>

1. **Can you make use of a ServletOutputStream object from within a JSP page?**
2. No. You are supposed to make use of only a JSPWriter object (given to you in the form of the implicit object out) for replying to clients.

A JSPWriter can be viewed as a buffered version of the stream object returned by response.getWriter(), although from an implementational perspective, it is not.

1. **Can we override the jspInit(), \_jspService() and jspDestroy() methods?**
2. We can override jspinit() and jspDestroy() methods but not \_jspService().
3. **Why is \_jspService() method starting with an '\_' while other life cycle methods do not?**
4. \_jspService() method will be written by the container hence any methods which are not to be overridden by the end user are typically written starting with an '\_'.

This is the reason why we don't override \_jspService() method in any JSP page.

1. **A JSP page, include.jsp, has an instance variable "int a", now this page is statically included in another JSP page, home.jsp, which also has an instance variable "int a" declared. What happens when the home.jsp page is requested by the client?**
   1. It causes compilation error, as two variables with same name can't be declared. This happens because, when a page is included statically, entire code of included page becomes part of the new page. at this time there are two declarations of variable 'a'. Hence compilation error.
2. **How is scripting disabled?**
3. Scripting is disabled by setting the scripting-invalid element of the deployment descriptor to true. It is a subelement of jsp-property-group. Its valid values are true and false. The syntax for disabling scripting is as follows:

<jsp-property-group>

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

<scripting-invalid>true</scripting-invalid>

</jsp-property-group>

1. **When do use application scope?**
2. If we want to make our data available to the entire application then we have to use application scope.
3. **How does JSP engines instantiate tag handler classes instances?**
4. JSP engines will always instantiate a new tag handler instance every time a tag is encountered in a JSP page. A pool of tag instances are maintained and reusing them where possible. When a tag is encountered, the JSP engine will try to find a Tag instance that is not being used and use the same and then release it.
5. **What’s the difference between JavaBeans and taglib directives?**
6. JavaBeans and taglib fundamentals were introduced for reusability. But following are the major differences between them:

* Taglibs are for generating presentation elements while JavaBeans are good for storing information and state.
* Use custom tags to implement actions and JavaBeans to present information.