JSP

Introduction to JSP and Servlets

Problems with servlets and how JSP provides solutions for these problems:

1. Problem: Single Servlet for Both Request Processing and Response Generation

- ☑ **Problem**: In traditional servlet-based applications, a single servlet class handles both the request processing and the response generation. This leads to a monolithic structure, making the code harder to maintain and extend.
  - ✓ **Solution via JSP**: JSP separates the presentation layer from the business logic. In JSP, the HTML content is embedded directly within the page, while the server-side logic (business processing) is handled by servlets or backend classes. JSP allows developers to focus on the layout and user interface without worrying about mixing it with complex Java code.
  - ✓ **How JSP Helps**: With JSP, developers can insert dynamic content directly into the HTML using special tags (like <%= %> or <jsp:useBean>). This separation of concerns reduces the complexity of the servlet code and makes the application more modular.

2. Problem: Detailed Java Programming Knowledge Required for Development and Maintenance

- ☑ **Problem**: Since the business logic and HTML are combined in servlets, developers need to have strong Java programming skills to handle everything from data processing to HTML generation.
  - ✓ **Solution via JSP**: JSP simplifies the web development process, as it allows for HTML markup and Java code to coexist in a more intuitive way. Java logic (such as loops and conditionals) can be embedded directly within HTML code without needing to understand the intricacies of servlet programming.
  - ✓ **How JSP Helps**: Developers can use JSP tags to dynamically generate content and handle user input, reducing the need for advanced Java knowledge in the UI layer. This is particularly helpful for web designers or front-end developers who may not be familiar with Java programming.

3. Problem: Difficulty in Changing Application's Look and Feel

- ☑ **Problem**: When the presentation logic is embedded within the servlet, making changes to the UI requires recompiling and redeploying the servlet, which slows down the process.

- ✓ **Solution via JSP**: With JSP, the user interface is more flexible. Changes to the UI can be made by simply editing the JSP files, without the need to modify any backend logic or recompile the application.
- ✓ **How JSP Helps**: The JSP file is just a plain text file with embedded Java code, and changes to its content are immediately reflected when the page is refreshed. This provides a faster, more agile approach to UI changes, improving the development cycle.

4. Problem: Adding Support for New Clients Requires Recompiling the Servlet

- ☑ **Problem**: If a new client (such as a mobile device or a different browser type) needs to be supported, the servlet code must be updated, and the application needs to be recompiled and redeployed.
  - ✓ **Solution via JSP**: JSP supports dynamic content generation based on client request headers (such as the "User-Agent" for mobile browsers). Different JSP pages can be created for different client types, and these can be selected dynamically at runtime based on the client's request.
  - ✓ **How JSP Helps**: JSP allows for conditional logic to handle different client types. For example, mobile users can be served a separate layout or template, without requiring changes to the servlet code itself. This makes it easier to support multiple types of clients without constantly recompiling and redeploying the application.

5. Problem: Challenges in Using Web Page Development Tools

- ☑ **Problem**: Web development tools that simplify the creation of HTML layouts (such as Dreamweaver or other visual tools) don't integrate well with servlet-based applications. The HTML code must be manually embedded into servlet code, making it difficult to use such tools efficiently.
  - ✓ **Solution via JSP**: JSP files allow developers to separate HTML design from Java code, making it much easier to use page development tools for creating and editing the layout. Designers can focus purely on the look and feel of the page without worrying about Java logic.
  - ✓ **How JSP Helps**: By creating a distinct separation of logic and presentation, JSP files allow web designers to work with standard HTML editors or design tools. This simplifies the process, making it easier to iterate on the layout and

appearance of the website without needing to embed HTML into Java code manually.

6. Problem: Manually Embedding HTML into Servlet Code

- ☑ **Problem**: In servlet-based applications, developers often need to manually embed HTML into Java code, which is time-consuming and error-prone. Small changes in the HTML layout require updates to the Java code, leading to repetitive tasks and potential mistakes.
  - ✓ **Solution via JSP**: JSP eliminates the need for manual embedding of HTML code into Java files. Developers can create HTML templates in JSP files and insert dynamic content through tags and expressions.
  - ✓ **How JSP Helps**: With JSP, the HTML is stored directly in the page, and Java code is only used where necessary (like in <%= %> tags or JSP directives). This reduces the time spent manually integrating HTML into Java code and makes the process less error-prone.

Advantages of JSP:

1. Separation of Concerns

- ✓ **Advantage**: JSP separates the presentation layer (HTML) from the business logic (Java). This makes the development process more organized, maintainable, and scalable.
- ✓ **Why It's Important**: Web designers can focus on the user interface (UI) without dealing with complex Java logic, while Java developers can concentrate on the application's backend logic.

2. Simplified Web Development

- ✓ **Advantage**: JSP allows embedding dynamic Java code directly within the HTML, making it easier to generate dynamic content (like database results or user input) in web pages.
- ✓ **Why It's Important**: Developers can write simpler code without needing to write verbose HTML in servlets. JSP is more intuitive for web page creation, reducing development time.

3. Reusability and Maintainability

- ✓ **Advantage**: JSP supports tag libraries and custom tags, making it easy to reuse components and avoid code duplication.

- ✓ **Why It's Important**: Common functionalities (like user login, data display, etc.) can be reused across different pages using custom tags, reducing the amount of repetitive code and simplifying long-term maintenance.

4. Faster Development

- ✓ **Advantage**: JSP allows the use of pre-written libraries and custom tag libraries to streamline development. Developers don't have to manually write complex HTML or Java code, speeding up the process.
- ✓ **Why It's Important**: This is especially useful in large-scale projects, where development time and rapid prototyping are critical.

5. Dynamic Content Generation

- ✓ **Advantage**: JSP supports dynamic content generation through Java code (like loops, conditionals, etc.) embedded directly in the HTML structure.
- ✓ **Why It's Important**: Developers can create rich, dynamic web pages that can respond to user input, display content from databases, or interact with APIs without needing to write a separate servlet for each page.

6. Improved Readability

- ✓ **Advantage**: JSP code is generally more readable than servlet-based code, as it follows the natural structure of HTML with embedded Java. Java logic is embedded only where necessary using expressions, tags, or scriptlets.
- ✓ **Why It's Important**: This makes the code more approachable and easier to maintain, especially for teams where designers and developers work together.

7. Automatic Compilation

- ✓ **Advantage**: JSP files are automatically compiled by the server the first time they are accessed. After that, the compiled servlet is used, improving performance.
- ✓ **Why It's Important**: Developers do not need to manually compile the JSP code into servlets, which saves time and simplifies the deployment process.

8. Cross-platform Support

- ✓ **Advantage**: Since JSP runs on any platform that supports a Java Servlet container (like Apache Tomcat, Jetty, etc.), it provides cross-platform support.
- ✓ **Why It's Important**: This allows JSP-based applications to run on various operating systems and hardware, providing flexibility for deployment.

9. Support for Expression Language (EL)

- ✓ **Advantage**: JSP provides built-in support for Expression Language (EL), which allows easier access to JavaBeans, request parameters, session attributes, and other objects.
- ✓ **Why It's Important**: EL makes the code more concise and easier to write, reducing the need for scriptlets, thus improving readability and maintainability.

10. Integration with JavaBeans and Custom Tags

- ✓ **Advantage**: JSP allows the integration of JavaBeans and custom tags, providing flexibility in separating logic and presentation. Custom tags allow you to define reusable components that handle specific tasks (like database querying or displaying navigation links).
- ✓ **Why It's Important**: By using JavaBeans and custom tags, the complexity of the code is reduced and the application is more modular and reusable.

11. Support for Multi-language Web Applications

- ✓ **Advantage**: JSP can easily support multi-language websites through the use of resource bundles and localization tags.
- ✓ **Why It's Important**: This makes it simpler to develop internationalized applications that support different languages and regions.

12. Error Handling and Debugging

- ✓ **Advantage**: JSP provides mechanisms for centralized error handling using error pages. You can define custom error pages for specific exceptions or HTTP error codes.
- ✓ **Why It's Important**: This feature helps to gracefully handle errors in production environments and provides a better user experience by displaying custom error messages.

13. Scalability

- ✓ **Advantage**: JSP allows the easy scaling of applications due to its clean architecture and modularity. You can add additional functionality with minimal impact on existing code.
- ✓ **Why It's Important**: As web applications grow, JSP's modular nature allows developers to add new features without rewriting existing code, which is crucial for long-term scalability.

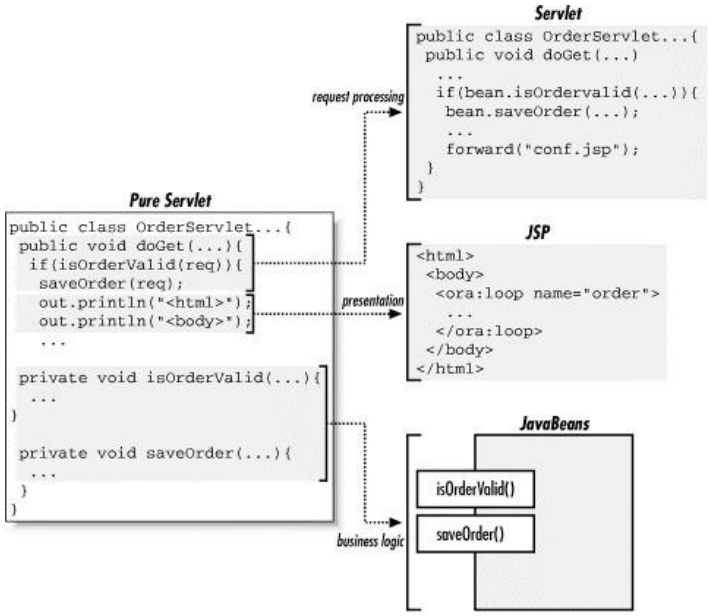
14. Tag Libraries and Rich Ecosystem

- ✓ **Advantage**: JSP has a rich ecosystem of libraries (such as JSTL—JavaServer Pages Standard Tag Library) that provide predefined tags for common tasks like iteration, conditionals, database access, etc.

- ✓ **Why It's Important**: Developers can leverage these libraries to implement common functionality without reinventing the wheel, speeding up development and reducing the chance of bugs.

15. Easy Integration with Backend Services

- ✓ **Advantage**: JSP can easily integrate with Java backend services, such as databases (JDBC), web services (RESTful APIs), and enterprise solutions (EJB).
- ✓ **Why It's Important**: This integration is crucial for modern web applications, where dynamic data and interactions with backend systems are essential for providing real-time functionality.



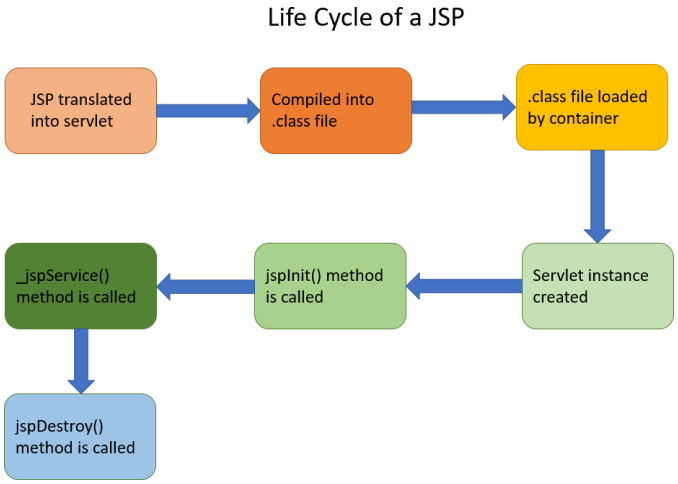
Client-side scripting vs Server-side scripting:

Aspect	Client-side Scripting	Server-side Scripting
Execution Location	Code is executed on the **client's browser** after the webpage is loaded.	Code is executed on the **server** before content is sent to the client.
Languages Used	JavaScript, HTML5, CSS3, jQuery, AngularJS, etc.	Java, PHP, Python, Ruby, ASP.NET, Node.js, etc.
Speed	Faster user interactions because no server requests are required for actions like form validation, animations, etc.	Slower since every user request requires communication with the server.
User Interaction	Immediate interaction, e.g., form validation, dynamic content updates.	Typically involves page reloads or additional requests to the server.
Resource Usage	Lighter on server resources as execution happens on the client-side.	Puts more load on the server because it processes requests and sends the results.

Security	Less secure since the code is exposed to the client, allowing users to manipulate it.	More secure as the code runs on the server, and the client only receives the output.
Data Access	Cannot directly access databases or handle sensitive data without server interaction.	Can access databases, manage sessions, and handle sensitive data securely.
Error Handling	Errors are visible to the user and can be handled via JavaScript (e.g., alerts).	Errors are handled server-side, often in logs or custom error pages.
Use Cases	Suitable for tasks that don't require server interaction like UI effects, form validation, and client-side logic.	Suitable for tasks that require database access, authentication, and dynamic content generation.
Examples in JSP	JavaScript within JSP pages for dynamic content (e.g., form validation before submission).	JSP pages generating dynamic content from backend logic and database queries.

JSP Lifecycle and Processing:

JSP life cycle:



Explanation of phases:

1. JSP Translated into Servlet (Initial Translation Phase)

- ☑ **Explanation**: When a JSP file is first requested, the web container (like Tomcat) translates the JSP code into a Java servlet. During this phase, all the HTML and Java embedded in the JSP are processed and converted into Java code. The dynamic parts of the JSP (like scriptlets and expressions) are transformed into Java code that will later handle the request/response.
- ☑ **Example**: Consider the following JSP file hello.jsp:

```jsp
<html>
  <body>
    <h1>Hello, <%= request.getParameter("name") %>!</h1>
  </body>
</html>
```

The JSP code is translated into a servlet that contains logic for generating the response dynamically using the request.getParameter("name") expression.

Generated Servlet Code:

```java
public class HelloServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
  IOException {
    String name = request.getParameter("name");
    response.getWriter().write("<html><body><h1>Hello, " + name + "!</h1></body></html>");
  }
}
```

The translation happens automatically when the JSP is requested for the first time.

2. Compiled into .class File

- ☑ **Explanation**: After the JSP is translated into a servlet, the container compiles the generated Java servlet into bytecode and creates a .class file. This compiled class is the actual servlet that will handle client requests.
- ☑ **Example**: For the above JSP, the generated servlet HelloServlet.java is compiled into HelloServlet.class by the container's compiler (like javac). This class contains the servlet logic to handle requests and generate responses.

3. .class File Loaded by Container

- ☑ **Explanation**: Once the servlet class is compiled, the container loads the compiled .class file into memory. This happens during the first request to the JSP or when the web application starts up (if the JSP is already used).
- ☑ **Example**: The container loads HelloServlet.class into memory so that it can be used to handle future requests for the hello.jsp page.

4. Servlet Instance Created

- ☑ **Explanation**: The container creates an instance of the servlet class. This instance is responsible for handling client requests. It will invoke the init() method if needed, which is used for initialization.
- ☑ **Example**: When the container creates the HelloServlet instance, it allocates memory for it and prepares it to process incoming requests.

5. jspInit() Method Called

- ☑ **Explanation**: The jspInit() method is called during the servlet initialization phase. It is typically used to perform initialization tasks such as setting up resources (e.g., database connections). This method is called only once, when the servlet is loaded into memory.

- ☑ **Example**: Suppose the servlet needs to connect to a database or set up any initial configuration. The jspInit() method would be used to perform these tasks.

```
public void jspInit() {
  // Initialize resources or configuration
  System.out.println("JSP Initialized!");
}
```

6. _jspService() Method Called

- ☑ **Explanation**: The _jspService() method is called for every request made to the JSP. This is where the main request processing happens. The method handles requests and generates responses, typically by calling methods like doGet() or doPost().
- ☑ **Example**: For the hello.jsp example, the _jspService() method will handle the request for the user's name, process it, and generate the response.

7. jspDestroy() Method Called

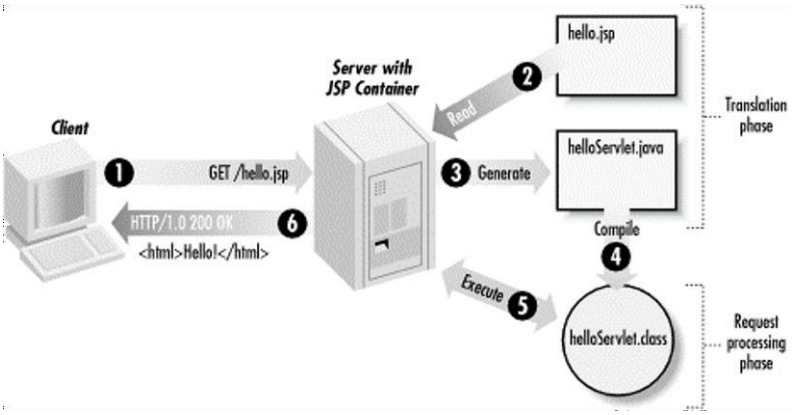
- ☑ **Explanation**: The jspDestroy() method is called when the servlet is about to be destroyed. This happens when the web application is stopped or when the container decides to unload the servlet. This method is used for cleanup tasks, such as releasing resources.
- ☑ **Example**: If the HelloServlet opened a database connection or held other resources, the jspDestroy() method would be used to close these resources before the servlet is destroyed.

```
public void jspDestroy() {
  // Release resources, close database connections, etc.
  System.out.println("JSP Destroyed!");
}
```

Summary of JSP Processing Phases:

Phase	Description	Example
1. JSP Translated into Servlet	JSP is converted into a Java servlet by the container.	JSP code like hello.jsp is turned into a servlet.
2. Compiled into .class File	The servlet is compiled into a .class file by the container.	HelloServlet.java is compiled into HelloServlet.class.
3. .class File Loaded by Container	The compiled .class file is loaded into memory.	The servlet is now available for request handling.
4. Servlet Instance Created	The container creates an instance of the servlet.	A new instance of HelloServlet is created.
5. jspInit() Method Called	Initializes the servlet and any required resources.	Database connections are opened in jspInit().
6. _jspService() Method Called	Processes the request and generates the response.	Processes the name parameter and sends a response.
7. jspDestroy() Method Called	Cleans up resources before the servlet is destroyed.	Closes any database connections in jspDestroy().

JSP page translation and processing phases:



Step 1: Client sends a request for a JSP page.

Step 2: The server reads the JSP page.

Step 3: The JSP container generates a servlet (helloServlet.java).

Step 4: The servlet is compiled (helloServlet.class).

Step 5: The servlet is executed to handle the request and generate a response.

Step 6: The response (HTML) is sent to the client.

1. Client Sends Request (HTTP GET Request):

- ☑ **Description**: The user requests a JSP page (e.g., hello.jsp) from the browser using an HTTP GET request.
- ☑ **Diagram Step**: The **client** sends the GET request to the server for the JSP page (hello.jsp).

Example:

- ☑ A user enters http://localhost/hello.jsp in their browser.

2. JSP Container Reads the JSP Page:

- ☑ **Description**: The JSP container on the server reads the contents of the requested JSP file (hello.jsp).
- ☑ **Diagram Step**: The server with the **JSP container** reads the hello.jsp file.

Explanation: The container prepares the JSP for translation. It parses the JSP file and separates the static HTML content from the dynamic Java code embedded in the JSP.

3. JSP Container Generates Java Servlet (Translation Phase):

- ☑ **Description**: The JSP container converts the JSP page into a corresponding Java servlet class (helloServlet.java). This process is referred to as **Translation Phase**.
- ☑ **Diagram Step**: The **JSP container generates** the servlet code (helloServlet.java).

Explanation: The dynamic parts of the JSP, such as <% %> (scriptlets) and <%= %> (expressions), are converted into Java code inside the generated servlet. The static HTML content is maintained as part of the generated servlet.

4. Compile the Java Servlet (Compilation Phase):

- ☑ **Description**: The generated servlet Java file (helloServlet.java) is compiled into bytecode (helloServlet.class).
- ☑ **Diagram Step**: The **JSP container compiles** the Java servlet to produce the class file (helloServlet.class).

Explanation: This step is necessary to convert the Java source code into bytecode that the JVM can execute. The compilation is done automatically by the server during the first request or when the JSP is modified.

5. Execution of the Servlet (Request Processing Phase):

- ☑ **Description**: After compilation, the JSP container executes the generated servlet (helloServlet.class) to process the user request and generate the response.
- ☑ **Diagram Step**: The **JSP container executes** the servlet (helloServlet.class) to handle the request.

Explanation: The servlet processes the user request, executes any embedded Java code (such as generating dynamic content), and sends the response (HTML) back to the client. This is the **request processing phase**.

6. Response Sent to Client:

- ☑ **Description**: The server sends the HTTP response containing the HTML (or other content) to the client (browser).
- ☑ **Diagram Step**: The **client** receives the HTTP response containing the HTML output (<html>Hello</html>).

Example: The browser displays the message Hello returned by the servlet, which was generated from the JSP page.

Key Points Based on the Diagram:

1. **Translation Phase**:
   - ☑ This phase occurs when the JSP page is first requested or modified. The JSP container converts the JSP page into a servlet (helloServlet.java) and compiles it into a .class file.
   - ☑ The **first request** for a JSP page may take a bit longer because the translation process occurs during that request. Subsequent requests are faster as the .class file is used directly.

2. **Request Processing Phase**:
   - ☑ Once the JSP page is translated and compiled, the servlet is executed for each request to generate the response dynamically. If the JSP page is unchanged, the container will directly execute the already compiled servlet (helloServlet.class).
3. **Precompilation**:
   - ☑ To avoid the initial delay caused by translation during the first request, you can precompile the JSP files. This step ensures that the servlet is already compiled and ready for execution.
4. **Changes in the JSP**:
   - ☑ If the JSP page is modified, it goes through the translation phase again before entering the request processing phase. This ensures that the servlet reflects any changes made in the JSP.

Overridable Life Cycle Methods in JSP:

In the JSP (JavaServer Pages) lifecycle, there are specific methods related to the underlying servlet that can be overridden. These methods are part of the generated servlet's lifecycle, and developers may override them to customize certain behaviors. Here are the main life cycle methods of JSP that can be overridden:

1. jspInit() Method

- ☑ **Description**: The jspInit() method is called when the servlet is initialized, which happens only once when the servlet is loaded into memory for the first time. This method is used for any initialization that needs to happen before the servlet starts handling requests.
- ☑ **Override**: You can override the jspInit() method to perform initialization tasks such as opening database connections, loading configurations, or preparing resources.

```
public void jspInit() {
    // Initialize resources like database connections or loggers
    System.out.println("JSP Initialized!");
}
```

2. _jspService() Method

- ☑ **Description**: The _jspService() method is called for each HTTP request made to the JSP. This is the core method that processes the client's request and generates the response. It is automatically called by the container whenever a request is made to the JSP.

☑ **Override**: Developers typically don't override this method directly, as it is generated by the container. However, for custom JSP servlets or if you need to modify how requests are processed, you can override it.

```
public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    // Request handling logic, process parameters, and generate a response
    String name = request.getParameter("name");
    response.getWriter().write("<html><body><h1>Hello, " + name + "!</h1></body></html>");
}
```

3. jspDestroy() Method

☑ **Description**: The jspDestroy() method is called when the servlet is being destroyed or unloaded, which typically occurs when the web application is stopped or the servlet is removed from memory. This method is used for cleanup tasks, such as releasing resources.

☑ **Override**: You can override jspDestroy() to close resources like database connections, file handles, or other resources that need to be released when the servlet is destroyed.

```
public void jspDestroy() {
    // Clean up resources like closing database connections
    System.out.println("JSP Destroyed!");
}
```

Summary of Overridable Life Cycle Methods in JSP:

Method	Description	When It's Called
jspInit()	Used for initialization tasks (like opening resources, database connections).	Called once when the servlet is loaded into memory for the first time.
_jspService()	Handles each request from the client and generates the response.	Called each time a request is made to the JSP.
jspDestroy()	Used for cleanup tasks (like closing database connections, releasing resources).	Called when the servlet is about to be destroyed or the web application is stopped.

Anatomy of a JSP Page

```
<%@ page language="java" contentType="text/html" %>                    — JSP element
<html>                                                                 — template text
 <body bgcolor="white">
<jsp:useBean                                                           — JSP element
 id="userInfo"
 class="com.ora.jsp.beans.userinfo.UserInfoBean">
 <jsp:setProperty name="userInfo" property="*"/>
</jsp:useBean>
  The following information was saved:                                 — template text
  <ul>
   <li>User Name:
    <jsp:getProperty name="userInfo"                                   — JSP element
     property="userName"/>
   <li>Email Address:                                                  — template text
    <jsp:getProperty name="userInfo"                                   — JSP element
    property="emailAddr"/>
  </ul>                                                                — template text
 </body>
</html>
```

The **anatomy of a JSP page** consists of several components that define its structure and functionality. A JSP (Java Server Pages) page is a **combination of static content** (like HTML, CSS, JavaScript) and **dynamic content** generated using Java-based JSP elements. The anatomy of a JSP page includes:

1. Directives (<%@ %>**)**

- These provide **global information** about the page to the JSP engine. In the directives we can **import packages**, **define error handling pages** or the **session** information of the JSP page.
- Example:
  - ```
    <%@ page language="java" contentType="text/html" %>
    ```
- **Types of directives**:
  - page: Defines attributes related to the entire page.
  - include: Includes a file at compile time.
  - taglib: Declares a custom tag library.

2. Declarations (<%! %>**)**

- Used to **declare variables and methods** that can be used anywhere in the JSP page. This tag is used for defining the **functions** and **variables** to be used in the JSP.
- Example:

```
<%! int count = 0; %>
```

Declaring variables and methods using <%! %>:

Variables and methods declared within <%! %> are available throughout the page and can be used in scriptlets or other parts of the page.

Example:

3. Scriptlets (<% %>**)**

- Contain Java code that gets executed when the page is requested. In this tag we can **insert** any **amount** of **valid java code** and **these codes** are **placed** in **_jspService** method by the JSP engine.
- Example:

```
<% count++; %>
```

- A JSP scriptlet allows Java code to be embedded directly within the HTML of a JSP page. The code in a scriptlet is executed when the page is requested and it is inserted into the response output.
```

- Scriptlets are written inside <% %> tags.

Example of a JSP scriptlet:

```
<%
    int a = 10;
    int b = 20;
    int sum = a + b;
    out.println("The sum is: " + sum);
%>
```

- The above scriptlet declares variables a, b, and sum, computes the sum, and uses the out object to print the result to the response page.

Comparison between JSP Scriptlet tag and declaration tag.

JSP Scriptlet Tag:

- Syntax: <% ... %>
- Used to embed Java code within a JSP page.
- Code is executed each time the page is requested.
- Cannot declare methods or member variables.
- Example:

```
<%
    int sum = 10 + 20;
    out.println("Sum: " + sum);
%>
```

JSP Declaration Tag:

- Syntax: <%! ... %>
- Used to declare variables and methods that will be available for the entire JSP page.
- Declared variables and methods are initialized once when the page is compiled and are available across the entire page.
- Example:

```
<%!
    int counter = 0;
    public void incrementCounter() {
        counter++;
    }
%>
```

4. Expressions (<%= %>)

- Used to insert **dynamic values** into the HTML output. We can use this tag to output any data on the generated page. These data are automatically converted to string and printed on the output stream.

- Example:

```
<p>Current Count: <%= count %></p>
```

JSP literals:

JSP **literals** are fixed values used in JSP expressions and scriptlets. They are directly represented in the code without requiring computation.

Types of JSP Literals

1. **Integer Literals**:
   - Whole numbers without decimal points.
   - Example:

     <% int x = 100; %>
2. **Floating-Point Literals**:
   - Decimal values or numbers in scientific notation.
   - Example:

     <% double pi = 3.14159; %>
3. **Boolean Literals**:
   - Represents true or false.
   - Example:

     <% boolean isUserLoggedIn = true; %>
4. **Character Literals**:
   - A single character enclosed in **single quotes (')**.
   - Example:

     <% char grade = 'A'; %>
5. **String Literals**:
   - A sequence of characters enclosed in **double quotes (")**.
   - Example:

     <% String message = "Welcome to JSP"; %>
6. **Null Literal**:
   - Represents the absence of a value.
   - Example:

     <% String str = null; %>
7. **Escape Sequences in String Literals**:
   - \n – Newline
   - \t – Tab

- \" – Double Quote
- \' – Single Quote
- \\ – Backslash
- Example:

  <% String path = "C:\\Program Files\\Java"; %>

5. JSP Elements

- Special tags to **simplify dynamic content generation**.
- Example

```
<jsp:useBean id="userInfo" class="com.example.UserBean" />
<jsp:getProperty name="userInfo" property="username" />
```

6. Template Text

- The **static content** (HTML, XML, WML, etc.) that remains unchanged.
- Example:

```
<h1>Welcome to JSP</h1>
```

Identifying the state of the program if a semicolon (;) is used in a JSP expression.

In JavaServer Pages (JSP), a semicolon (;) is used in expressions to terminate individual statements, much like in Java. A JSP expression is a snippet of Java code enclosed in <%= %> tags, and it can be used to output values directly into the HTML response. When you use a semicolon within a JSP expression, it has a specific role related to the syntax of Java code.

Understanding the Role of the Semicolon in JSP Expressions:

1. **Separation of Statements:** In Java, each statement ends with a semicolon. Similarly, in a JSP expression, the semicolon is used to separate individual statements if there are multiple statements inside the <%= %> tag.
2. **JSP Expression Syntax:** A JSP expression is written inside <%= %> tags. Any valid Java code inside these tags will be executed, and the result will be inserted into the HTML response.

State of the Program When Semicolon is Used:

- The semicolon in a JSP expression marks the end of a statement.
- If multiple statements are present in the JSP expression, semicolons are used to separate them.
- However, JSP expressions **cannot** contain multiple statements; they should only contain a single expression. If you attempt to add more than one statement in a JSP expression, the code might result in a syntax error or an unexpected output.

Example:

JSP Expression (Incorrect Usage)

JSP Expression (Correct Usage)

JSP Scriptlet

JSP Declaration

Summary:

- **JSP Expression (`<%= %>`)**: This is used to output the result of a Java expression. A semicolon **should not** be used for statements or variable declarations within it.
- **JSP Scriptlet (`<% %>`)**: This allows you to write multiple Java statements and use semicolons to terminate them.
- **JSP Declaration (`<%! %>`)**: This is used for declaring methods or variables with semicolons ending each declaration.

In conclusion, the semicolon within a JSP expression is used in Java syntax to terminate statements, but only a single statement can be evaluated in a JSP expression. If you need multiple statements, use a scriptlet instead of an expression.

7. Implicit Objects

- JSP provides **predefined objects** that do not need to be declared explicitly.
- Examples: request, response, session, application, out, etc.

Different Implicit Objects:

i. The request Object:

The **request** object is an instance of a javax.servlet.http.HttpServletRequest object. Each time a client requests a page, the JSP engine creates a new object to represent that request. The **request** object provides methods to get the HTTP header information, including form data, cookies, HTTP methods, etc.

ii. The response Object:

The **response** object is an instance of a javax.servlet.http.HttpServletResponse object. Just as the server creates the **request** object, it also creates an object to represent the response to the client. The **response** object also defines the interfaces that deal with creating new HTTP headers. Through this object, the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

iii. The out Object:

The **out** implicit object is an instance of a javax.servlet.jsp.JspWriter object and is used to send content in a response. The initial JspWriter object is instantiated differently depending

on whether the page is buffered or not. Buffering can be easily turned off by using the buffered='false' attribute of the page directive. The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions.

iv. The session Object:

The **session** object is an instance of javax.servlet.http.HttpSession and behaves exactly the same way that session objects behave under Java Servlets. The **session** object is used to track the client session between client requests.

v. The application Object:

The **application** object is a direct wrapper around the ServletContext object for the generated Servlet and, in reality, an instance of a javax.servlet.ServletContext object. This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method.

vi. The config Object:

The **config** object is an instantiation of javax.servlet.ServletConfig and is a direct wrapper around the ServletConfig object for the generated servlet. This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters, such as the paths or file locations, etc.

vii. The pageContext Object:

- The **pageContext** object is an instance of a javax.servlet.jsp.PageContext object. The **pageContext** object is used to represent the entire JSP page. This object is intended as a means to access information about the page while avoiding most of the implementation details. This object stores references to the **request** and **response** objects for each request. The **application**, **config**, **session**, and **out** objects are derived by accessing attributes of this object.
- The **pageContext** object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope. The PageContext class defines several fields, including PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, and APPLICATION_SCOPE, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the javax.servlet.jsp.JspContext class.

viii. The page Object:

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The **page** object is really a direct synonym for this object.

ix. The exception Object:

The **exception** object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

Explanation with examples for each:

1. request

- **Description**: Represents the HttpServletRequest object. It contains the information the client sends to the server when making a request. It is used to access request headers, form data, query parameters, HTTP methods (GET, POST), etc.
- **Methods**:
  - getParameter(): Retrieves form data or query parameters.
  - getRemoteAddr(): Retrieves the IP address of the client.
  - getMethod(): Retrieves the HTTP method used for the request (e.g., GET, POST).
- **Example**:

```
<h3>Request Information:</h3>
<p>Your IP Address: <%= request.getRemoteAddr() %></p>
<p>Request Method: <%= request.getMethod() %></p>
```

2. response

- **Description**: Represents the HttpServletResponse object. It allows you to modify the response to the client, including setting headers, sending cookies, and sending data back to the client.
- **Methods**:
  - setHeader(): Sets HTTP headers in the response.
  - setStatus(): Sets the HTTP status code for the response.
- **Example**:

```
<%
    response.setHeader("Custom-Header", "This is a custom header");
%>
<p>Custom HTTP Header has been set.</p>
```

3. out

- **Description**: Represents the JspWriter object. It is used to send content to the client's response. It is used to display output on the page.
- **Methods**:
  - println(): Prints a line of text to the client.
  - print(): Prints a text without moving to the next line.
- **Example**:

```
<h3>Current Date and Time:</h3>
<p><%= new java.util.Date() %></p>
```

4. session

- **Description**: Represents the HttpSession object. It allows you to store data that is specific to a user session. This object is used to track the state of a user across multiple requests.
- **Methods**:
  - setAttribute(): Stores an attribute in the session.
  - getAttribute(): Retrieves an attribute from the session.
- **Example**:

```
<%
    session.setAttribute("user", "John Doe");
%>
<p>Session user: <%= session.getAttribute("user") %></p>
```

5. application

- **Description**: Represents the ServletContext object. It allows you to store application-wide data that is shared across all users and requests within the application.
- **Methods**:
  - setAttribute(): Sets an attribute in the application context.
  - getAttribute(): Gets an attribute from the application context.
- **Example**:

```
<%
    application.setAttribute("appName", "JSP Example App");
%>
<p>Application-wide data (App Name): <%= application.getAttribute("appName") %></p>
```

6. config

- **Description**: Represents the ServletConfig object. It allows access to the servlet's initialization parameters (like paths or file locations). It is available only to the current servlet.

- **Methods**:
  - getInitParameter(): Retrieves an initialization parameter from the servlet.
- **Example**:

```
<%
    String initParam = config.getInitParameter("configParam");
%>
<p>Servlet Initialization Parameter: <%= initParam %></p>
```

7. pageContext

- **Description**: Represents the PageContext object. It encapsulates the various implicit objects and allows easy access to them. It provides access to page-specific data and scope information.
- **Methods**:
  - setAttribute(): Sets an attribute in the page context.
  - getAttribute(): Retrieves an attribute from the page context.
- **Example**:

```
<%
    pageContext.setAttribute("pageInfo", "Page-specific data");
%>
<p>Page scope data: <%= pageContext.getAttribute("pageInfo") %></p>
```

8. page

- **Description**: Represents the current JSP page instance. It is a synonym for this in the context of the page.
- **Methods**:
  - This object refers to the current JSP page instance, and you can use it as you would use this in a Java class.
- **Example**:

```
<h3>Page Information:</h3>
<p>Current page instance: <%= this %></p>
```

9. exception

- **Description**: Represents an exception object that is thrown during the execution of the page. It is typically used in error pages to handle exceptions thrown in the previous request.
- **Methods**:
  - getMessage(): Retrieves the error message associated with the exception.
  - printStackTrace(): Prints the stack trace of the exception.

- **Example**:

```
<%
    try {
        int result = 10 / 0; // This will throw an exception
    } catch (Exception e) {
        exception = e; // Using the 'exception' implicit object to catch the error
    }
%>
<p>Error Message: <%= exception.getMessage() %></p>
```

Summary Table:

Implicit Object	Description
Request	Represents the HTTP request made by the client. Used to access request data, headers, form data, cookies, and HTTP methods.
response	Represents the HTTP response to be sent to the client. Used to set HTTP headers, cookies, status codes, etc.
out	An instance of JspWriter, used to send content to the client's response. Can handle buffering and throws IOExceptions.
session	Tracks client sessions between requests using HttpSession.
application	Represents the entire web application as an instance of ServletContext, accessible throughout the lifecycle of the JSP page.
config	Provides access to the Servlet or JSP engine's initialization parameters, such as paths or file locations.
pageContext	Represents the entire JSP page, providing access to request, response, application, session, and out objects. Includes directives and scope information.
page	A reference to the current JSP page, synonymous with the this object.
exception	A wrapper around the exception thrown in the previous page, often used for error handling and generating appropriate responses.

Anatomy of JSP with a suitable example.

Example-1:

Example-2:

Implicit Objects Example:

Attributes on page directives:

The **page directive (<%@ page %>)** is used to provide **global settings** for a JSP page. It allows defining **properties** like language, buffer size, session control, error handling, etc.

Common Attributes of Page Directive:

Attribute	Description	Example

Language	Defines the programming language (default is Java).	<%@ page language="java" %>
contentType	Specifies the MIME type of the response.	<%@ page contentType="text/html" %>
import	Imports Java classes.	<%@ page import="java.util.*, java.io.*" %>
session	Enables/disables session tracking.	<%@ page session="true" %>
buffer	Sets the buffer size for output stream.	<%@ page buffer="8kb" %>
autoFlush	Determines if the buffer should auto-flush.	<%@ page autoFlush="true" %>
isThreadSafe	Defines if multiple requests can be processed simultaneously.	<%@ page isThreadSafe="false" %>
errorPage	Specifies a page to handle errors.	<%@ page errorPage="error.jsp" %>
isErrorPage	Defines if the page is an error page.	<%@ page isErrorPage="true" %>
extends	Allows extending a Java class.	<%@ page extends="com.example.MyServlet" %>

Detailed Explanation of Each Attribute with a Use Case

1. language **Attribute**

- Defines the **scripting language** used in the JSP file.
- Default is **Java**, so this is rarely changed.
- Example:

```
<%@ page language="java" %>
```

2. contentType **Attribute**

- Specifies the **MIME type** of the response (like text/html, application/json, etc.).
- Can also define **character encoding**.
- Example:

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

3. import **Attribute**

- Used to **import Java classes** into the JSP file.
- Works similarly to the import statement in Java.
- Example:

```
<%@ page import="java.util.Date" %>
<p>Current Date: <%= new Date() %></p>
```

4. session **Attribute**

- Enables (true) or disables (false) **session tracking**.
- If session="false", you **cannot** use session implicit object.

- Example:

```
<%@ page session="true" %>
<p>Session ID: <%= session.getId() %></p>
```

5. buffer **Attribute**

- Defines the **size of the output buffer** before content is sent to the browser.
- Example:

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

6. autoFlush **Attribute**

- Controls whether **buffered output is flushed automatically** when full.
- true: Flushes automatically.
- false: Throws an exception if the buffer overflows.
- Example:

```
<%@ page buffer="8kb" autoFlush="true" %>
```

7. isThreadSafe **Attribute**

- Defines whether the JSP page can **handle multiple requests concurrently**.
- true (default): Allows multiple threads.
- false: Ensures **single-threaded execution**.
- Example:

```
<%@ page isThreadSafe="false" %>
```

8. errorPage **Attribute**

- Specifies a **JSP page** to handle errors.
- Example:

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

9. isErrorPage **Attribute**

- Defines whether this page is an **error handler**.
- If true, it allows using the exception implicit object.
- Example:

```
<%@ page isErrorPage="true" %>
<p>Error Occurred: <%= exception.getMessage() %></p>
```

10. extends **Attribute**

- Allows extending a **Java class** instead of the default JSP servlet.
- Used in advanced JSP applications.

- Example:

```
<%@ page extends="com.example.MyCustomServlet" %>
```

Attributes Example:

Action tags used in JSP with suitable examples:

JSP Action Tags: JSP action tags provide functionality to access various server-side resources and components. These tags are not Java code but are used for specific tasks.

1. **jsp:include**: Includes another page during request processing (inserts content of one page into another).

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

2. **jsp:forward**: Forwards the request to another resource.

```
<jsp:forward page="newPage.jsp" />
```

3. **jsp:param**: Used in conjunction with <jsp:include> and <jsp:forward> to pass parameters.

```
<jsp:include page="nextPage.jsp">
    <jsp:param name="user" value="John" />
</jsp:include>
```

4. **jsp:useBean**: Instantiates or references a JavaBean component in a JSP page.

```
<jsp:useBean id="userBean" class="com.example.UserBean" scope="session" />
```

5. **jsp:setProperty**: Sets properties of a JavaBean.

```
<jsp:setProperty name="userBean" property="name" value="John" />
```

6. **jsp:getProperty**: Retrieves the value of a JavaBean property.

```
<jsp:getProperty name="userBean" property="name" />
```

7. **jsp:plugin**: Embeds an applet or a JavaBean in a JSP page.

```
<jsp:plugin type="applet" code="MyApplet.class" width="200" height="200" />
```

ActionTags Example:

Various scope values of JSP action tags:

In JSP, action tags that work with JavaBeans (<jsp:useBean>, <jsp:setProperty>, and <jsp:getProperty>) support different scope values that determine the lifespan and accessibility of the bean. The four possible scope values are:

1. page Scope (Default)

- The bean is available only within the current JSP page.

- It is not accessible in other pages or requests.
- The bean is created when the page is requested and destroyed when the response is sent.

Example:

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

2. request Scope

- The bean is available throughout the entire request lifecycle.
- It can be accessed by multiple JSP pages (via RequestDispatcher.forward() or <jsp:forward>).
- The bean is destroyed once the request is completed.

Example:

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

3. session Scope

- The bean is available throughout a user's session.
- It can be accessed across multiple JSP pages and requests from the same user.
- The bean is destroyed when the session expires or is invalidated.

Example:

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

4. application Scope

- The bean is available to all JSP pages and users within the entire application.
- It is created once and remains active until the application is stopped or restarted.
- The bean is destroyed when the server shuts down.

Example:

```
<jsp:useBean id="config" class="com.example.Config" scope="application"/>
```

Summary of JSP Scopes:

Scope	Lifetime	Accessibility	Stored In
Page	Single JSP page only	Current page	PageContext
request	Current HTTP request	Forwarded JSPs & servlets	HttpServletRequest
session	User session	Multiple JSPs & servlets	HttpSession
application	Entire web app	All users & JSPs	ServletContext

Data Handling in JSP

Illustrating the methods used in reading data from a form using JSP.

Reading Data from a Form Using JSP

In JSP, we can read data from an HTML form using the request.getParameter() method. The form data is sent to the JSP page using either the GET or POST method, and JSP retrieves the submitted values using the HttpServletRequest object.

Example:

index.html

process.jsp

Single code:

Explanation

1. **HTML Form (index.html)**
    - The form collects user input for userName and userEmail.
    - The form uses the POST method to send data securely.
    - The action="process.jsp" specifies that the form will be processed by process.jsp.
2. **Processing JSP (process.jsp)**
    - Retrieves form data using request.getParameter("parameter_name").
    - The values are stored in String variables.
    - The retrieved values are displayed in an HTML response.

Code snippet to include the results of another page:

☑ Approach 1: Using <jsp:include> (Dynamic Inclusion)

- This **includes the response** of another JSP file at runtime.
- The included file **can change dynamically** based on the request.

Example: Displaying a Header from Another JSP File

Step 1: Create header.jsp (Page to Include)

```
<h2>Welcome to My Website</h2>
<hr>
```

Step 2: Create home.jsp (Main Page)

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>JSP Include Example</title>
</head>
<body>

    <!-- Including header.jsp dynamically -->
    <jsp:include page="header.jsp" />

    <h3>Main Content</h3>
    <p>This is the main page content.</p>

</body>
</html>
```

Output When home.jsp is Accessed

```
Welcome to My Website
---------------------------------
Main Content
This is the main page content.
```

How <jsp:include> Works?

- **Executes header.jsp first**, then includes its output inside home.jsp.
- Any **changes in header.jsp** are **automatically reflected** in home.jsp.
- **Best for dynamic content**, such as including a header, footer, or sidebar.

Approach 2: Using @include **(Static Inclusion)**

- This **directly copies the content** of another JSP file at compile time.
- The included file **must be known at design time**.

Example: Using @include

```
<%@ include file="header.jsp" %>
<h3>Main Content</h3>
<p>This is the main page content.</p>
```

Key Differences

Feature	<jsp:include> (Dynamic)	@include (Static)
Inclusion Type	At **runtime**	At **compile time**
Changes in File	**Reflected immediately**	Requires **recompilation**
Best Use Case	**Dynamic content** (e.g., user details, database results)	**Static content** (e.g., header, footer)

Conclusion

- Use **<jsp:include>** when you need **dynamic** inclusion.
- Use **@include** when including **static** content that does not change often.

Forwarding the request from a JSP page to the servlet:

1. Request Dispatcher

What is RequestDispatcher in JSP?

RequestDispatcher is an interface in Java **used to forward or include** requests from one JSP/Servlet to another **without changing the URL**.

It is mainly used for:

1. **Forwarding Requests** → Transfers control to another JSP/Servlet **on the server side**.
2. **Including Responses** → Includes output from another JSP/Servlet into the current page.

Types of Request Dispatching:

Method	Usage	Request Data Passed?	Changes URL?
forward(request, response)	Forwards request from one JSP/Servlet to another.	☑ Yes	✗ No
include(request, response)	Includes another JSP/Servlet's output into the current page.	☑ Yes	✗ No

1. Request Forwarding Using RequestDispatcher

When a request is forwarded, the **control is transferred** to another JSP/Servlet **on the server side** without changing the URL in the browser.

Example:

input.jsp

output.jsp

2. Including Another JSP File Using RequestDispatcher

Instead of forwarding, we can **include another JSP's response** within a page.

Example: Including header.jsp Inside main.jsp

header.jsp

main.jsp

Differences Between Forward & Include:

Feature	forward(request, response)	include(request, response)
Purpose	Redirects request to another JSP/Servlet	Embeds another JSP's response
Request Data Passed?	☑ Yes	☑ Yes

Changes URL?	❌ **No**	❌ **No**
Use Case	**Processing logic in another page**	**Reusing common UI components**

Conclusion:

☑ **Use forward()** → When you want to **redirect processing** to another JSP/Servlet.

☑ **Use include()** → When you want to **embed another JSP's content** into the current page.

Best method among request.getRequestDispatcher() and context.getRequestDispatcher() methods:

Feature	request.getRequestDispatcher()	context.getRequestDispatcher()
Path Type	**Relative to current request**	**Absolute within the app**
Best for	**Forwarding to a file within the same folder**	**Accessing any file in the app**
Security	**Cannot access /WEB-INF directly**	☑ **Can access /WEB-INF**
Use Case	**Forwarding to a page in the same directory**	**Global forwarding (e.g., /WEB-INF)**

☑ **Use request.getRequestDispatcher() when:**

- The file is **in the same directory** or **subdirectory**.
- You need a **relative** path.

☑ **Use context.getRequestDispatcher() when:**

- You need an **absolute** path starting from the web application root (/).
- You want to forward to files inside /WEB-INF, which is not directly accessible via URL.

Both methods **get a RequestDispatcher object** to forward or include a request, but they have key differences. The **best method depends on the use case**.

1. **request.getRequestDispatcher(String path)**

☑ **Relative to the current request (JSP/Servlet)**

☑ The path **must be within the same web application**

☑ Cannot access resources **outside the current request path**

RequestDispatcher dispatcher = request.getRequestDispatcher("next.jsp");

dispatcher.forward(request, response);

Here, next.jsp is relative to the current JSP/Servlet's location.

2. **context.getRequestDispatcher(String path)**

☑ **Absolute path within the web application**

☑ Can **access resources from anywhere** in the app

☑ Starts from the root / of the web application

RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/WEB-INF/pages/next.jsp");

dispatcher.forward(request, response);

Here, /WEB-INF/pages/next.jsp is an absolute path inside the app.

Conclusion:

☑ **context.getRequestDispatcher() is more powerful** as it allows access to any resource **anywhere in the app**, especially **inside /WEB-INF**.

☑ **Use request.getRequestDispatcher() when forwarding within the same directory**, and **use context.getRequestDispatcher() for more flexibility**.

2. <jsp:forward>

Example:

input.jsp

output.jsp

Cookies in JSP:

📌 What is a Cookie?

A **cookie** is a small piece of data **stored on the client's browser** by the web server. Cookies help in **maintaining user state** across multiple HTTP requests because **HTTP is stateless** (i.e., each request is independent).

📌 Why Are Cookies Used in JSP?

- **Remember User Preferences** – Store language settings, theme preferences, etc.
- **Session Tracking** – Maintain login state across multiple pages.
- **User Identification** – Recognize returning users.
- **Shopping Carts** – Store cart items before checkout.

📌 Types of Cookies

Type	Stored In Browser?	Expires When?	Set Using
Session Cookie	❌ Only in memory (RAM)	When the browser is closed	setMaxAge(-1) or no expiration set
Persistent Cookie	☑ Stored on disk	After a specified time	setMaxAge(seconds)

Session Cookies:

Example:

setSessionCookie.jsp

getSessionCookie.jsp

Persistant Cookies:

Example:

setCookie.jsp

getCookie.jsp

Summary of Cookie Methods:

Method	Description
new Cookie(name, value)	Creates a cookie with a name and value
response.addCookie(cookie)	Sends the cookie to the client's browser
request.getCookies()	Retrieves all cookies sent by the client
cookie.getName()	Gets the name of the cookie
cookie.getValue()	Gets the value of the cookie
cookie.setMaxAge(seconds)	Sets the expiration time of the cookie
cookie.setPath(path)	Defines where the cookie is accessible
cookie.setMaxAge(0)	Deletes the cookie

Conclusion:

✓ **Cookies help maintain user state in JSP**.

✓ **Persistent cookies store long-term data**, while **session cookies expire when the browser closes**.

✓ **Use setMaxAge(seconds) for expiration**, or **setMaxAge(-1) for session-only cookies**.

✓ **Use request.getCookies() to retrieve cookies and setMaxAge(0) to delete cookies**.

Sessions in JSP:

📌 What is a Session in JSP?

A **session** in JSP is a way to store **user-specific data** across multiple requests. Unlike cookies, session data is stored **on the server** rather than on the client's browser.

Key Features of JSP Sessions

- **Session data is stored on the server** (more secure than cookies).
- **Each user gets a unique session ID** (stored in a session cookie).
- **Session lasts until the user closes the browser** (or timeout occurs).
- **Ideal for storing sensitive data** (e.g., user authentication, shopping cart).

📌 How JSP Sessions Work?

1. **User requests a JSP page** → A **new session is created**.
2. **Session ID (JSESSIONID) is sent as a cookie** to the browser.
3. **User makes another request** → The server identifies the user by **retrieving the session ID**.
4. **Session data remains available** until the session expires or is invalidated.

Summary of JSP Session Methods:

Method	Description
session.setAttribute("key", value)	Stores data in session
session.getAttribute("key")	Retrieves data from session
session.removeAttribute("key")	Removes a specific attribute
session.invalidate()	Destroys the session completely
session.setMaxInactiveInterval(seconds)	Sets session timeout
<%@ page session="false" %>	Disables session in a JSP page

Conclusion:

✓ Sessions store user data securely on the server.

✓ Use session.setAttribute() and session.getAttribute() to manage session data.

✓ Use session.invalidate() to delete the session.

✓ Sessions are ideal for user authentication, shopping carts, and session tracking.

Examples:

setSession.jsp

getSession.jsp

removeSession.jsp

disableSession.jsp

Example for Session Timeout:

setSessionTimeout.jsp

getSessionTimeout.jsp

checkSession.jsp

sessionTimeout.jsp

Difference Between Cookies & Sessions:

Feature	Cookies	Sessions
Storage Location	**Client-side (browser)**	**Server-side**
Security	**Less secure**	**More secure**
Data Lifetime	**Based on expiration time (setMaxAge())**	**Until user closes browser or timeout occurs**
Used For	**Remembering user preferences**	**User authentication, shopping carts**

Disable the caching on the back button of a particular browser:

To prevent users from accessing a JSP page after logging out (or after submitting a form), we need to **disable caching**. This ensures that when the user presses the **Back** button, the browser does not load the page from its cache but requests a fresh copy from the server.

login.jsp

welcome.jsp

logout.jsp

Error Handling & exceptions in JSP:

1. **Try-Catch Block for Exception Handling**
   - **Purpose**: Wrap potentially risky code in a try-catch block to catch specific exceptions and handle them appropriately.
   - **Usage**: This technique is used when you want to catch and handle specific exceptions within a JSP (like division by zero, file I/O errors, etc.).

     trycatch
2. **Accessing Exception Details in the Error Page**
   - **Purpose**: The exception that occurred in the JSP page is stored as a request attribute (jakarta.servlet.error.exception). This allows you to fetch and display the exception details (e.g., error message, stack trace) on the error page.
   - **Usage**: This is beneficial when you want to display detailed error information for debugging purposes or to give users insight into the error that occurred.

     pagecontext
3. **Error Page Directive (**errorPage**)**
   - **Purpose**: The errorPage directive specifies the URL of a JSP page that will handle exceptions for the current JSP page. When an error occurs in a JSP, the user will be redirected to this error page.
   - **Syntax**: <%@ page errorPage="errorPage.jsp" %>
   - **Usage**: This is useful when you want a custom page to handle errors, rather than displaying a generic error message.

     index.jsp **errorPage.jsp**
4. **Global Error Handling via** web.xml
   - **Purpose**: You can define custom error pages for different HTTP status codes like 404 (Not Found), 500 (Internal Server Error), etc., globally across your entire application in the web.xml file.

- **Usage**: This is helpful when you want a global error handling strategy for all JSPs in your web application.

```
<error-page>
    <error-code>500</error-code>
    <location>/errorPage.jsp</location>
</error-page>
```

Best Practices for Error Handling in JSP*:*

☑ **Avoid exposing sensitive information**: Be careful not to display sensitive information (e.g., database details, passwords) in the error messages shown to end-users.

☑ **Custom error messages**: Design user-friendly error messages that don't overwhelm the user but provide enough information to understand the issue.

☑ **Logging**: For critical errors, consider logging the exception details to a file for future analysis without exposing them to the user.

☑ **Graceful error recovery**: Instead of showing a generic error page, consider providing options for the user to continue browsing or contact support.

Summary Table: Error Handling & Exceptions in JSP

Technique	Description	Code Example	Usage Scenario
Page-Level Error Handling	Use the errorPage directive to specify a page to handle exceptions when they occur.	<%@ page language="java" contentType="text/html; charset=ISO-8859-1" errorPage="errorPage.jsp" %>	Used to specify a fallback error page when an exception occurs in the current JSP.

Exception Handling (try-catch)	Wrapping risky operations (like division by zero) inside try-catch blocks to manage	jsp
		<%
		 try {
		 int result = num1 / num2;
		 } catch (ArithmeticException e) {
		 out.println(" Error: Division by Zero! ");
		 }
		%>
			Used when you want to handle specific exceptions directly in the JSP (for

	exceptions explicitly.		custom error messages).
Global Error Handling (web.xml)	Define error pages globally for specific error codes (like 404 or 500) in the web.xml configuration file.	xml <error-page> <error-code>500</error-code> <location>/errorPage.jsp</location> </error-page>	Used to specify a global error page for common HTTP error codes (like 404 or 500), which applies to all JSPs in the application.

Exception Object (request attribute)	Retrieve and display the exception object from the request attribute in the error page.	jsp
		 Error Details: <%= request.getAttribute("jakarta.servlet.error.exception") %>
		<pre>
		 Throwable throwable = (Throwable) request.getAttribute("jakarta.servlet.error.exception");
		 throwable.printStackTrace(new java.io.PrintWriter(out));</pre>
			Used to capture and display exception details on the custom error page. It is particularly useful when you want to display specific exception information or a stack trace for debugging purposes.

Note:

You **can use the exception implicit object** in any JSP page, but it is only available when an error occurs and is forwarded to an error page, or when the error handling mechanism is triggered.

When is the exception Implicit Object Available?

- The exception implicit object is **only available** when an exception occurs during the request processing **and** the request is forwarded to an error page.

- It will be **null** or **unavailable** if:
  - No exception occurred.
  - The page is not an error page.
  - The exception handling mechanism is not invoked (for example, if no errorPage directive is defined or no error is caught in the page).

Comparison of <jsp:forward> **and** response.sendRedirect()

Both <jsp:forward> and response.sendRedirect() are used to redirect a user or forward a request to another resource in a web application, but they work in different ways and serve different purposes. Below is a detailed comparison between the two:

Aspect	<jsp:forward>	response.sendRedirect()
Purpose	Forwards the request and response from one JSP or servlet to another resource (e.g., another JSP, servlet, or HTML page) within the same server.	Redirects the client's browser to a different URL (can be within the same server or to an external URL). This sends a new request from the client to the server.
Scope of Operation	The request and response are passed on to the new resource, allowing it to handle the request. The request is not terminated.	The request terminates on the server side, and a new HTTP request is sent by the browser to the new URL. It's a client-side redirection.
Server or Client-side?	Server-side operation. The control remains on the server, and the client is unaware of the forwarding.	Client-side operation. The client is aware of the redirection and sends a new HTTP request to the server.
Request and Response	The original request and response objects are passed along to the forwarded resource, so the forwarded resource can access the original data.	The original request and response objects are not passed. A new request is sent to the new URL, and the new resource does not have access to the original request or response objects.
URL Visibility	The URL in the browser's address bar does not change, as it is a server-side forward (transparent to the user).	The URL in the browser's address bar is updated to the new URL, as it is a client-side redirection.
Performance	More efficient as it doesn't involve a round-trip to the client; it stays within the server.	Less efficient, as it involves a round-trip to the client and then back to the server with the new request.
Use Case	Used for internal resource forwarding, like when a JSP	Used when redirecting the user to a completely new URL, either on the same server or to a different

	needs to forward to another JSP or servlet. It's often used to share the same request data.	website. Commonly used for login/logout or after processing form submissions.
Redirect Status Code	Does not change the HTTP status code unless explicitly set by the server (normally status code 200).	Sends a 302 (Found) status code to the client, signaling that the browser needs to send a new request to the new location.
Session Handling	The session remains intact when forwarding between resources within the same web application.	Session remains intact when the user is redirected, as long as cookies (session cookies) are preserved during the redirect.
Syntax	<jsp:forward page="somePage.jsp" />	response.sendRedirect("somePage.jsp");

When to Use Each:

1. **<jsp:forward>**:
   - Use when you want to forward the request and response to another resource **internally** (within the same server).
   - The user is unaware of the redirection, and it's typically used to forward to another JSP or servlet.
   - Commonly used in **request dispatching** where the same request and response need to be passed along to the new resource for further processing.
2. **response.sendRedirect()**:
   - Use when you need to **redirect the client's browser** to a different resource (either on the same server or on an external server).
   - The user will see the URL change in the browser's address bar.
   - Commonly used after form submissions, for example, after successfully logging in or logging out, or redirecting to a different page based on user input.
   - Can be used to **redirect to a different domain**.

Example Usage:

1. <jsp:forward>:

```
<%-- Forward the request to another JSP page --%>
<jsp:forward page="success.jsp" />
```

2. response.sendRedirect():

```
// Redirect to another page
response.sendRedirect("https://www.example.com");
```

Key Differences:

1. **Visibility**:
   - <jsp:forward> keeps the same URL in the browser, whereas response.sendRedirect() changes the URL.
2. **Request Passing**:
   - With <jsp:forward>, the request and response are passed to the next resource, while response.sendRedirect() sends a new request from the browser to the new resource.
3. **Client vs. Server-Side**:
   - <jsp:forward> is a server-side operation, while response.sendRedirect() is a client-side operation.
4. **Efficiency**:
   - <jsp:forward> is more efficient since it does not require a new request-response cycle, whereas response.sendRedirect() causes the client to make a new request.

Conclusion:

- Use **<jsp:forward>** when you need to forward a request internally without the user noticing and when you want to keep the request and response intact.
- Use **response.sendRedirect()** when you need the client to navigate to a different page, typically when dealing with different resources or external websites.

MVC architecture in JSP page:

◇ Model 1 Architecture (JSP-Centric Approach)

In **Model 1**, JSP pages handle **both presentation (UI) and business logic**.

How Model 1 Works:

1. **User requests a JSP page**.
2. **JSP page**:
   - Retrieves data from Java Beans.
   - Processes logic.
   - Generates the response.
3. **Response is sent back to the browser**.

Advantages of Model 1:

✓ **Quick and Easy Development** – Suitable for small applications.

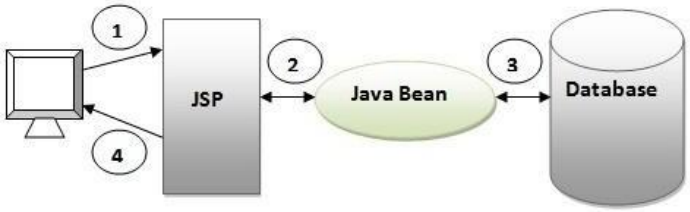
✓ **No need for separate Servlets** – Everything is handled in JSP.

Disadvantages of Model 1:

❌ **Tightly Coupled Code** – Hard to maintain as UI and logic are mixed.

❌ **Difficult to Extend** – Making changes requires modifying multiple JSPs.

❌ **Navigation Control Issues** – Each page determines its next step, leading to redundant code.



Explanation

1. **User Request** → The client (browser) sends a request to a JSP page.
2. **JSP Handles Logic** → The JSP page contains business logic and directly communicates with a **JavaBean** (if used).
3. **Database Interaction** → The JSP page or JavaBean fetches data from the database.
4. **Response to Client** → The JSP page processes the data and sends the response back to the client.

Characteristics of Model 1

- The **JSP page** acts as both the controller and the view.
- The **business logic** and **presentation logic** are mixed inside the JSP.
- **Simple structure**, suitable for small applications.
- **Harder to maintain** due to tight coupling between logic and UI.

◇ **Model 2 (MVC Architecture)**

Model 2 (MVC) is an **improved design**, where:

- **Model** = Handles business logic and data.
- **View** = Represents the UI using JSP.
- **Controller** = Processes user requests using Servlets.

How Model 2 (MVC) Works:

1. **User sends a request** via a browser.
2. **Controller (Servlet)** processes the request.
3. **Controller interacts with the Model** (JavaBeans, Database).
4. **Model returns data** to the Controller.

5. **Controller forwards the data to a JSP** (View).
6. **JSP renders the response** and sends it to the user.

Advantages of Model 2 (MVC)

✓ **Separation of Concerns** – UI, logic, and data are independent.

✓ **Scalability** – Easy to add new features.

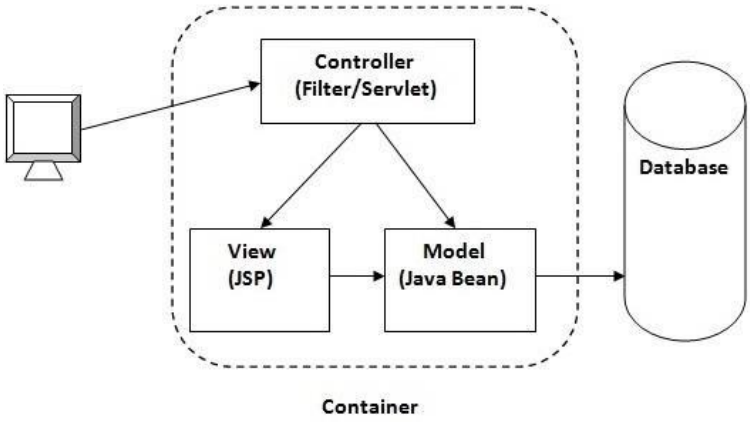
✓ **Reusability** – Business logic can be reused across multiple views.

✓ **Maintenance-Friendly** – Changes to UI do not affect logic.

Disadvantages of Model 2

❌ **Requires More Effort** – Developers need to write Servlets and manage request flow.

❌ **More Complex than Model 1** – Suitable for large applications.



Explanation

1. **User Request** → The client sends a request to a **Servlet (Controller)**.
2. **Controller Processes Request** → The **Servlet** acts as a controller, processing user input and deciding what to do.
3. **Model Handles Business Logic** → The **JavaBean (Model)** interacts with the database and processes data.
4. **View (JSP) Displays Data** → The controller forwards data to a **JSP page (View)**, which presents the result.

Characteristics of Model 2

- **Separation of Concerns**:
  - Servlet (Controller) → Handles request processing.
  - JavaBean (Model) → Manages data and business logic.
  - JSP (View) → Displays the results.

- **More maintainable** and **scalable**.
- **Recommended for larger applications**.

◇ **Comparison: Model 1 vs. Model 2:**

Feature	Model 1 (JSP-Centric)	Model 2 (MVC)
Code Organization	Mixed logic in JSP	Separate Model, View, and Controller
Scalability	Limited	High
Navigation Control	Decentralized	Centralized (in Controller)
Ease of Maintenance	Difficult	Easier
Reusability	Poor	High

JSP Programs and Examples:

☑ Write a JSP program to check if a given number is an Armstrong number or not.

☑ Write a JSP to generate the multiplication table of a given number.

☑ Write a JSP program to apply all arithmetic operations on 2 numbers (say a, b). Input has to be taken from a user form.

☑ Write a JSP program to find the gross salary of an employee where BASIC salary is taken as input from the user. Use the following rules to compute the gross salary: DA = 90% of BASIC, HRA = 10% of BASIC, GROSS = BASIC + DA + HRA.

☑ Write a JSP page to display the current date and time.

☑ Write a JSP program to print the first n numbers with their squares in a table.

☑ Write a JSP program to find the largest of three numbers.

☑ Write a JSP program to find the factorial of a given number.

☑ Write a JSP program to check if a given number is prime or not.

☑ Develop a code snippet to implement a hit counter in JSP.