**1. Introduction to JSP**

* **Overview of JSP**: What is JSP? How does it differ from Servlets?
* **JSP Lifecycle**: Translation, Compilation, Initialization, Execution, and Destruction.
* **JSP vs. Servlet**: Differences between JSP and Servlets.
* **JSP Architecture**: Client-Server architecture, deployment.

**2. JSP Basics**

* **First JSP Program**: How to create and run your first JSP page.
* **Scripting Elements**:
  + **Expressions**: <%= expression %>
  + **Scriptlets**: <% scriptlet code %>
  + **Declarations**: <%! declaration %>
* **Comments in JSP**:
  + HTML Comments: <!-- Comment -->
  + JSP Comments: <%-- JSP Comment --%>

**3. JSP Directives**

* **Page Directive**: Attributes like language, import, session, errorPage, etc.
* **Include Directive**: Include other JSP files using <%@ include %>.
* **Taglib Directive**: Importing custom tag libraries with <%@ taglib %>.

**4. JSP Implicit Objects**

* **Request**: request
* **Response**: response
* **Session**: session
* **Out**: out
* **Application**: application
* **Config**: config
* **Page**: page
* **PageContext**: pageContext
* **Exception**: exception

**5. JSP Scopes**

* **Page Scope**: Attributes available within a single JSP page.
* **Request Scope**: Attributes available for a single HTTP request.
* **Session Scope**: Attributes available throughout the session.
* **Application Scope**: Attributes available throughout the entire application.

**6. JSP Actions**

* **JSP Action Tags**:
  + **<jsp:useBean>**: Creating and using JavaBeans in JSP.
  + **<jsp:setProperty>**: Setting properties in a JavaBean.
  + **<jsp:getProperty>**: Retrieving properties from a JavaBean.
  + **<jsp:include>**: Including a static or dynamic resource.
  + **<jsp:forward>**: Forwarding requests to another resource.
  + **<jsp:param>**: Adding parameters to an include or forward action.

**7. JSP Expression Language (EL)**

* **Introduction to EL**: Simplifying the access to data stored in JavaBeans, implicit objects, etc.
* **EL Syntax**: ${expression}.
* **Operators in EL**: Arithmetic, relational, logical, and conditional operators.
* **Implicit Objects in EL**: pageContext, session, application, request, etc.

**8. JSP Standard Tag Library (JSTL)**

* **Introduction to JSTL**: Standard tags used for common tasks.
* **Core Tags**:
  + **Iteration Tags**: <c:forEach>, <c:forTokens>.
  + **Conditional Tags**: <c:if>, <c:choose>, <c:when>, <c:otherwise>.
  + **Import Tag**: <c:import>.
  + **URL Tag**: <c:url>.
  + **Out Tag**: <c:out>.
* **Formatting Tags**: Formatting numbers, dates, and messages.
* **SQL Tags**: Database access with JSTL SQL tags.
* **XML Tags**: Parsing and transforming XML.

**10. Error Handling in JSP**

* **Exception Handling**: Using try-catch blocks in JSP.
* **Error Pages**: Creating custom error pages using the isErrorPage and errorPage attributes.
* **Handling Uncaught Exceptions**: Using the <%@ page errorPage="error.jsp" %> directive.

**15. JSP Custom Tags**

* **Introduction to Custom Tags**: Defining and using custom tags in JSP.
* **Creating Custom Tags**: Using tag handler classes.
* **Tag Libraries**: Packaging custom tags into tag libraries.
* **Tag Attributes**: Passing data to custom tags.

**17. MVC Architecture in JSP**

* **Model-View-Controller (MVC) Pattern**: Introduction and how JSP fits into the MVC architecture.
* **JSP as View**: Using JSP as the View in MVC.
* **Servlet as Controller**: Using Servlets as the Controller in MVC.
* **JSP and JSTL in MVC**: Role of JSP and JSTL tags in the MVC framework.

**Introduction of JSP (JavaServer Pages)**

JavaServer Pages (JSP) is a technology used to create dynamic web applications, similar to Servlet technology. JSP can be considered an extension of Servlet technology, offering additional functionalities such as Expression Language (EL) and JavaServer Pages Standard Tag Library (JSTL). JSP pages are generally easier to maintain than Servlets because they allow for the separation of design and development concerns.

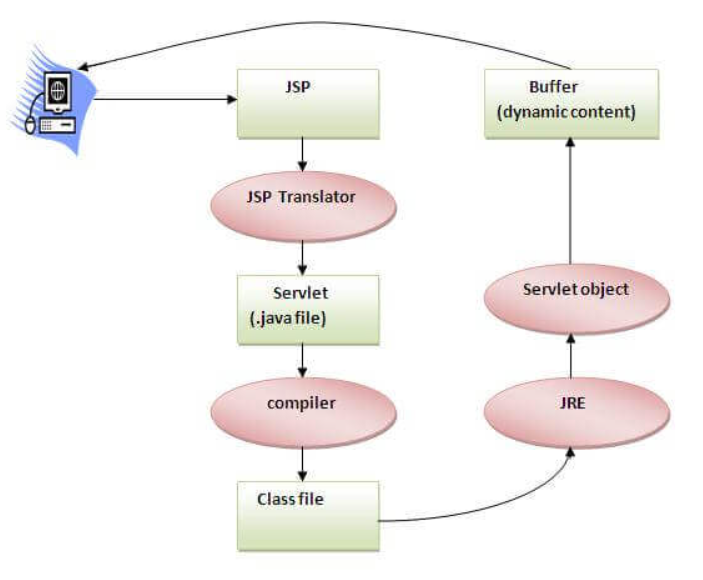
**Advantages of JSP over Servlets**

1. **Extension to Servlet**
   * **Integration**: JSP technology extends Servlet technology, inheriting all its features.
   * **Enhanced Features**: JSP allows the use of implicit objects, predefined tags, expression language (EL), and custom tags, making development more straightforward and feature-rich.
2. **Easy to Maintain**
   * **Separation of Concerns**: JSP allows separation of business logic and presentation logic, making it easier to manage and maintain compared to Servlets, which often mix both concerns.
3. **Fast Development**
   * **Dynamic Changes**: JSP pages can be modified without requiring recompilation or redeployment, unlike Servlets where changes in the look and feel necessitate recompiling the entire Servlet code.
4. **Less Code**
   * **Tag Usage**: JSP reduces code complexity through the use of tags like action tags, JSTL, custom tags, and EL, leading to cleaner and more maintainable code.

**JSP Lifecycle**

The lifecycle of a JSP page includes several phases:

1. **Translation of JSP Page**
   * The JSP page is converted into a Servlet by the JSP translator, which is part of the web server.
2. **Compilation of JSP Page**
   * The generated Servlet code is compiled into a bytecode .class file by the compiler.
3. **Classloading**
   * The compiled Servlet class is loaded into memory by the class loader.
4. **Instantiation**
   * An object of the compiled Servlet is created.
5. **Initialization**
   * The container calls the jspInit() method to initialize the JSP.
6. **Request Processing**
   * The container invokes the \_jspService() method to handle client requests and generate responses.
7. **Destroy**
   * The container calls the jspDestroy() method to clean up resources when the JSP is no longer needed.



**Conversion of JSP to Servlet**

JSP pages are converted into Servlets by the JSP translator. The process involves:

1. **Translation**: The JSP translator converts the JSP into a Servlet.
2. **Compilation**: The Servlet code is compiled into a .class file.
3. **Servlet Lifecycle**: The servlet undergoes the typical Servlet lifecycle phases, including initialization, request handling, and destruction.

**Creating a Simple JSP Page**

To create and run a simple JSP page:

1. **Write JSP Code**: Create a file named index.jsp with the following content:

<html>

<body>

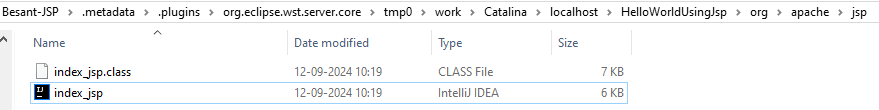
<% out.print(2\*5); %>

</body>

</html>

This code will print 10 on the browser.

1. **Save and Deploy**: Save the file with a .jsp extension. Place it in the webapps directory of your Apache Tomcat server.
2. **Start Server**: Ensure that your Apache Tomcat server is running.
3. **Access JSP Page**: Open a browser and navigate to http://localhost:portnumber/contextRoot/index.jsp.
4. For example, <http://localhost:8888/myapplication/index.jsp>.

Note: This Location you can check how JSP page converted into Servlet.

**Directory Structure for JSP**

The directory structure for JSP pages is generally similar to that of Servlets. Key points include:

* **Placement**: JSP files can be placed outside the WEB-INF folder or within any directory structure of the web application.
* **No Structure Needed**: For simple JSPs without additional components like Beans, Servlets, or TLD files, directory structure requirements are minimal.
* **When Structure is Needed**: Directory structure becomes important when using Beans, Servlets, or TLD files, which should be organized appropriately.

**Summary**

* **JSP vs Servlet**: JSP is an extension of Servlets, offering additional features and a more manageable approach to web application development.
* **Lifecycle**: JSPs go through a lifecycle of translation, compilation, classloading, instantiation, initialization, request processing, and destruction.
* **Simple JSP Creation**: Writing, saving, deploying, and accessing JSP pages is straightforward.
* **Directory Structure**: JSPs can be deployed with minimal structure, but a proper directory layout is essential for more complex applications.

### ****JSP Tag Explanation in details****

### ****1. Scriptlet Tag (****<% %>****)****

The **Scriptlet tag** in JSP is used to write Java code inside JSP pages. Anything you write inside a scriptlet tag is inserted directly into the service() method of the JSP’s servlet when the JSP page is converted into a servlet.

#### **Syntax**:

<%

// Java code here

%>

#### **Example**:

<%

int count = 10;

out.println("Count is: " + count);

%>

#### **Explanation**:

* The Java code is placed inside <% %>.
* The out implicit object is used to send output to the client.

### ****2. Expression Tag (****<%= %>****)****

The Expression tag is used to output the value of a Java expression to the client (browser). It is mainly used to display variable values or results of expressions.

#### **Syntax**:

<%= expression %>

#### **Example**:

<%

int length = 20;

int width = 10;

%>

<p>The area of the rectangle is: <%= length \* width %></p>

#### **Explanation**:

* The expression inside <%= %> is evaluated, and the result is inserted into the response stream.

### ****3. Declaration Tag (****<%! %>****)****

The Declaration tag allows you to declare variables and methods in a JSP page that are shared by all requests. The declarations are translated into instance variables or methods of the servlet class that the JSP is compiled into.

#### **Syntax**:

<%!

// Variable or method declaration

%>

#### **Example**:

<%!

int count = 0;

public int getCount() {

return ++count;

}

%>

<p>Current count is: <%= getCount() %></p>

#### **Explanation**:

* You can declare methods or variables using <%! %>.
* These declarations are shared across all requests to the JSP page.

### ****4. Directive Tags****

Directive tags provide instructions to the JSP container. They do not produce output directly but control how the JSP page is processed. There are three types of directives:

#### **1. Page Directive**

Used to define page-specific attributes, such as error handling, import statements, etc.

#### **Syntax**:

<%@ page attribute="value" %>

#### **Example**:

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

#### **Common attributes**:

* import: Imports Java classes.
* contentType: Defines the content type of the response (e.g., text/html).
* errorPage: Defines an error page.
* isErrorPage: Indicates whether the page is an error page.

#### **2. Include Directive**

Used to include a file at translation time (when the JSP is compiled).

#### **Syntax**:

<%@ include file="relativeURL" %>

#### **Example**:

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

#### **3. Taglib Directive**

Used to define and use custom tags in a JSP page.

#### **Syntax**:

<%@ taglib uri="URI" prefix="prefix" %>

#### **Example**:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

### ****5. Action Tags****

JSP Action tags are used to control the flow of a JSP page or interact with JavaBeans components and other server resources.

#### **1.** jsp:include

Includes another JSP or static resource at runtime (request time).

#### **Syntax**:

<jsp:include page="relativeURL" />

#### **Example**:

<jsp:include page="footer.jsp" />

#### **2.** jsp:forward

Forwards the request to another resource (JSP, Servlet, etc.).

#### **Syntax**:

<jsp:forward page="relativeURL" />

#### **Example**:

<jsp:forward page="nextPage.jsp" />

#### **3.** jsp:useBean

Instantiates or references a JavaBean.

#### **Syntax**:

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

#### **Example**:

<jsp:useBean id="user" class="com.example.User" />

<jsp:setProperty name="user" property="username" value="JohnDoe" />

<jsp:getProperty name="user" property="username" />

#### **Explanation**:

* jsp:useBean: Creates an instance of User bean.
* jsp:setProperty: Sets the property of the bean.
* jsp:getProperty: Retrieves the property of the bean.

#### **4.** jsp:setProperty

Used to set the value of a JavaBean property.

#### **Syntax**:

<jsp:setProperty name="beanName" property="propertyName" value="value" />

#### **Example**:

<jsp:setProperty name="user" property="username" value="John" />

#### **5.** jsp:getProperty

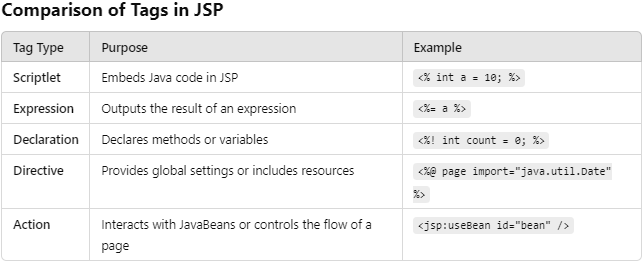
Used to get the value of a JavaBean property.

#### **Syntax**:

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

#### **Example**:

<jsp:getProperty name="user" property="username" />



**JSP Tag Practical notes in details**

### ****Scriptlet Elements****

### ****1. Scriptlet Tags(****<% %>****)****

**Description**: The scriptlet tag allows you to write Java code inside JSP pages. The code within the tag is executed when the page is requested.

#### **Practical Example**:

* **index.jsp**:

<html>

<head>

<title>Scriptlet Example</title>

</head>

<body>

<h2>Scriptlet Tag Example</h2>

<%

// Declaring a Java variable

String name = "Amarjeet Kumar Singh";

int count = 5;

// Outputting the variables in HTML

out.println("<p>Hello, " + name + "</p>");

out.println("<p>Count: " + count + "</p>");

// Iterating using a loop

for(int i = 1; i <= count; i++) {

out.println("<p>Iteration " + i + "</p>");

}

%>

</body>

</html>

**Explanation**:

* Inside <% %>, you can write any Java code like variable declarations, loops, and conditional statements.
* The output is directly inserted into the HTML response using the out object, which is automatically available in JSP.

### 2. ****Expression Tag (****<%= %>****)****

**Description**: The **expression tag** is mainly used to **print the values of a variable or method**. It evaluates a Java expression and converts the result into a string, which is then inserted directly into the output (HTML page).

#### **Practical Example**:

* **index.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Expression Tag Example</title>

</head>

<body>

<h2>Expression Tag Example</h2>

<%

// Declare a Java variable

String message = "Welcome to JSP!";

int length = message.length();

// Declare a method to get current year

int getCurrentYear() {

return 2024;

}

%>

<!-- Printing the value of the variable using Expression Tag -->

<p>Message: <%= message %></p>

<!-- Printing the length of the string -->

<p>Length of the message: <%= length %></p>

<!-- Printing the result of the method using Expression Tag -->

<p>Current Year: <%= getCurrentYear() %></p>

</body>

</html>

**Explanation**:

* **Main Use**: The **expression tag** is mainly used to **print the values** of a variable (message) or method (getCurrentYear()).
* The **message** is declared and then output using <%= message %>.
* The **length** of the message is calculated and printed with <%= length %>.
* The **getCurrentYear()** method is defined, and its return value is displayed using <%= getCurrentYear() %>.

### Key Points:

1. **No out.println() Needed**: The expression tag automatically converts the result of the expression into a string and appends it to the response.
2. **Simple & Concise**: It simplifies the process of outputting dynamic content on a JSP page without writing explicit print statements.
3. **Evaluates Java Code**: The Java code inside the tag is evaluated and the result is placed in the HTML output.

### 3. ****Declaration Tag (****<%! %>****)****

**Description**: The declaration tag is used to declare variables and methods at the class level in JSP. These declarations are shared across multiple requests.

#### **Practical Example**:

* **index.jsp**:

<html>

<head>

<title>Declaration Tag Example</title>

</head>

<body>

<h2>Declaration Tag Example</h2>

<p>Current Count: <%= getCurrentCount() %></p>

<%

out.println("<p>Current Message: " + currentMessage + "</p>");

%>

</body>

</html>

<%!

// Class-level variable

String currentMessage = "Hello, JSP!";

// Method that returns the current count

public int getCurrentCount() {

return 10;

}

%>

**Explanation**:

* The variable currentMessage and the method getCurrentCount() are declared inside <%! %>, making them accessible throughout the JSP page.
* These declarations exist as class members of the generated servlet and can be reused across requests.

### ****Directive Tags****

**Description**: Directives provide global information about the JSP page, like importing classes, including other files, and error handling.

#### **Page Directive**:

Used to define attributes for the page, like importing Java classes.

* **index.jsp**:

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

<html>

<head>

<title>Page Directive Example</title>

</head>

<body>

<h2>Page Directive Example</h2>

<p>Current Date and Time: <%= new Date() %></p>

</body>

</html>

**Explanation**:

* The import attribute imports the java.util.Date class, allowing you to use it in the JSP page.
* This example prints the current date and time using the Date class.

#### **Include Directive**:

Used to include another file at compile time.

* **header.jsp**:

<h1>Header Section</h1>

<hr/>

* **index.jsp**:

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

<html>

<head>

<title>Include Directive Example</title>

</head>

<body>

<h2>Body Section</h2>

<p>This is the body of the page.</p>

</body>

</html>

**Explanation**:

* The header.jsp file is included at the top of the main index.jsp page.
* The included content becomes part of the JSP file at translation time (when the JSP is converted into a servlet).

### 5. ****Action Tags****

**Description**: Action tags are used to control the flow of the application and interact with JavaBeans and other resources.

#### **jsp**

#### :

Dynamically includes a resource (JSP or static file) at request time.

* **header.jsp**:

<h1>Header Section</h1>

<hr/>

* **index.jsp**:

<html>

<head>

<title>jsp:include Example</title>

</head>

<body>

<jsp:include page="header.jsp" />

<h2>Body Section</h2>

<p>This is the body of the page.</p>

</body>

</html>

**Explanation**:

* The jsp:include tag includes the header.jsp file when the page is requested.
* The included file can be a JSP, HTML, or other resources.

#### **jsp**

#### :

Forwards the request to another resource (JSP or servlet).

* **index.jsp**:

<jsp:forward page="welcome.jsp" />

* **welcome.jsp**:

<html>

<head>

<title>Forwarded Page</title>

</head>

<body>

<h2>Welcome Page</h2>

<p>You have been forwarded to this page.</p>

</body>

</html>

**Explanation**:

* When the user accesses index.jsp, they are immediately forwarded to welcome.jsp.
* The jsp:forward tag transfers the control to another JSP or servlet.

#### **jsp**

#### , **jsp**

#### , and **jsp**

#### :

These tags are used to interact with JavaBeans on the JSP page.

* **User.java (JavaBean)**:

package com.example;

public class User {

private String username;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

}

* **index.jsp**:

<jsp:useBean id="user" class="com.example.User" />

<jsp:setProperty name="user" property="username" value="John Doe" />

<html>

<head>

<title>jsp:useBean Example</title>

</head>

<body>

<h2>Welcome, <jsp:getProperty name="user" property="username" />!</h2>

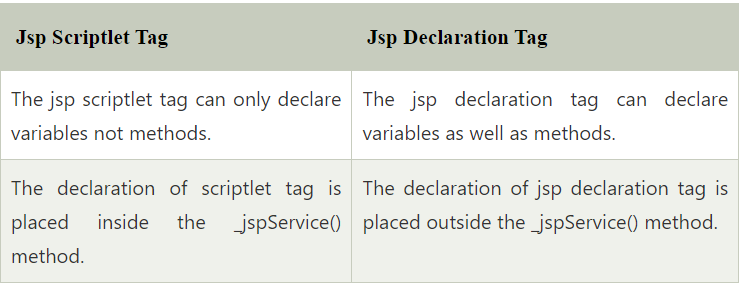
</body>

</html>

**Explanation**:

* The jsp:useBean tag creates an instance of the User bean.
* The jsp:setProperty tag sets the username property of the bean to "John Doe."
* The jsp:getProperty tag retrieves and displays the username property.

**Difference between Scriptlet tag vs Declaration tag**



**JSP Implicit Objects:**

**Implicit objects** in JSP are pre-defined objects that the JSP container creates and makes available to developers for use within the JSP pages without needing to instantiate them. These objects provide access to various functionalities such as handling requests, responses, sessions, and more.

There are **9 implicit objects** in JSP:

1. **request** - (HttpServletRequest)
2. **response** - (HttpServletResponse)
3. **session** - (HttpSession)
4. **application** - (ServletContext)
5. **out** - (JspWriter)
6. **config** - (ServletConfig)
7. **pageContext** - (PageContext)
8. **page** - (Object)
9. **exception** - (Throwable)

### 1. request ****Object**** (HttpServletRequest)

**index.html**:

<!DOCTYPE html>

<html>

<head>

<title>Request Object Example</title>

</head>

<body>

<form action="requestExample.jsp" method="POST">

<label for="uname">Enter your name:</label>

<input type="text" name="uname" id="uname">

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

</form>

</body>

</html>

**requestExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Request Object Example</title>

</head>

<body>

<%

// Getting data from the request

String userName = request.getParameter("uname");

out.print("Hello, " + userName + "!<br/>");

// Fetching additional request information

out.print("Request Method: " + request.getMethod() + "<br/>");

out.print("Client IP Address: " + request.getRemoteAddr() + "<br/>");

%>

</body>

</html>

### 2. response ****Object**** (HttpServletResponse)

**responseExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Response Object Example</title>

</head>

<body>

<%

// Redirecting the response to another page

response.sendRedirect("https://www.example.com");

%>

</body>

</html>

### 3. session ****Object**** (HttpSession)

**sessionExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Session Object Example</title>

</head>

<body>

<%

// Storing a value in the session

session.setAttribute("userName", "Amarjeet Kumar Singh");

// Retrieving the session attribute

String user = (String) session.getAttribute("userName");

out.print("Session User: " + user + "<br/>");

// Invalidating the session (optional)

// session.invalidate();

%>

</body>

</html>

### 4. application ****Object**** (ServletContext)

**applicationExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Application Object Example</title>

</head>

<body>

<%

// Storing a value in the application scope

application.setAttribute("appName", "My Web Application");

// Retrieving the application attribute

String appName = (String) application.getAttribute("appName");

out.print("Application Name: " + appName);

%>

</body>

</html>

### 5. out ****Object**** (JspWriter)

**outExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Out Object Example</title>

</head>

<body>

<%

// Using out to print dynamic content

out.print("Current time: " + new java.util.Date() + "<br/>");

// You can use out.println() for line breaks

out.println("This is printed using out.println() method.<br/>");

%>

</body>

</html>

### 6. config ****Object**** (ServletConfig)

**configExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Config Object Example</title>

</head>

<body>

<%

// Getting servlet config information

String servletName = config.getServletName();

out.print("Servlet Name: " + servletName + "<br/>");

// You can also fetch initialization parameters using config.getInitParameter("paramName")

%>

</body>

</html>

### 7. pageContext ****Object**** (PageContext)

**pageContextExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>PageContext Object Example</title>

</head>

<body>

<%

// Setting an attribute in the page context

pageContext.setAttribute("pageVar", "This is a page context attribute");

// Retrieving the page context attribute

String pageVar = (String) pageContext.getAttribute("pageVar");

out.print(pageVar + "<br/>");

// Forwarding the request to another page (optional)

// pageContext.forward("anotherPage.jsp");

%>

</body>

</html>

### 8. page ****Object**** (Object)

**pageExample.jsp**:

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<html>

<head>

<title>Page Object Example</title>

</head>

<body>

<%

// Printing the page object (equivalent to 'this')

out.print("This is the current JSP page instance: " + page.toString());

%>

</body>

</html>

### 9. exception ****Object**** (Throwable)

**errorPage.jsp**:

<%@ page isErrorPage="true" %>

<html>

<head>

<title>Error Page</title>

</head>

<body>

<h2>Error Details</h2>

<%

// Displaying exception details

if (exception != null) {

out.print("Error: " + exception.getMessage() + "<br/>");

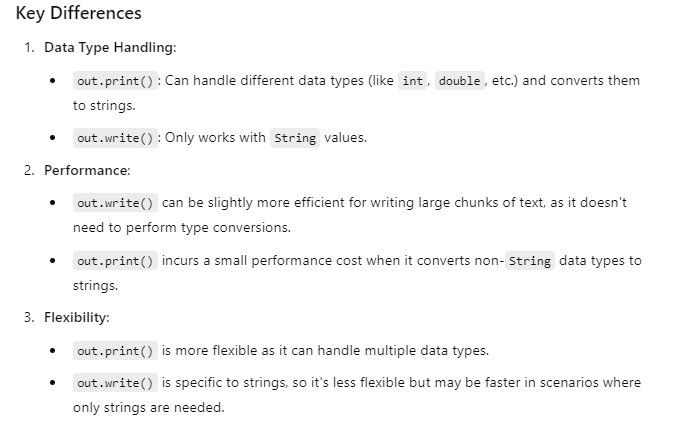
}

%>

</body>

</html>

To trigger the error page, set the errorPage attribute in the main JSP:<%@ page errorPage="errorPage.jsp" %>



**JSP (JavaServer Pages) directives**

In the context of JavaServer Pages (JSP), **directives** are special instructions that provide global settings for the JSP page and influence its overall behavior. They are used to define properties of the JSP page, specify how it should be processed by the server, and control various aspects of its execution.

**Key Points About Directives**

1. **Purpose**: Directives are used to set configuration options for a JSP page. They help in controlling the way the page is compiled and processed by the server.
2. **Syntax**: Directives are written within <%@ %> tags. They are placed at the top of the JSP file, before any HTML or scriptlet code.
3. **Usage**: They are used to:
   * Set up attributes for the page.
   * Import Java classes or packages.
   * Define the content type and character encoding.
   * Manage error handling.
   * Control session management and buffering.

### 1. ****Page Directive****

The page directive defines attributes for the entire JSP page. It affects the behavior and characteristics of the page.

#### Syntax

<%@ page attribute="value" %>

#### Attributes

* **language**: Specifies the scripting language used (e.g., Java).
* **contentType**: Defines the MIME type and character encoding of the response (e.g., text/html; charset=UTF-8).
* **import**: Imports Java classes or packages.
* **session**: Determines if the page should use HTTP sessions (true or false).
* **buffer**: Defines the buffer size for the response (e.g., 8kb).
* **errorPage**: Specifies a JSP page to handle errors.
* **isErrorPage**: Indicates if the page can be an error page (true or false).
* **extends**: Specifies a class to extend.

### 1. language ****Attribute****

The language attribute specifies the scripting language used in the JSP page. For JSP, this is typically set to "java".

#### Syntax

<%@ page language="java" %>

#### Practical Example

<%@ page language="java" %>

<!DOCTYPE html>

<html>

<head>

<title>Language Example</title>

</head>

<body>

<h1>Language Attribute Example</h1>

<%

// Java code

out.println("The scripting language used is Java.");

%>

</body>

</html>

### 2. contentType ****Attribute****

The contentType attribute defines the MIME type and character encoding for the response. This is important for ensuring the content is correctly interpreted by the client.

#### Syntax

<%@ page contentType="text/html; charset=UTF-8" %>

#### Practical Example

<%@ page contentType="text/html; charset=UTF-8" %>

<!DOCTYPE html>

<html>

<head>

<title>Content Type Example</title>

</head>

<body>

<h1>Content Type Attribute Example</h1>

<p>This page uses UTF-8 character encoding.</p>

</body>

</html>

### 3. import ****Attribute****

The import attribute imports Java classes or packages, allowing you to use them in the JSP page.

#### Syntax

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

#### Practical Example

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

<!DOCTYPE html>

<html>

<head>

<title>Import Example</title>

</head>

<body>

<h1>Import Attribute Example</h1>

<%

Date now = new Date();

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

out.println("Current date and time: " + sdf.format(now));

%>

</body>

</html>

### 4. session ****Attribute****

The session attribute determines whether the page should use HTTP sessions. By default, it is true. Setting it to false means the page will not use sessions.

#### Syntax

<%@ page session="false" %>

#### Practical Example

<%@ page session="false" %>

<!DOCTYPE html>

<html>

<head>

<title>Session Example</title>

</head>

<body>

<h1>Session Attribute Example</h1>

<p>This page does not use HTTP sessions.</p>

</body>

</html>

### 5. buffer ****Attribute****

The buffer attribute defines the size of the buffer for the response. This can be set in kilobytes (e.g., 8kb).

#### Syntax

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

#### Practical Example

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

<!DOCTYPE html>

<html>

<head>

<title>Buffer Example</title>

</head>

<body>

<h1>Buffer Attribute Example</h1>

<p>This page uses a 16KB buffer for the response.</p>

</body>

</html>

### 6. errorPage ****Attribute****

The errorPage attribute specifies a JSP page that will handle errors if an exception occurs on the current page.

#### Syntax

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

#### Practical Example

Create an error page (error.jsp):

<!-- error.jsp -->

<%@ page isErrorPage="true" %>

<!DOCTYPE html>

<html>

<head>

<title>Error Page</title>

</head>

<body>

<h1>An Error Occurred</h1>

<p>We apologize for the inconvenience. Please try again later.</p>

</body>

</html>

In your main JSP page:

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

<!DOCTYPE html>

<html>

<head>

<title>Error Handling Example</title>

</head>

<body>

<h1>Error Handling Example</h1>

<%

// Deliberate error for demonstration

int result = 10 / 0;

%>

</body>

</html>

### 7. isErrorPage ****Attribute****

The isErrorPage attribute indicates whether the JSP page can handle errors (i.e., it is an error page). It should be set to true for error pages.

#### Syntax

<%@ page isErrorPage="true" %>

#### Practical Example

If the error page (error.jsp) should handle errors:

<%@ page isErrorPage="true" %>

<!DOCTYPE html>

<html>

<head>

<title>Error Page</title>

</head>

<body>

<h1>Handling Errors</h1>

<p>An error occurred: <%= exception %></p>

</body>

</html>

### 8. extends ****Attribute****

The extends attribute specifies a Java class that the JSP page will extend. This is used to add functionality to the generated servlet.

#### Syntax

<%@ page extends="com.example.MyBaseClass" %>

#### Practical Example

Assuming you have a base class (MyBaseClass.java):

package com.example;

public class MyBaseClass extends HttpServlet {

// Custom functionality

public void customMethod() {

// Method implementation

}

}

In your JSP page:

<%@ page extends="com.example.MyBaseClass" %>

<!DOCTYPE html>

<html>

<head>

<title>Extends Example</title>

</head>

<body>

<h1>Extends Attribute Example</h1>

<%

// Call method from base class

customMethod();

out.println("Method from base class called.");

%>

</body>

</html>

#### Example

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" import="java.util.Date, java.text.SimpleDateFormat" session="false" %>

#### Practical Example

<%@ page language="java" contentType="text/html; charset=UTF-8" import="java.util.Date" %>

<!DOCTYPE html>

<html>

<head>

<title>Page Directive Example</title>

</head>

<body>

<h1>Current Date and Time</h1>

<%

Date now = new Date();

out.println("Current date and time: " + now);

%>

</body>

</html>

### 2. ****Include Directive****

The include directive is used to include the content of another file at compile time. This helps in reusing common code and modularizing JSP pages.

#### Syntax

<%@ include file="filename" %>

#### Example

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

#### Practical Example

Create a common header file (header.jsp):

<!-- header.jsp -->

<header>

<h1>Welcome to My Website</h1>

<nav>

<a href="index.jsp">Home</a> |

<a href="about.jsp">About</a> |

<a href="contact.jsp">Contact</a>

</nav>

</header>

Include the header file in your main JSP (index.jsp):

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

<!DOCTYPE html>

<html>

<head>

<title>Home Page</title>

</head>

<body>

<h2>Home Page Content</h2>

<p>This is the home page of the website.</p>

</body>

</html>

### 3. ****Taglib Directive****

The taglib directive is used to declare a tag library and make custom tags available in a JSP page.

#### Syntax

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

#### Attributes

* **prefix**: Defines the prefix used for the custom tags in the JSP.
* **uri**: Specifies the URI of the tag library descriptor (TLD) file.

#### Example

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

#### Practical Example

Using JSTL (JSP Standard Tag Library) to format dates:

1. Add JSTL library to your project.
2. Use the fmt taglib in your JSP file:

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<!DOCTYPE html>

<html>

<head>

<title>Date Formatting Example</title>

</head>

<body>

<h1>Formatted Date</h1>

<fmt:formatDate value="${now}" pattern="yyyy-MM-dd HH:mm:ss"/>

</body>

</html>

1. Ensure you have the JSTL library included in your project. For Maven, include:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

### Summary

* **Page Directive**: Configures settings for the JSP page (e.g., language, content type).
* **Include Directive**: Includes content from another JSP file at compile time.
* **Taglib Directive**: Declares a tag library for custom tags.(waiting list)

**JSP Action Elements**

**JSP Action Elements** are built-in tags in JavaServer Pages (JSP) that provide ways to perform common operations in a JSP page. They help in controlling the flow of execution, interacting with JavaBeans, including other resources, forwarding requests, and more. They are essential for managing and simplifying various tasks in JSP pages.

**Key JSP Action Elements**

1. **<jsp:include>**:
   * **Purpose**: Includes the content of another JSP file, servlet, or static file at request time. This allows you to reuse code and modularize your JSP pages.
   * **Why Use It**: To maintain modularity and reusability of code. For example, including common headers or footers across multiple pages.
2. **<jsp:forward>**:
   * **Purpose**: Forwards a request from one JSP page or servlet to another resource. This is useful for transferring control within your application.
   * **Why Use It**: To redirect processing to another page or servlet, often used to handle requests in a controlled manner or to chain actions.
3. **<jsp:useBean>**:
   * **Purpose**: Declares and instantiates a JavaBean. It helps in creating and managing reusable components.
   * **Why Use It**: To create and access JavaBeans, which are Java classes used for encapsulating data and business logic in a reusable manner.
4. **<jsp:setProperty>**:
   * **Purpose**: Sets the property values of a JavaBean. It can be used to initialize or modify bean properties.
   * **Why Use It**: To assign values to bean properties dynamically from request parameters or other sources.
5. **<jsp:getProperty>**:
   * **Purpose**: Retrieves the value of a property from a JavaBean and outputs it directly to the client.
   * **Why Use It**: To display the value of a bean’s property on the JSP page, facilitating the dynamic generation of content.
6. **<jsp:plugin>**:
   * **Purpose**: Embeds applets or other plugins in a JSP page.
   * **Why Use It**: For integrating Java applets or other plugin-based content into your JSP pages. (Note: Modern web development practices generally avoid applets and plugins due to compatibility and security concerns.)

**Why Use JSP Action Elements?**

1. **Modularity**:
   * **Reuse**: Action elements like <jsp:include> allow you to reuse common code (e.g., headers, footers) across multiple JSP pages, improving maintainability and consistency.
   * **Encapsulation**: <jsp:useBean> helps in encapsulating data and behavior within JavaBeans, making it easier to manage complex data interactions.
2. **Separation of Concerns**:
   * **Forwarding Requests**: <jsp:forward> helps in separating request processing logic by delegating requests to other resources, making the application easier to manage and debug.
3. **Dynamic Content**:
   * **Interactive Data**: With <jsp:getProperty> and <jsp:setProperty>, you can dynamically interact with JavaBeans and update the content based on user interactions or other conditions.
4. **Integration**:
   * **Embedding**: <jsp:plugin> allows for embedding external components like applets, although its use is less common in modern web development.

**Examples**

1. **Including a Header with <jsp:include>**:

<!-- mainPage.jsp -->

<html>

<head>

<title>Main Page</title>

</head>

<body>

<jsp:include page="header.jsp" />

<h1>Welcome to the Main Page</h1>

<p>This is the main content area.</p>

<jsp:include page="footer.jsp" />

</body>

</html>

1. **Forwarding a Request with <jsp:forward>**:

<!-- processRequest.jsp -->

<jsp:forward page="resultPage.jsp" />

1. **Using JavaBeans with <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty>**:

<!-- userInfo.jsp -->

<jsp:useBean id="user" class="com.example.User" scope="session" />

<jsp:setProperty name="user" property="name" value="Alice" />

<html>

<head>

<title>User Info</title>

</head>

<body>

<p>User Name: <jsp:getProperty name="user" property="name" /></p>

</body>

</html>

In JavaServer Pages (JSP), **action elements** are used to control the flow of the JSP page, handle dynamic content, and perform various operations. They are essentially built-in JSP tags that help in invoking actions and manipulating content. The main action elements in JSP include:

1. **<jsp:include>**
2. **<jsp:forward>**
3. **<jsp:useBean>**
4. **<jsp:setProperty>**
5. **<jsp:getProperty>**
6. **<jsp:plugin>**

**1. <jsp:include>**

The <jsp:include> action is used to include the content of another resource (e.g., a JSP file, servlet, or HTML file) at runtime. This is similar to an HTML <iframe>, but the included content is processed on the server side.

**Syntax:**

<jsp:include page="relativeURL" flush="true|false" />

**Attributes:**

* page: The URL of the resource to be included.
* flush: (Optional) Whether to flush the output buffer before including the content. Default is false.

**Example:**

<!-- main.jsp -->

<html>

<head>

<title>Main Page</title>

</head>

<body>

<h1>Welcome to the Main Page</h1>

<jsp:include page="header.jsp" />

<p>This is the main content of the page.</p>

<jsp:include page="footer.jsp" />

</body>

</html>

<!-- header.jsp -->

<div>

<h2>Header Section</h2>

</div>

<!-- footer.jsp -->

<div>

<p>Footer Section</p>

</div>

**2. <jsp:forward>**

The <jsp:forward> action is used to forward a request to another resource (e.g., another JSP, servlet, or HTML file). This is typically used to pass control from one resource to another, often with some data.

**Syntax:**

<jsp:forward page="relativeURL" />

**Attributes:**

* page: The URL of the resource to which the request is forwarded.

**Example:**

<!-- first.jsp -->

<html>

<head>

<title>First Page</title>

</head>

<body>

<jsp:forward page="second.jsp" />

</body>

</html>

<!-- second.jsp -->

<html>

<head>

<title>Second Page</title>

</head>

<body>

<h1>This is the Second Page</h1>

</body>

</html>

**3. <jsp:useBean>**

The <jsp:useBean> action is used to declare and instantiate JavaBeans components. It helps in creating and using reusable components in JSP pages.

**Syntax:**

<jsp:useBean id="beanName" class="beanClassName" scope="page|request|session|application" />

**Attributes:**

* id: The name of the bean instance.
* class: The fully qualified name of the bean class.
* scope: (Optional) The scope of the bean (page, request, session, application). Default is page.

**Example:**

<!-- useBeanExample.jsp -->

<jsp:useBean id="user" class="com.example.User" scope="session" />

<html>

<head>

<title>Use Bean Example</title>

</head>

<body>

<%

com.example.User user = (com.example.User) pageContext.getAttribute("user");

user.setName("John Doe");

%>

<p>User Name: <jsp:getProperty name="user" property="name" /></p>

</body>

</html>

**4. <jsp:setProperty>**

The <jsp:setProperty> action is used to set the value of a property of a JavaBean.

**Syntax:**

<jsp:setProperty name="beanName" property="propertyName" value="value" />

**Attributes:**

* name: The name of the bean.
* property: The name of the property to be set.
* value: The value to be assigned to the property.

**Example:**

<!-- setPropertyExample.jsp -->

<jsp:useBean id="user" class="com.example.User" scope="session" />

<jsp:setProperty name="user" property="name" value="John Doe" />

<html>

<head>

<title>Set Property Example</title>

</head>

<body>

<p>User Name: <jsp:getProperty name="user" property="name" /></p>

</body>

</html>

**5. <jsp:getProperty>**

The <jsp:getProperty> action is used to retrieve the value of a property of a JavaBean and output it directly to the response.

**Syntax:**

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

**Attributes:**

* name: The name of the bean.
* property: The name of the property to be retrieved.

**Example:**

<!-- getPropertyExample.jsp -->

<jsp:useBean id="user" class="com.example.User" scope="session" />

<jsp:setProperty name="user" property="name" value="Jane Doe" />

<html>

<head>

<title>Get Property Example</title>

</head>

<body>

<p>User Name: <jsp:getProperty name="user" property="name" /></p>

</body>

</html>

**6. <jsp:plugin>**

The <jsp:plugin> action is used to embed applets and other plugins in a JSP page. It is primarily used for embedding Java applets in the web page.

**Syntax:**

<jsp:plugin type="applet" code="appletClass" width="width" height="height">

<jsp:params>

<jsp:param name="paramName" value="paramValue" />

</jsp:params>

</jsp:plugin>

**Attributes:**

* type: The type of plugin (e.g., applet).
* code: The applet class name.
* width: The width of the applet.
* height: The height of the applet.

**Example:**

<!-- pluginExample.jsp -->

<html>

<head>

<title>Plugin Example</title>

</head>

<body>

<jsp:plugin type="applet" code="com.example.MyApplet.class" width="300" height="300">

<jsp:param name="color" value="red" />

</jsp:plugin>

</body>

</html>

**Summary**

**JSP action elements** help in controlling various aspects of JSP processing and interaction. They facilitate including content from other resources, forwarding requests, managing JavaBeans, and integrating applets or plugins into JSP pages. Understanding and effectively using these actions can greatly enhance the functionality and maintainability of JSP-based web applications.

### Include Directive vs. jsp:forward

In JSP (JavaServer Pages), the **Include Directive** and the **jsp:forward** action are both used to include content from other resources, but they serve different purposes and have distinct behaviors. Below, we explore their characteristics, uses, and differences in detail.

### Include Directive

#### Definition

The **Include Directive** is a way to include the contents of another JSP file or resource at compile time into the current JSP file. This means that the included content is merged into the main JSP page during the translation phase, before it is sent to the client.

#### Syntax

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

#### Characteristics

* **Compile-Time Inclusion**: The content of the included file is merged into the JSP page during the compilation stage, so any changes made to the included file will require recompilation of the main JSP.
* **Static**: It behaves like a static inclusion; the included file’s code becomes part of the including file.
* **Shared Context**: The included file has access to all the variables and attributes of the including JSP.
* **No Request Handling**: The request and response objects are shared, meaning both files share the same request context.

#### Use Case

* When you want to include common header or footer files across multiple JSP pages (e.g., navigation bars, footers).
* When you need to include code that should be executed at the time the JSP is compiled.

#### Example

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

<h1>Welcome to My Website</h1>

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

### jsp:forward

#### Definition

The jsp:forward action is used to forward a request to another JSP page or resource at runtime. This means that the current response is terminated, and control is passed to another resource.

#### Syntax

<jsp:forward page="target.jsp" />

#### Characteristics

* **Runtime Forwarding**: The request is forwarded to another resource during runtime, allowing for dynamic processing.
* **New Request Context**: The new resource can process the request and create its response. The original page is not included; rather, the client receives the response of the forwarded resource.
* **No Access to Previous Variables**: Since it’s a new request, the variables in the original JSP page are not accessible in the forwarded JSP.
* **Request and Response Objects**: The request and response objects are forwarded, but any changes made by the original JSP do not persist in the forwarded page.

#### Use Case

* When you want to redirect the user to another page after processing some business logic.
* When you want to navigate to a new JSP page without changing the URL in the browser.

#### Example

<%

// Some business logic here

if (someCondition) {

// Forward to another page

%>

<jsp:forward page="success.jsp" />

<%

} else {

// Forward to an error page

%>

<jsp:forward page="error.jsp" />

<%

}

%>

### Key Differences

| **Feature** | **Include Directive** | **jsp:forward** |
| --- | --- | --- |
| **Timing** | Compile-time inclusion | Runtime forwarding |
| **Behavior** | Merges the content into the JSP page | Redirects the request to another resource |
| **Variable Access** | Shares variables between included files | Does not share variables; new request context |
| **Use Cases** | Common headers, footers, or reusable code | Conditional redirection, navigating between pages |
| **Impact on Response** | Part of the same response | Creates a new response |
| **URL Change** | URL remains the same | URL changes to the forwarded resource |

### Conclusion

The **Include Directive** and **jsp:forward** are essential features in JSP for including content and redirecting requests, respectively. Understanding their differences is crucial for effectively structuring JSP applications, improving maintainability, and managing request flow. Each has its appropriate use case, and knowing when to use one over the other can significantly impact the functionality and performance of a web application.

JSTL in JSP

JSTL (JavaServer Pages Standard Tag Library) is a collection of tags that simplify the development of JSP pages. It provides tags for common tasks such as iteration, conditionals, formatting, and internationalization. By using JSTL, developers can create dynamic web pages without embedding Java code in the JSP, promoting a cleaner separation of concerns.

### Discussion on JSTL Tag Library

1. **Core Tags**: These include basic tags for control flow, iteration, and setting variables.
   * Examples: <c:if>, <c:forEach>, <c:choose>, <c:out>, and <c:set>.
2. **Formatting Tags**: These tags help in formatting numbers, dates, and messages.
   * Examples: <fmt:formatNumber>, <fmt:formatDate>, and <fmt:message>.
3. **SQL Tags**: These allow SQL operations within JSP.
   * Examples: <sql:query>, <sql:update>.
4. **XML Tags**: Used for parsing and transforming XML documents.
   * Examples: <x:parse>, <x:out>.
5. **Function Tags**: These provide functions for string manipulation, collection handling, and more.
   * These functions are defined in the fn namespace.

### How to Implement and Use JSTL

To implement JSTL in your JSP application, follow these steps:

#### Step 1: Add JSTL Dependency

If you are using Maven, add the following dependency to your pom.xml:

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

For a non-Maven project, download the JSTL JAR files and add them to your project's WEB-INF/lib directory.

#### Step 2: Import JSTL in JSP

At the top of your JSP files, include the JSTL core and other tag libraries:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

### Example Application Using JSTL

Here's an example of a simple web application that uses JSTL for displaying a list of employees.

#### Project Structure

/src

├── com.example.mvc

├── Employee.java

├── EmployeeDAO.java

├── EmployeeController.java

/WebContent

├── /WEB-INF

│ ├── web.xml

├── index.jsp

├── listEmployees.jsp

#### Step 3: Model (Employee.java)

package com.example.mvc;

public class Employee {

private int id;

private String name;

private String department;

// Constructors

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

public Employee() {}

// Getters and Setters

public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getDepartment() { return department; }

public void setDepartment(String department) { this.department = department; }

}

#### Step 4: Data Access Object (EmployeeDAO.java)

package com.example.mvc;

import java.util.ArrayList;

import java.util.List;

public class EmployeeDAO {

private List<Employee> employees = new ArrayList<>();

public void addEmployee(Employee employee) {

employees.add(employee);

}

public List<Employee> getAllEmployees() {

return employees;

}

}

#### Step 5: Controller (EmployeeController.java)

package com.example.mvc;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@WebServlet("/EmployeeController")

public class EmployeeController extends HttpServlet {

private EmployeeDAO employeeDAO = new EmployeeDAO();

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

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

if (action.equals("add")) {

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

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

int id = employeeDAO.getAllEmployees().size() + 1; // Auto-incrementing ID

Employee employee = new Employee(id, name, department);

employeeDAO.addEmployee(employee);

response.sendRedirect("EmployeeController?action=list");

}

}

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

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

if (action.equals("list")) {

request.setAttribute("employees", employeeDAO.getAllEmployees());

request.getRequestDispatcher("listEmployees.jsp").forward(request, response);

}

}

}

#### Step 6: JSP Pages

##### index.jsp

<!DOCTYPE html>

<html>

<head>

<title>Employee Management</title>

</head>

<body>

<h1>Welcome to Employee Management</h1>

<form action="EmployeeController" method="post">

<input type="hidden" name="action" value="add">

<label>Name:</label><input type="text" name="name" required>

<label>Department:</label><input type="text" name="department" required>

<input type="submit" value="Add Employee">

</form>

<a href="EmployeeController?action=list">View Employees</a>

</body>

</html>

##### listEmployees.jsp

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

<%@ page import="com.example.mvc.Employee" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!DOCTYPE html>

<html>

<head>

<title>Employee List</title>

</head>

<body>

<h2>Employee List</h2>

<table border="1">

<tr>

<th>ID</th>

<th>Name</th>

<th>Department</th>

</tr>

<c:forEach var="emp" items="${employees}">

<tr>

<td>${emp.id}</td>

<td>${emp.name}</td>

<td>${emp.department}</td>

</tr>

</c:forEach>

</table>

<a href="index.jsp">Add New Employee</a>

</body>

</html>

#### Step 7: Web Descriptor (WEB-INF/web.xml)

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

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

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<servlet>

<servlet-name>EmployeeController</servlet-name>

<servlet-class>com.example.mvc.EmployeeController</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>EmployeeController</servlet-name>

<url-pattern>/EmployeeController</url-pattern>

</servlet-mapping>

</web-app>

### Summary of JSTL Usage

* **Iteration**: The <c:forEach> tag is used to loop through the list of employees.
* **Dynamic Content**: JSP expressions ${emp.id}, ${emp.name}, and ${emp.department} dynamically display the employee details without embedding Java code in the JSP.
* **Cleaner Code**: By using JSTL, your JSP pages are cleaner and easier to maintain, adhering to the MVC pattern.

### Custom Tag: What and Why?

**What is a Custom Tag?** A custom tag in JSP (JavaServer Pages) is a user-defined tag that allows developers to encapsulate complex functionality into reusable components. Custom tags simplify the JSP development process by reducing the need for Java code directly in the JSP, promoting better separation of concerns and enhancing code maintainability.

**Why Use Custom Tags?**

1. **Reusability**: Custom tags can be reused across multiple JSP pages, reducing duplication and promoting consistent functionality.
2. **Encapsulation**: They encapsulate complex logic, making the JSP cleaner and easier to read.
3. **Maintainability**: Changes made to a custom tag affect all pages using it, easing maintenance.
4. **Separation of Concerns**: Custom tags help separate business logic from presentation, adhering to best practices in web development.

### Custom Tag API

The custom tag API is part of the JSP specification and includes several interfaces and classes that developers can use to create custom tags. The primary interfaces include:

1. **Tag**: The basic interface for all custom tags.
2. **SimpleTag**: A more flexible interface that allows for simpler tag implementation without requiring the developer to manage the lifecycle.
3. **TagSupport**: A base class that provides default implementations of the Tag interface.

### Custom Tag Example

Below is a step-by-step example of creating and using a custom tag in a JSP application.

#### Step 1: Create a Custom Tag Class

Create a class that extends SimpleTagSupport or implements Tag. For this example, we'll create a simple tag that displays a greeting message.

**GreetingTag.java**

package com.example.tags;

import java.io.IOException;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.tagext.SimpleTagSupport;

public class GreetingTag extends SimpleTagSupport {

private String name;

// Setter for the 'name' attribute

public void setName(String name) {

this.name = name;

}

@Override

public void doTag() throws JspException, IOException {

if (name != null) {

getJspContext().getOut().println("Hello, " + name + "!");

} else {

getJspContext().getOut().println("Hello, Guest!");

}

}

}

#### Step 2: Define the Tag in tld File

Create a Tag Library Descriptor (TLD) file to define your custom tag.

**tags.tld**

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

<taglib xmlns="http://java.sun.com/xml/ns/javaee"

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

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-jsptaglibrary\_2\_1.xsd"

version="2.1">

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

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

<uri>http://example.com/tags</uri>

<tag>

<name>greet</name>

<tag-class>com.example.tags.GreetingTag</tag-class>

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

<attribute>

<name>name</name>

<required>false</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

</taglib>

Place the tags.tld file in the WEB-INF directory.

#### Step 3: Use the Custom Tag in JSP

Now, you can use your custom tag in a JSP page.

**index.jsp**

<%@ taglib uri="http://example.com/tags" prefix="greeting" %>

<!DOCTYPE html>

<html>

<head>

<title>Custom Tag Example</title>

</head>

<body>

<h1>Custom Tag Example</h1>

<greeting:greet name="John" />

<greeting:greet />

</body>

</html>

### Attributes

Custom tags can accept attributes, as demonstrated in the GreetingTag class. Attributes are defined in the TLD file and can be set in the JSP using the tag's name.

### Iteration

You can create custom tags that iterate over collections or arrays. For example, if you want to create a custom tag to display a list of employees, you could implement it similarly to the greeting tag but with iteration logic.

**EmployeeListTag.java**

package com.example.tags;

import java.io.IOException;

import java.util.List;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.tagext.SimpleTagSupport;

public class EmployeeListTag extends SimpleTagSupport {

private List<String> employees;

// Setter for the 'employees' attribute

public void setEmployees(List<String> employees) {

this.employees = employees;

}

@Override

public void doTag() throws JspException, IOException {

if (employees != null) {

for (String employee : employees) {

getJspContext().getOut().println("<p>" + employee + "</p>");

}

}

}

}

### Custom URI

A custom URI (Uniform Resource Identifier) is specified in the TLD file and is used to uniquely identify the tag library. In the example, the custom URI is defined as http://example.com/tags. This URI is used in the JSP to import the tag library.

### Complete Example

#### Project Structure

/src

├── com.example.tags

├── GreetingTag.java

├── EmployeeListTag.java

/WebContent

├── /WEB-INF

│ ├── web.xml

│ ├── tags.tld

├── index.jsp

#### Step 1: Create EmployeeListTag Class

**EmployeeListTag.java**

package com.example.tags;

import java.io.IOException;

import java.util.List;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.tagext.SimpleTagSupport;

public class EmployeeListTag extends SimpleTagSupport {

private List<String> employees;

// Setter for the 'employees' attribute

public void setEmployees(List<String> employees) {

this.employees = employees;

}

@Override

public void doTag() throws JspException, IOException {

if (employees != null) {

getJspContext().getOut().println("<ul>");

for (String employee : employees) {

getJspContext().getOut().println("<li>" + employee + "</li>");

}

getJspContext().getOut().println("</ul>");

}

}

}

#### Step 2: Update TLD File

Update the tags.tld file to include the new custom tag.

**tags.tld**

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

<taglib xmlns="http://java.sun.com/xml/ns/javaee"

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

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-jsptaglibrary\_2\_1.xsd"

version="2.1">

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

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

<uri>http://example.com/tags</uri>

<tag>

<name>greet</name>

<tag-class>com.example.tags.GreetingTag</tag-class>

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

<attribute>

<name>name</name>

<required>false</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

<tag>

<name>employeeList</name>

<tag-class>com.example.tags.EmployeeListTag</tag-class>

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

<attribute>

<name>employees</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

</taglib>

#### Step 3: Update JSP to Use EmployeeListTag

Modify your JSP to use the EmployeeListTag to display a list of employees.

**index.jsp**

<%@ taglib uri="http://example.com/tags" prefix="greeting" %>

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

<!DOCTYPE html>

<html>

<head>

<title>Custom Tag Example</title>

</head>

<body>

<h1>Custom Tag Example</h1>

<greeting:greet name="John" />

<greeting:greet />

<h2>Employee List</h2>

<%

List<String> employeeNames = new ArrayList<>();

employeeNames.add("Alice");

employeeNames.add("Bob");

employeeNames.add("Charlie");

request.setAttribute("employeeNames", employeeNames);

%>

<greeting:employeeList employees="${employeeNames}" />

</body>

</html>

### Summary of Custom Tags

* **Custom Tags** encapsulate reusable functionality in JSP.
* **Attributes** allow data to be passed to custom tags.
* **Iteration** can be implemented to display collections.
* **Custom URI** uniquely identifies the tag library, enabling its usage in JSP.

### Conclusion

Creating custom tags in JSP provides a powerful mechanism to enhance the reusability and maintainability of web applications. This example demonstrates the entire process of creating, defining, and using

### Expression Language (EL) in JSP

**What is Expression Language (EL)?** Expression Language (EL) is a feature of JSP (JavaServer Pages) that allows for easy access to application data stored in JavaBeans, request parameters, and other objects in a simplified syntax. EL is designed to separate the presentation layer from business logic, enabling developers to write less code while achieving greater functionality.

### Key Features of Expression Language

1. **Simplified Syntax**: EL provides a simpler syntax for accessing data, using ${} notation.
2. **Implicit Objects**: EL can automatically access several implicit objects like request, session, application, etc.
3. **Expression Evaluation**: EL can evaluate expressions to retrieve data, manipulate values, and perform logical operations.

### How to Use EL

1. **Direct Access**: You can directly access JavaBeans properties using dot notation.
2. **Implicit Objects**: EL provides a way to access data from implicit objects without needing to write Java code.

### Define Expression and Use Over the Service Flow

In a typical service flow in a web application, you may have a servlet that processes requests and forwards the data to a JSP page using request attributes. EL allows you to access these attributes easily.

### Achieving EL in JSP: End-to-End Example

#### Step 1: Create a JavaBean

Create a simple JavaBean that represents an employee.

**Employee.java**

package com.example.models;

public class Employee {

private int id;

private String name;

private String department;

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public String getDepartment() {

return department;

}

}

#### Step 2: Create a Servlet to Handle Requests

Create a servlet that populates the Employee object and sets it as a request attribute.

**EmployeeServlet.java**

package com.example.servlets;

import com.example.models.Employee;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

@WebServlet("/EmployeeServlet")

public class EmployeeServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

Employee employee = new Employee(1, "John Doe", "IT");

request.setAttribute("employee", employee);

request.getRequestDispatcher("employee.jsp").forward(request, response);

}

}

#### Step 3: Create the JSP Page

In your JSP page, use EL to access the Employee properties.

**employee.jsp**

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<title>Employee Details</title>

</head>

<body>

<h1>Employee Details</h1>

<p>ID: ${employee.id}</p>

<p>Name: ${employee.name}</p>

<p>Department: ${employee.department}</p>

</body>

</html>

#### Step 4: Configure web.xml (If Necessary)

If you are not using annotations, you may need to define your servlet in web.xml.

**web.xml**

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

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

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<servlet>

<servlet-name>EmployeeServlet</servlet-name>

<servlet-class>com.example.servlets.EmployeeServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>EmployeeServlet</servlet-name>

<url-pattern>/EmployeeServlet</url-pattern>

</servlet-mapping>

</web-app>

### Summary

1. **Expression Language (EL)** provides a simpler way to access data in JSP.
2. **JavaBeans** are used to encapsulate data that can be accessed using EL.
3. **Servlets** process requests and set attributes that can be accessed in JSP using EL.
4. The syntax ${} is used to access properties of JavaBeans in JSP.

### Conclusion

Expression Language enhances the readability and maintainability of JSP pages by simplifying data access. By utilizing JavaBeans and servlets in conjunction with EL, you can create a clean and efficient flow of data in your web applications.

Top of Form

Here's a detailed comparison of **JSTL (JavaServer Pages Standard Tag Library)** and **Expression Language (EL)** in JSP:

### 1. Definition

* **JSTL**:
  + JSTL is a collection of custom tags that simplify the development of JSP pages by providing standard functionalities for common tasks such as looping, conditionals, formatting, and internationalization.
* **Expression Language (EL)**:
  + EL is a simplified language used to access data stored in JavaBeans, collections, and other objects without requiring complex Java code in JSP. It allows developers to express dynamic content using simple expressions.

### 2. Purpose

* **JSTL**:
  + Designed to provide a set of ready-to-use tags that perform common operations, reducing the need for Java code in JSP and promoting cleaner and more maintainable JSP pages.
* **Expression Language (EL)**:
  + Primarily used to evaluate and manipulate data dynamically within JSP pages. EL simplifies the syntax for accessing data stored in JavaBeans and other objects.

### 3. Syntax

* **JSTL**:
  + JSTL uses custom tags with specific prefixes, such as <c:forEach> for iteration and <c:if> for conditionals.
  + Example:

<c:if test="${userLoggedIn}">

<p>Welcome back!</p>

</c:if>

* **Expression Language (EL)**:
  + EL uses a simple syntax for expressions, enclosed in ${} to access properties, attributes, and methods.
  + Example:

<p>User Name: ${user.name}</p>

### 4. Components

* **JSTL**:
  + Comprises various tag libraries, such as:
    - Core tags (<c:\*> for control flow)
    - Formatting tags (<fmt:\*> for date and number formatting)
    - SQL tags (<sql:\*> for database operations)
    - XML tags (<x:\*> for XML processing)
* **Expression Language (EL)**:
  + Provides access to:
    - JavaBeans properties
    - Request, session, and application attributes
    - Collections and arrays
    - Implicit objects like param, header, cookie, etc.

### 5. Functionality

* **JSTL**:
  + Offers more complex functionalities such as looping, conditional rendering, and XML processing.
  + Enables control flow and templating within JSP pages.
* **Expression Language (EL)**:
  + Focused on data access and manipulation.
  + It does not control the flow of logic but allows for dynamic data display.

### 6. Performance

* **JSTL**:
  + Performance may vary based on the complexity of the tags and the number of custom tag libraries used. Generally, it is optimized for common operations.
* **Expression Language (EL)**:
  + Lightweight and efficient, as it directly evaluates expressions and retrieves values from the specified scopes without additional overhead.

### 7. Usage

* **JSTL**:
  + Used when developers need to perform common operations, such as iterating over collections, conditional rendering, formatting data, etc.
  + Example:

<c:forEach var="item" items="${itemList}">

<p>${item.name}</p>

</c:forEach>

* **Expression Language (EL)**:
  + Used for accessing and displaying data in JSP pages, often in conjunction with JSTL.
  + Example:

<p>The total amount is: ${order.totalAmount}</p>

### Conclusion

In summary, **JSTL** provides a set of standard tags for common tasks in JSP development, while **Expression Language (EL)** simplifies data access and manipulation. They are often used together in JSP pages to create dynamic and maintainable web applications. JSTL handles control flow and templating, while EL focuses on retrieving and displaying data.

Top of Form

Bottom of Form

Bottom of Form