# Introduction to JSP (JavaServer Pages)

## 1.1 What is JSP?

### Overview of JSP as a Technology for Creating Dynamic Web Pages

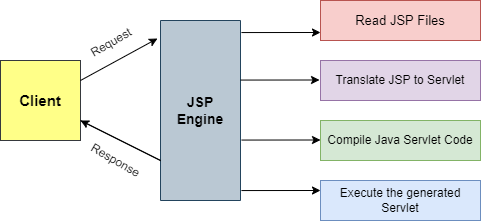
JavaServer Pages (JSP) is a technology used for developing dynamic, interactive web applications. It is part of Java EE (Enterprise Edition) and allows web developers to create dynamic content by embedding Java code directly into HTML pages. JSP provides an easy way to separate the user interface (HTML) from the application logic (Java code), which improves the maintainability and scalability of web applications.

JSP files are essentially HTML pages with embedded Java code, which is executed on the server. This dynamic content is then sent to the client's browser as standard HTML. Unlike static HTML pages, JSP enables the generation of content that can change based on user inputs, database queries, or session states.

#### JSP vs Servlets

* **JSP** allows embedding Java code in HTML using special tags and scriptlets. It's a higher-level abstraction of Servlets, which primarily focus on handling requests and responses in Java.
* **Servlets** are Java classes that handle HTTP requests and generate dynamic responses. While both serve similar purposes, JSP is more focused on presentation, and Servlets focus more on processing logic.

## Architecture of JSP



JSP pages are compiled into Servlets by the web container (e.g., Apache Tomcat) before being executed. This means that when a JSP page is requested, the web container converts it into a Servlet, which is then invoked to process the request.

### Writing Your First JSP Page and Deploying it on the Server

1. Create a new file with a .jsp extension (e.g., index.jsp).
2. Write a basic JSP code inside this file:

<html>

<body>

<h1>Hello, World from JSP!</h1>

</body>

</html>

1. Place the index.jsp file in the webapps directory of your Tomcat installation (or your web container's equivalent).
2. Start Tomcat and visit http://localhost:8080/your\_project\_name/index.jsp to see the result.

## 1.4 Basic JSP Syntax and Structure

### Understanding the Basic Syntax of JSP

JSP syntax allows the mixing of HTML with Java code. There are specific JSP tags to embed Java code:

* **Directives**: Provide information about the page (e.g., page settings, imports).
* **Scriptlets**: Contain Java code that is executed during the request-response cycle.
* **Expressions**: Short Java code that gets evaluated and inserted directly into the output stream.
* **Declarations**: Define variables or methods for use within the JSP page.

## Directives

Directives in JSP are special instructions that provide meta-information about the page.

### Types of Directives

There are three main types of directives in JSP:

1. **Page Directive**:
   * This directive provides overall settings for the JSP page, such as content type, language, buffer size, and error pages.

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

<html>

<body>

<h1>Welcome to JSP Directives!</h1>

</body>

</html>

1. **Include Directive**:
   * This directive is used to include the content of another file (typically another JSP page or static HTML file) into the current JSP page at the time of translation.

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

<html>

<body>

<h1>Content goes here!</h1>

</body>

</html>

1. **Taglib Directive**:
   * This directive is used to import a custom tag library into the JSP page. It allows you to use custom tags that are defined in tag libraries (like JavaServer Pages Standard Tag Library (JSTL)).

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

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

<p>${item}</p>

</c:forEach>

## Scriptlets

### Introduction to Scriptlets in JSP

Scriptlets in JSP allow you to embed Java code directly within your JSP pages. Scriptlets are enclosed in <% %> tags.

**Syntax of Scriptlets**:

<%

// Java code goes here

int x = 5;

int y = 3;

int sum = x + y;

%>

### Example:

Suppose you have a simple form where the user inputs two numbers, and you want to calculate the sum using a scriptlet.

**HTML Form (index.jsp)**:

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

Number 1: <input type="text" name="num1" /><br/>

Number 2: <input type="text" name="num2" /><br/>

<input type="submit" value="Calculate" />

</form>

**JSP Page with Scriptlet (sum.jsp)**:

<%

String num1 = request.getParameter("num1");

String num2 = request.getParameter("num2");

int n1 = Integer.parseInt(num1);

int n2 = Integer.parseInt(num2);

int sum = n1 + n2;

%>

<h2>Sum: <%= sum %></h2>

## Expressions

JSP expressions can be used for inserting dynamic content in an HTML page. The expression is evaluated and the result is written directly to the output stream.

### Examples

<html>

<body>

<h2>Current Date and Time: <%= new java.util.Date() %></h2>

</body>

</html>

This example will display the current date and time when the page is loaded.

## Declarations

In JSP, you can declare variables or methods that are accessible throughout the page. Declarations are placed inside <%! %> tags.

* **Syntax for Declaring Variables**:

<%! int counter = 0; %>

* **Syntax for Declaring Methods**:

<%!

public int factorial(int n) {

int result = 1;

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

result \*= i;

}

return result;

}

%>

<html>

<body>

<h2>Factorial of 5: <%= factorial(5) %></h2>

</body>

</html>

In this example, the factorial() method is declared at the top of the page and used later to compute and display the factorial of 5.

## JSP Implicit Objects

### Overview of Implicit Objects Available in JSP

JSP provides a set of implicit objects, which are automatically available to developers and represent various components of the web request, session, and response. These objects make it easier to work with requests, sessions, and responses without explicitly passing them around.

### Common Implicit Objects

1. **request**: Represents the HttpServletRequest object, which contains information about the HTTP request (e.g., form data, query parameters).
2. **response**: Represents the HttpServletResponse object, which is used to send data back to the client (e.g., sending HTML or redirecting).
3. **session**: Represents the HttpSession object, which is used to store session data between requests from the same user.
4. **application**: Represents the ServletContext, which allows sharing data across the entire application.
5. **out**: Represents the PrintWriter object used to send output to the client.
6. **config**: Represents the ServletConfig object, which contains initialization parameters for the Servlet.
7. **pageContext**: Provides access to various context objects (e.g., request, response, session, application).

### Example: Using request and response Objects for Handling Form Data

Let’s look at a simple example where the request object is used to retrieve form data and display it:

**HTML Form (form.jsp)**:

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

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

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

</form>

**JSP Page (process.jsp)**:

<%

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

%>

<h2>Hello, <%= username %>!</h2>

In this example, the request.getParameter("username") method retrieves the value of the username form field, and the result is displayed on the page.

### Comments in JSP

* **Comments**: Comments in JSP can be written in two ways:
  + JSP comment (does not appear in the output):
  + <%-- This is a JSP comment --%>
  + HTML comment (appears in the output):
  + <!-- This is an HTML comment -->

This concludes the comprehensive introduction to JSP.

# Working with JavaBeans and EL (Expression Language)

## Introduction to JavaBeans

### What is a JavaBean and How Does It Integrate with JSP?

A **JavaBean** is a reusable software component that follows certain conventions for creating and manipulating objects in Java. JavaBeans are used to represent data and business logic in web applications. They can be easily integrated into JSP pages for dynamic content generation.

A JavaBean class typically follows these conventions:

1. **Private Properties**: All properties (fields) in the class are private, ensuring encapsulation.
2. **Public Getter and Setter Methods**: JavaBeans provide public methods to access (get) and modify (set) the values of private properties.
3. **Default Constructor**: JavaBeans must provide a no-argument constructor so that they can be instantiated easily.
4. **Serializable**: JavaBeans should implement the Serializable interface to allow the bean’s state to be saved or transferred.

In a JSP application, JavaBeans act as a model or data container, which can be used to store and manipulate data (such as a user’s details, settings, etc.). You can access and modify the properties of a JavaBean directly in JSP using special tags or Expression Language (EL).

### Example:

Here’s an example of a simple JavaBean:

public class Person implements java.io.Serializable {

private String name;

private int age;

// Default constructor

public Person() {}

// Getter for 'name'

public String getName() {

return name;

}

// Setter for 'name'

public void setName(String name) {

this.name = name;

}

// Getter for 'age'

public int getAge() {

return age;

}

// Setter for 'age'

public void setAge(int age) {

this.age = age;

}

}

In this example, the Person class has two private fields: name and age. The class also provides public getter and setter methods for each property.

## Using JavaBeans in JSP

### How to Bind JavaBeans Properties to JSP Elements

You can bind a JavaBean’s properties to JSP elements using the following JSP tags:

* **<jsp:useBean>**: Declares and initializes a JavaBean.
* **<jsp:setProperty>**: Sets the property of a JavaBean.
* **<jsp:getProperty>**: Retrieves the property of a JavaBean.

These tags allow you to interact with JavaBeans in JSP without embedding Java code directly in the page.

### Using <jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty> Tags

1. **<jsp:useBean> Tag**: The <jsp:useBean> tag is used to declare and instantiate a JavaBean. If the bean already exists in the specified scope (e.g., session, request), it is reused; otherwise, a new instance is created.

**Syntax**:

<jsp:useBean id="beanName" class="beanClassName" scope="scope" />

* + id: The name of the JavaBean (used to refer to it in the JSP).
  + class: The fully qualified class name of the JavaBean.
  + scope: (optional) The scope in which the bean is available (page, request, session, application).

**Example**:

<jsp:useBean id="person" class="com.example.Person" scope="session" />

1. **<jsp:setProperty> Tag**: The <jsp:setProperty> tag is used to set the value of a property in a JavaBean. You can specify the property name and value to assign to the bean.

**Syntax**:

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

* + name: The ID of the JavaBean.
  + property: The property of the JavaBean to be set.
  + value: The value to assign to the property.

**Example**:

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

1. **<jsp:getProperty> Tag**: The <jsp:getProperty> tag retrieves the value of a property from a JavaBean.

**Syntax**:

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

* + name: The ID of the JavaBean.
  + property: The property whose value you want to retrieve.

**Example**:

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

### Example:

**Form to Display and Update Person Object (person.jsp)**:

<jsp:useBean id="person" class="com.example.Person" scope="session" />

<!-- Display current person's name and age -->

<h2>Person Information:</h2>

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

<p>Age: <jsp:getProperty name="person" property="age" /></p>

<!-- Form to update person's details -->

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

Name: <input type="text" name="name" value="<jsp:getProperty name='person' property='name' />" /><br/>

Age: <input type="text" name="age" value="<jsp:getProperty name='person' property='age' />" /><br/>

<input type="submit" value="Update" />

</form>

**Updating the Person Object (updatePerson.jsp)**:

<jsp:useBean id="person" class="com.example.Person" scope="session" />

<!-- Set new values for the person's properties -->

<jsp:setProperty name="person" property="name" value="<%= request.getParameter("name") %>" />

<jsp:setProperty name="person" property="age" value="<%= request.getParameter("age") %>" />

<!-- Redirect back to person.jsp -->

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

In this example:

* The first JSP (person.jsp) displays the current Person object's properties and allows the user to modify them.
* The second JSP (updatePerson.jsp) updates the Person object with new values and then redirects back to the first page to display the updated information.

## Introduction to Expression Language (EL)

### What is EL and How It Simplifies Accessing Data in JSP?

Expression Language (EL) is a powerful and concise way to access and manipulate data in JSP pages. It is designed to simplify the access of JavaBeans properties, collections, and other objects within the JSP, and it eliminates the need for Java code (like scriptlets) in the page.

EL allows you to:

* Access bean properties, session attributes, and request parameters.
* Perform basic operations (e.g., math, string manipulation).
* Iterate over collections (e.g., lists or maps).

### EL Syntax and Operators

1. **Basic EL Syntax**: EL expressions are written within ${}.

Example:

${person.name}

This retrieves the name property of the person JavaBean.

1. **Operators in EL**:
   * **Dot Operator**: Accesses a property or method of an object (e.g., ${person.name}).
   * **Subscript Operator**: Accesses an element from a collection (e.g., ${list[0]}).
   * **Logical Operators**: Supports logical operations (&&, ||, !).
   * **Arithmetic Operators**: Supports basic math operations (+, -, \*, /).

### Examples

* **Reading JavaBean Property**:

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

* **Reading Collection Elements**:

<p>${items[0]}</p>

* **Accessing Session Attributes**:

<p>${sessionScope.username}</p>

* **Performing Arithmetic Operations**:

<p>${5 + 3}</p>

# Handling Forms and User Input in JSP

### Handling Form Data Submission via POST and GET Methods

Forms can submit data to the server using two HTTP methods: **GET** and **POST**.

* **GET Method**: The data is appended to the URL and is visible to the user.
* **POST Method**: The data is sent in the body of the HTTP request and is not visible in the URL.

**Example**:

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

<input type="text" name="username" placeholder="Enter username" />

<input type="password" name="password" placeholder="Enter password" />

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

</form>

In this example, the form uses the **POST** method to submit the data to processForm.jsp.

## Processing Form Data in JSP

### Retrieving Form Data Using request.getParameter()

In JSP, you can retrieve form data submitted by the user using request.getParameter("parameterName"). This method fetches the value of the form field with the specified name.

**Example**: Retrieving Form Data in JSP

<%

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

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

%>

<p>Username: <%= username %></p>

<p>Password: <%= password %></p>

Here, request.getParameter("username") retrieves the value entered in the username field, and request.getParameter("password") retrieves the value of the password field.

### Using <jsp:useBean> to Handle Form Data

You can also use JavaBeans to handle form data in a structured way. The <jsp:useBean> tag creates or accesses a JavaBean, and the <jsp:setProperty> tag is used to set the form data into the JavaBean.

**Example**: Capturing User Input and Displaying It on a New Page

1. **Form Page (inputForm.jsp)**:

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

<input type="text" name="username" placeholder="Enter username" />

<input type="password" name="password" placeholder="Enter password" />

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

</form>

1. **Processing Page (processForm.jsp)**:

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

<jsp:setProperty name="user" property="username" value="<%= request.getParameter("username") %>" />

<jsp:setProperty name="user" property="password" value="<%= request.getParameter("password") %>" />

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

In this example, processForm.jsp uses the User JavaBean to handle the form data and display the username.

## Handling Multiple Forms

### Managing Multiple Forms in a Single JSP Page

When you have multiple forms on the same JSP page, you need to distinguish between them to determine which form was submitted. You can do this by using **hidden fields** or form names.

1. **Using Hidden Fields**: You can use a hidden field to identify which form was submitted.

**Example**:

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

<input type="hidden" name="formType" value="login" />

<input type="text" name="username" />

<input type="password" name="password" />

<input type="submit" value="Login" />

</form>

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

<input type="hidden" name="formType" value="register" />

<input type="text" name="username" />

<input type="password" name="password" />

<input type="submit" value="Register" />

</form>

1. **Processing Multiple Forms**: In your processForm.jsp, you can use the hidden field to identify which form was submitted.

**Example**:

<%

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

if ("login".equals(formType)) {

// Process login form

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

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

out.println("Login successful for: " + username);

} else if ("register".equals(formType)) {

// Process registration form

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

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

out.println("Registration successful for: " + username);

}

%>

In this example, by checking the value of formType, the server can determine which form was submitted and process the data accordingly.

## Custom Tags in JSP

### What Are Custom Tags and Why Use Them?

**Custom tags** in JSP allow you to define your own tags for use within JSP pages. These tags are a powerful feature because they allow you to encapsulate complex logic in a reusable and easy-to-understand manner. Custom tags enable developers to create modular and maintainable code by abstracting logic and functionality away from the page itself.

Custom tags are typically used to:

* Reuse common logic across multiple JSP pages.
* Simplify JSP code by abstracting complex logic.
* Improve the readability of JSP pages by using descriptive custom tags.

### How to Create and Use Custom Tags in JSP

1. **Create a Tag Handler Class**: A Java class that defines the behavior of the custom tag. The tag handler class must extend TagSupport or BodyTagSupport (depending on the type of tag).
2. **Define the Tag in the TLD (Tag Library Descriptor)**: The TLD file defines the tag and links it to the tag handler class.
3. **Use the Tag in a JSP**: After defining the custom tag in the TLD, it can be used within a JSP page.

#### Example of a Custom Tag:

1. **Tag Handler Class (UppercaseTag.java)**:

import javax.servlet.jsp.tagext.TagSupport;

import javax.servlet.jsp.JspException;

import java.io.IOException;

public class UppercaseTag extends TagSupport {

private String text;

public void setText(String text) {

this.text = text;

}

public int doStartTag() throws JspException {

try {

pageContext.getOut().write(text.toUpperCase());

} catch (IOException e) {

e.printStackTrace();

}

return SKIP\_BODY; // Skip the body content of the tag

}

}

1. **TLD File (tags.tld)**:

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

<taglib xmlns="http://java.sun.com/xml/ns/jsp" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/jsp http://java.sun.com/xml/ns/jsp/web-jsptaglibrary\_2\_0.xsd" version="2.0">

<tag>

<name>uppercase</name>

<tag-class>com.example.UppercaseTag</tag-class>

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

<attribute>

<name>text</name>

<required>true</required>

</attribute>

</tag>

</taglib>

1. **JSP Page (example.jsp)**:

<%@ taglib uri="/WEB-INF/tags.tld" prefix="custom" %>

<html>

<body>

<custom:uppercase text="hello world" />

</body>

</html>

In this example, the custom tag <custom:uppercase> will output the text in uppercase (HELLO WORLD).

### Overview of Tag Libraries: Java Standard Tag Library (JSTL)

JSTL provides a collection of useful tags that simplify common tasks in JSP pages, such as iteration, conditional rendering, and formatting. JSTL eliminates the need for Java code inside JSP, helping to maintain cleaner and more readable pages.

## JSTL (JavaServer Pages Standard Tag Library)

### Introduction to JSTL and Its Core Tags

JSTL is a tag library that encapsulates common functionality that developers typically perform in Java code within JSP pages. Some of the most common JSTL tags include:

**<c:forEach>**: Used for iterating over a collection or array.

**<c:if>**: Used for conditional rendering based on a boolean expression.

**<c:choose>, <c:when>, and <c:otherwise>**: Used for multi-condition checks (similar to switch/case).

### Using JSTL to Simplify Java Code in JSP

JSTL allows you to eliminate the need for scriptlets in JSP, which improves readability and maintainability of the page.

#### Example: Looping Through a Collection Using <c:forEach>

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

<c:set var="names" value="${['John', 'Jane', 'Doe']}" />

<ul>

<c:forEach var="name" items="${names}">

<li>${name}</li>

</c:forEach>

</ul>

This example uses the <c:forEach> tag to loop over a list of names and display each name inside a list item (<li>).

#### Example: Conditional Rendering with <c:if>

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

<c:if test="${user.age >= 18}">

<p>You are an adult.</p>

</c:if>

<c:if test="${user.age < 18}">

<p>You are a minor.</p>

</c:if>

In this example, the <c:if> tag conditionally displays different messages depending on the value of user.age.

## Error Handling in JSP

### Techniques for Handling Errors and Exceptions in JSP Pages

Error handling is an important aspect of any web application to ensure that users have a smooth experience, even when things go wrong. There are two primary methods for handling errors in JSP:

1. **Using <error-page> in web.xml**: You can map specific error codes (e.g., 404 or 500) to a custom error page in the web.xml configuration file.
2. **Using try-catch in JSP**: You can manually handle exceptions in JSP using standard Java exception handling mechanisms.

#### Using <error-page> in web.xml

<web-app>

<error-page>

<error-code>404</error-code>

<location>/error/404.jsp</location>

</error-page>

<error-page>

<exception-type>java.lang.Exception</exception-type>

<location>/error/generalError.jsp</location>

</error-page>

</web-app>

In this example:

* If a 404 Not Found error occurs, the user is redirected to 404.jsp.
* If any exception occurs, the user is redirected to generalError.jsp.

#### Handling Exceptions within JSP

You can use the exception implicit object in JSP to display detailed error information.

<%@ page isErrorPage="true" %>

<html>

<body>

<h3>An error occurred:</h3>

<p><%= exception.getMessage() %></p>

</body>

</html>

This example displays the exception message on a custom error page.

## JSP and Session Management

Session management allows web applications to track user interaction across multiple requests. In a typical web application, sessions are used to store user-specific data such as login credentials, user preferences, or shopping cart contents.

### Using HttpSession Object for Storing Session Data

The HttpSession object allows you to store data for a particular user across multiple requests. This data is typically stored on the server.

#### Example: Storing User Login Details in a Session

<%@ page import="javax.servlet.http.HttpSession" %>

<%

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

HttpSession session = request.getSession();

session.setAttribute("username", username);

%>

<p>Welcome, <%= session.getAttribute("username") %>!</p>

This example stores the username in the session and displays it on the page.

## JSP Filters and Listeners

Filters and listeners provide hooks into the request and response lifecycle, allowing you to perform specific actions before or after a request is processed by a JSP page or servlet.

#### Filters: Manipulating Request and Response

A **filter** is an object that performs filtering tasks on either the request to a resource, the response from a resource, or both.

* **Example Use Cases**:
  + Logging request details.
  + Modifying request/response content (e.g., compressing response data).

#### Example: Creating a Simple Filter

import javax.servlet.\*;

import javax.servlet.annotation.WebFilter;

import java.io.IOException;

@WebFilter("/myServlet")

public class LoggingFilter implements Filter {

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

System.out.println("Request received: " + request.getRemoteAddr());

chain.doFilter(request, response); // Continue processing the request

}

}

In this example, the filter logs the remote address of the incoming request before passing it on to the servlet.

#### Listeners: Tracking Lifecycle Events

A **listener** is an object that listens to specific events in the web application's lifecycle, such as session creation, session destruction, or application startup.

* **Example Use Cases**:
  + Tracking user login/logout events.
  + Performing initialization or cleanup tasks when a session is created or destroyed.

#### Example: Creating a Session Listener

import javax.servlet.http.\*;

public class SessionListener implements HttpSessionListener {

public void sessionCreated(HttpSessionEvent se) {

System.out.println("Session created: " + se.getSession().getId());

}

public void sessionDestroyed(HttpSessionEvent se) {

System.out.println("Session destroyed: " + se.getSession().getId());

}

}

In this example, the listener tracks the creation and destruction of sessions.