What is Java Server Page (JSP)?

- A **dynamic web page technology** (HTML + embedded Java).
- Auto-translated to a **servlet** by the container.
- Lifecycle managed by **JSP container** (part of web container).

Why JSP?

- Separates **Presentation Logic (PL)** from **Business Logic (BL)**.
- Auto compilation = easier/quicker than writing servlets.
- But in modern apps, replaced by **front-end libraries/frameworks (React/Angular/Vue)**.

JSP Lifecycle (in a nutshell)

- Translation → Compilation → Loading → Instantiation → jspInit() → _jspService() (per request) → jspDestroy().
- **Never override _jspService()**.

JSP API Overview

- **Part of Java EE / Jakarta EE specs**
- Provided in **jsp-api.jar** (the *specification classes/interfaces*).
- In Tomcat:
  - **jsp-api.jar** → defines the API (interfaces).
  - **jasper.jar** → Tomcat's **JSP engine implementation** (compiler: Jasper).

Inheritance Hierarchy

jakarta.servlet.Servlet (interface)

↑

jakarta.servlet.jsp.JspPage (interface)

↑

jakarta.servlet.jsp.HttpJspPage (interface)

↑

Generated Servlet class for your JSP (implements HttpJspPage)

Methods in Interfaces

jakarta.servlet.jsp.JspPage

- Extends **Servlet**.
- Adds JSP lifecycle methods:
  - void jspInit() → like init(), called once at JSP init.
  - void jspDestroy() → like destroy(), called once before JSP unload.

You **can override** these in JSP page using <%! %> declaration.

jakarta.servlet.jsp.HttpJspPage

- Extends JspPage.
- Adds:
- void _jspService(HttpServletRequest req, HttpServletResponse res)

  throws ServletException, IOException;
- **Called for every request**.
- Auto-generated by JSP container → **Do NOT override**.
- JSP page body (HTML + JSP elements) gets translated into this method.

4. Big Picture (Generated Servlet)

When you write a JSP page(tes.jsp), container generates something like:

```
public final class test_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements jakarta.servlet.jsp.HttpJspPage {
  public void jspInit() {
    // custom init (if declared in JSP)
  }
  public void jspDestroy() {
    //custom cleanup code(if declared in JSP)
  }
```

```
    public void _jspService(HttpServletRequest request,
HttpServletResponse response)
            throws IOException, ServletException {
        // auto-generated code from JSP body
        response.setContentType("text/html");
        JspWriter out = response.getWriter();
        out.write("<html> ... </html>");
    }
}
```

JSP Life Cycle (Step by Step)

1. **Client Request**
    - Browser sends a request for test.jsp.
2. **Translation Phase (by JSP Container → Jasper in Tomcat)**
    - .jsp → translated into an equivalent servlet .java file (e.g., test_jsp.java).
    - If there are **JSP syntax errors**, translation fails and lifecycle aborts.
3. **Compilation Phase (by JSP Container)**
    - .java (servlet source) → compiled into .class file (e.g., test_jsp.class). It's a final class.
    - If there are **Java syntax errors**, compilation fails.
4. **Class Loading & Initialization (by Servlet Container → Catalina in Tomcat)**
    - Compiled .class is loaded into JVM.
    - An instance of the generated servlet is created.
    - jspInit() is called **once per JSP lifecycle** . At that point, the **ServletConfig** is already injected by the WC .
      → Used for **one-time initialization logic**.
5. **Request Processing (Run Time Phase)**
    - For each incoming client request:
        - A new thread is taken from the thread pool.

- _jspService(HttpServletRequest req, HttpServletResponse res) is invoked.
  → This method is auto-generated and contains the logic from JSP + HTML.
- After _jspService returns, the thread goes back to the pool.

6. **Destruction Phase**
   - When the application is undeployed or server shuts down:
     - jspDestroy() is called **once**.
     - JSP instance is garbage collected afterwards.

JSP 3.x Syntax

1. JSP Comments

Type	Syntax	Notes
Server-side comment	<%-- comment text --%>	Ignored by JSP translator/compiler. never sent to client.
Client-side comment	<!-- comment text -->	Included in HTML output; ignored by browser while rendering.

2. JSP Implicit Objects

Created **inside _jspService** (available in scriptlets & expressions) cannot use these in declaration blocks or

Object	Type	Purpose
out	jakarta.servlet.jsp.JspWriter	Buffered writer to client (similar to PrintWriter).
request	HttpServletRequest	Current client request.
response	HttpServletResponse	Response to the client.
config	ServletConfig	Servlet config,to access init params.
session	HttpSession	Client session; created by default.
exception	Throwable	Available **only on error pages**.

Object	Type	Purpose
pageContext	PageContext	Centralized object storing page, session, request, application, out, exception, config. Useful for managing page-scoped attributes.
application	ServletContext	Application-scoped attributes, logging, request dispatching.
page	Object	Current JSP instance (this), rarely used directly.

3. Scripting Elements

1. **Scriptlets** <% code %>
    - Java code injected directly into _jspService.
    - Avoid in modern JSP; prefer JSTL / EL.
    - Can access implicit objects: request, session, application, etc.
2. **Expressions** <%= expression %>
    - Evaluated and converted to String, sent to client.
    - Examples:
    - <%= new Date() %>
    - <%= session.getId() %>
    - <%= request.getAttribute("user") %>
3. **Declarations** <%! declaration %>
    - Define class-level fields or methods.
    - Executed **outside _jspService**.
    - Example:
    - <%! int counter = 0; %>
    - <%! public int increment() { return ++counter; } %>

4. Expression Language (EL)

- **Concise alternative** to <%= %>.

- Syntax: ${expression}
- Added **directly in JSP body**, not in declaration blocks.
- Mainly used for accessing attributes , under different scopes.

EL Implicit Objects / Maps:

EL Object	**Description**
param	Map of request parameters (single value)
paramValues	Map of request parameters (multi-values)
header	Map of request headers
cookie	Map of cookies
pageScope	Page-scoped attributes
requestScope	Request-scoped attributes
sessionScope	Session-scoped attributes
applicationScope	Application-scoped attributes

Example:

${param.username} request.getParameter("username")

${sessionScope.user.name} session.getAttribute("user").getName()

${applicationScope.discount_offer}

application.getAttribute("discount_offer")

- If you use ${abc}, EL searches for “abc” attributes in **page** → **request** → **session** → **application** in order. Returns blank if not found.

${paramValues.hobby}

request.getParameterValues(“hobby”) : String[]

${cookie.theme.value} Fetches value of cookie named theme.

${pageContext.session.id} Get current session ID.

${pageContext.request.contextPath} Get context path of web app.

${pageContext.session.maxInactiveInterval}	Get session timeout interval in seconds.

```
<c:set var="a" value="10" />
<c:set var="b" value="20" />
```

${a + b} Arithmetic addition →30

${a * b} Multiplication → 200

${a == b} Comparison → false

${a lt b} Less than comparison → true (lt is alternative to <)

Collections (Lists, Maps, Arrays)

```
<%
List<String> fruits = Arrays.asList("Apple","Banana","Cherry");
request.setAttribute("fruits", fruits);
Map<String, String> colors = Map.of("Apple","Red","Banana","Yellow");
request.setAttribute("colors", colors);
%>
```

EL Example	Description
${fruits[0]}	Access first element of list → "Apple"
${colors['Apple']}	Access value from map → "Red"
${colors.Apple}	Alternative dot notation → "Red"

Conditional Rendering

```
<c:set var="loggedIn" value="true" />
```

EL Example	Description
${loggedIn ? 'Welcome User' : 'Please login'}	Ternary operator (true → 'Welcome User').
${empty param.username ? 'Guest' : param.username}	Checks if username parameter is empty.

Using EL in JSTL

```
<c:forEach var="fruit" items="${fruits}">
   ${fruit}<br/>
</c:forEach>


<c:if test="${param.action == 'delete'}">
   Action is delete
</c:if>
```

EL **can call public methods** (like ${pageContext.session.invalidate()}), but it's **not recommended** for side effects.

Later with Java Beans , you can call any public method , even with parameters using EL syntax.

Use EL **instead of scriptlets** for cleaner JSPs.

JSP Declarations

Syntax:

- <%! declaration block %>
- **Placed outside <body>** (so outside _jspService()).
- **Usage**:
    1. Define **instance variables** and **methods** of the translated servlet class.
    2. Override lifecycle methods (jspInit, jspDestroy).
    3. Define static variables/methods if needed.

Example:

```
<%!
  int counter = 0;
  public void jspInit() {
     System.out.println("JSP initialized!");
  }
%>
```

<body>

<%= ++counter %>

</body>

JSP Directives

Directives are instructions to the **JSP Engine (Jasper in Tomcat)** during **translation phase**.
Syntax:

<%@ directiveName attribute="value" %>

1. page directive

Applies to **current JSP only**.
Common attributes of page directive

1. **import**
    - Import Java classes/packages.
    - Eg. <%@ page import="java.util.*,java.text.SimpleDateFormat" %>
2. **contentType**
    - Sets MIME type + charset.

    Eg. <%@ page contentType="text/html;charset=UTF-8" %>
3. **session** (default = true)
    - Whether JSP participates in session tracking.
    - If false, session implicit object is not created.
4. **errorPage / isErrorPage**
    - Used in Centralized error handling.

Eg. <%@ page errorPage="error.jsp" %> & in error.jsp

<%@ page isErrorPage="true" %>

- If isErrorPage="true", implicit object “exception” is available.
- You can also access ErrorData object then.

EL usage in error page:

Error Message : ${pageContext.exception.message}

Request URI: ${pageContext.errorData.requestURI}

Status Code: ${pageContext.errorData.statusCode}

Throwable: ${pageContext.errorData.throwable}

5. **isThreadSafe** (default = true)
    - true: Developer ensures thread-safety manually. (Recommended)
    - false: Web Container synchronizes _jspService().(Bad practice since removes concurrency).

Best practice: Keep isThreadSafe="true" and synchronize only critical sections (like application scoped attributes).

```
<%
 synchronized(application) {
   application.setAttribute("loan_scheme", "HomeLoan2025");
 }
%>
```

Equivalent step in Servlets: implement SingleThreadModel (deprecated interface).

2. include directive

Eg. In one .jsp

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

- Contents of included page are **merged** into one.jsp at **translation time** → part of the same translated servlet class.
- **Scope**: Page scope (behaves like one page).
- Best for including static content (headers, footers, menus).
- If included file changes, JSP must be re-translated.

3. taglib directive

To use **JSTL** or **custom tags**.

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

JSP Actions

Commands for the **Web Container (WC)**, to be executed at **request processing** (runtime) phase.

Syntax:

<jsp:actionName attribute="value" />

OR

<jsp:actionName> ...body... </jsp:actionName>

Why JavaBeans in JSP?

- Separates **Business Logic (BL)** from JSP (Presentation Layer).
- **Stateful component**: Bean properties represent client's conversational state.
- **Reusable** across JSPs.
- **Automatic type conversion**: WC converts request parameter Strings → proper primitive types in setters.

What is a JavaBean (JB)?

A **JavaBean** is simply a **packaged, reusable Java class** that follows specific conventions so that frameworks (like JSP/Servlet containers, Spring, etc.) can easily create and manage them.

Characteristics of a JavaBean

1. **Packaged public Java class**
   - Must be in a package (e.g., beans.UserBean).
   - JSP/Servlet container loads it via <jsp:useBean>.
   - The container stores it as an **attribute** in the specified scope (page, request, session, application).

2. **Default Constructor (No-arg constructor)**
   - Mandatory in JSP Bean usage (because WC instantiates via reflection).

   Eg. public UserBean() { }

3. **Properties (state)**

- Represent **conversational state of the client** (e.g., email, password, regAmount).
- Properties must be:
  - private
  - non-static
  - non-transient

Example:

private double regAmount;

- **Getter and Setter naming convention** (strict):
- public void setRegAmount(double val) { this.regAmount = val; }
- public double getRegAmount() { return regAmount; }

Property name is derived from method name:

- Eg. setRegAmount() → property name = regAmount.
- So in JSP: <jsp:setProperty name="bean" property="regAmount" value="5000"/>

4. **Business Logic Methods**
   - Any **public methods** can be added freely.
   - Example:
   - public boolean validateUser() {…}

Lifecycle of java bean in JSP

When JSP encounters:

<jsp:useBean id="user" class="beans.UserBean" scope="session"/>

then WC does:

1. Checks if "user" exists in session → session.getAttribute("user").
2. If not found → loads class, instantiates via default constructor, stores it:
3. session.setAttribute("user", new UserBean());
4. From then on, you can call setters/getters/EL via:

5. <jsp:setProperty name="user" property="email" value="${param.email}"/>
6. ${sessionScope.user.email}

1 JSP - JavaBean actions

(a) <jsp:useBean>

Creates or retrieves a JavaBean in the given scope.

<jsp:useBean id="user" class="beans.UserBean" scope="session"/>

- **WC process**:
    1. Looks up attribute "user" in session.
    2. If found → reuse existing bean.
    3. If not found → load class, call default constructor, store under scope.
    4. UserBean u = new UserBean();
    5. session.setAttribute("user", u);

Bean class must have a **default constructor**.

(b) <jsp:setProperty>

Sets bean property values.

1. **Static value**
2. <jsp:setProperty name="user" property="email" value="a@b"/>

→ user.setEmail("a@b");

3. **Dynamic (expression or EL)**
4. <jsp:setProperty name="user" property="email" value="${param.f1}"/>

→ user.setEmail(request.getParameter("f1"));

5. **From request param directly**
6. <jsp:setProperty name="user" property="email" param="f1"/>

→ user.setEmail(request.getParameter("f1"));

7. **All matching params → properties**

8. <jsp:setProperty name="user" property="*"/>
    - For each request parameter with matching property name → WC auto calls the setter.

(c) <jsp:getProperty>

Outputs property value to client.

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

→ Equivalent EL:

${sessionScope.user.email}

2 RequestDispatcher related actions

(a) <jsp:forward>

Forwards request to another resource.

Eg. – In one.jsp , <jsp:forward page="two.jsp"/>

Equivalent servlet code:

RequestDispatcher rd = request.getRequestDispatcher("two.jsp");

rd.forward(request, response);

(b) <jsp:include>

Includes output of another resource at runtime.

Eg. – In one.jsp , <jsp:include page="two.jsp"/>

Equivalent servlet code:

RequestDispatcher rd = request.getRequestDispatcher("two.jsp");

rd.include(request, response);

This is **dynamic include** (different from the include directive).

- Included page can change independently.
- Indicates request scope

(c) <jsp:param>

Passes additional request parameters in forward or include.

```
<jsp:forward page="two.jsp">
  <jsp:param name="role" value="admin"/>
</jsp:forward>
```

→ In two.jsp, you can access request.getParameter("role").

Why JSP standard tag library (JSTL)?

- JSPs initially relied on **scriptlets** (<% ... %>), which mixed Java code with HTML → messy, hard to maintain.
- JSP standard actions (<jsp:useBean>, <jsp:setProperty>, etc.) were limited for dynamic content handling .
- **JSTL was** introduced to provide:
    - A standard tag-based alternative (no need for scriptlets).
    - Readability, maintainability, MVC compliance.

JSTL Components

JSTL consists of **four main tag libraries** (each with its own URI):

1. **Core Tags** (c): flow control, variable manipulation
   http://java.sun.com/jsp/jstl/core
   Examples: <c:set>, <c:if>, <c:forEach>, <c:choose>
2. **Formatting Tags** (fmt): internationalization, formatting dates/numbers
   http://java.sun.com/jsp/jstl/fmt
3. **SQL Tags** (sql): database queries/updates (rarely used in production, discouraged)
   http://java.sun.com/jsp/jstl/sql
4. **XML Tags** (x): parsing and transforming XML
   http://java.sun.com/jsp/jstl/xml

Steps

- Add **JSTL JAR** to runtime classpath (WEB-INF/lib or <tomcat_home>/lib):
    - JSTL 1.2 → javax.servlet.jsp.jstl-1.2.jar
    - For Tomcat 10+ (Jakarta EE), use updated versions:

    - jakarta.servlet.jsp.jstl-api-3.0.x.jar
    - jakarta.servlet.jsp.jstl-3.0.x.jar
- Import into JSP:
- <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Tag	Purpose	Syntax / Example
<c:set>	Store a value in a scope attribute	<c:set var="name" value="Madhura" scope="session"/> ${sessionScope.name} → Madhura
<c:remove>	Remove an attribute from scope	<c:remove var="name" scope="session"/>
<c:out>	Print value safely (null-safe, XSS-safe)	<c:out value="${param.city}" default="Unknown"/>
<c:if>	Conditional execution	<c:if test="${param.btn eq 'Deposit'}"> Deposit Block </c:if>
<c:choose>	Switch-like conditional	<c:choose> <c:when test="${age >= 18}">Adult</c:when> <c:otherwise>Minor</c:otherwise> </c:choose>
<c:forEach>	Loop over collections, arrays, ranges	<c:forEach var="acct" items="${sessionScope.bank.accts}"> ${acct.id} ${acct.balance} </c:forEach> <c:forEach var="i" begin="1" end="5">${i}</c:forEach>
<c:forTokens>	Loop over tokens in a string	<c:forTokens items="red,green,blue" delims="," var="color"> ${color} </c:forTokens>
<c:catch>	Exception handling	<c:catch var="err"> <%= 10/0 %> </c:catch> Error: ${err}
<c:import>	Import/include another resource (local/remote)	<c:import url="header.jsp"/> <c:import url="http://example.com"/>
<c:param>	Pass parameters with <c:import>, <c:redirect>	<c:import url="test.jsp"> <c:param name="id" value="123"/> </c:import>
<c:redirect>	Send client redirect (URL rewriting supported)	<c:redirect url="home.jsp"/> <c:redirect url="next.jsp"> <c:param name="user" value="Madhura"/></c:redirect>
<c:url>	Create a URL with session ID (URL rewriting)	<c:url var="next" value="next.jsp"/> Next

Some Examples

1. <c:out value="${param.name}"/>

<c:out value="${param.city}" default="Unknown" />

<c:out value="<h1>Hello</h1>" escapeXml="false" />

2. Setting Attributes

<c:set var="abc" value="${param.f1}" scope="session"/>

Equivalent to:

session.setAttribute("abc", request.getParameter("f1"));

2. Removing Attributes

<c:remove var="abc" scope="request"/>

Equivalent to:

request.removeAttribute("abc");

3. Conditional Execution

<c:if test="${param.btn eq 'Deposit'}">

  Deposit branch

</c:if>

<c:if test="${param.btn eq 'Withdraw'}">

  Withdraw branch

</c:if>

Equivalent to:

if ("Deposit".equals(request.getParameter("btn"))) { ... }

4. Iteration

<c:forEach var="acct"
items="${sessionScope.my_bank.acctSummary}">

 ${acct.acctID} ${acct.type} ${acct.balance}

</c:forEach>

Equivalent to:

for(Account acct :
((Bank)session.getAttribute("my_bank")).getAcctSummary()) {

```
  out.print(acct.getAcctID() + " " + acct.getType() + " " +
acct.getBalance());
}
```

5. Redirects

```
<c:redirect url="${sessionScope.my_bank.closeAccount()}"/>
```

Equivalent to:

```
response.sendRedirect(
  ((Bank)session.getAttribute("my_bank")).closeAccount()
);
```

6. URL Rewriting

```
<c:url var="nextUrl" value="next.jsp"/>
<a href="${nextUrl}">Next</a>
```

Equivalent to:

```
pageContext.setAttribute("nextUrl", response.encodeURL("next.jsp"));
```

JSTL makes JSP pages **clean, declarative, and tag-driven** instead of embedding business logic directly.